



AP Computer Science Principles

Python Programming Using Processing

What is Computer Programming?

Computer Hardware and Software: The Book Analogy

- Computer Hardware is like the cover and pages of a book.
- Computer Software is like the story—the language of the book.
- Programming a computer is creating software by writing computer language.

Computer Language

- Like human (also called "natural") language, Computer languages have words, numbers, grammar, syntax and punctuation.
- Unlike natural language, computer languages are very limited and specialized.
- Human language often has 100,000's of words, computer languages get by with a few hundred.
- In Python, these keywords are pretty much all you would ever need.

Low Level Computer Language

- Computers only "understand" sequences of numbers
- The "language of numbers" is called machine language—a low level language. It looks like

100011 00011 01000 000000000001000100

- Writing low level programs is difficult and tedious
- According to Wikipedia “Currently, programmers almost never write programs directly in machine code, because it requires attention to numerous details that a high-level language handles automatically.”

Low Level Computer Language

- Another example of a Low Level Computer Language is Assembly code.
- It is specific to the particular hardware architecture of a computer.
- Assembly code has a limited number of commands.

```
fib:  
    mov edx, [esp+8]  
    cmp edx, 0  
    ja @@  
    mov eax, 0  
    ret  
  
@@:  
    cmp edx, 2  
    ja @@  
    mov eax, 1  
    ret  
  
@@:  
    push ebx  
    mov ebx, 1  
    mov ecx, 1  
  
@@:  
    lea eax, [ebx+ecx]  
    cmp edx, 3  
    jbe @@  
    mov ebx, ecx  
    mov ecx, eax  
    dec edx  
    jmp @@  
  
@@:  
    pop ebx  
    ret
```

High Level Computer Language

- Uses words, easier for humans.
- High level programs are written using a special program that is sort of like a word processor for computer language.
- When the code is finished, another program (called a compiler or interpreter) converts the high level code to assembly or machine code.
- Grace Hopper, who you will learn more about, invented the idea of a high level language.

High Level Languages

- C: flexible, powerful
- Javascript: good for the internet
- Java: Used for large enterprises.
- Python: easy to learn and use, many libraries.
- Scheme: good for artificial intelligence.
- Many Others: C#, C++, Basic, LISP, Fortran, Pascal, Objective C, Forth, Lua, COBOL, Logo.

Test Question from UCSD

- A former student had this question on a test in his 3rd computer science class at UCSD.
- The test asked him to convert the High Level C code on the left to Low Level Assembly code on the right.

Abstraction

- High Level computer languages are an example of abstraction
- We replace a complex set of instructions in low level code with simpler high level code
- Abstraction is reducing detail to manage complexity

Fourth Generation Languages

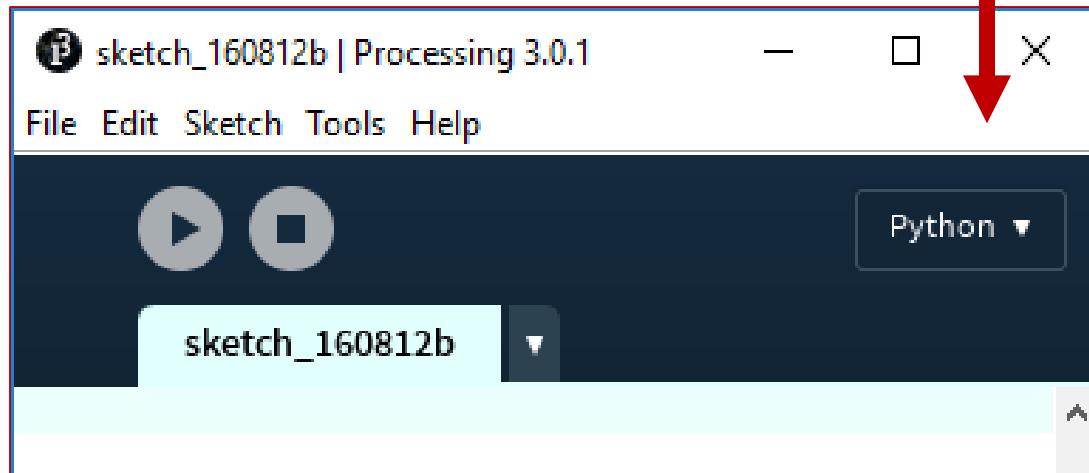
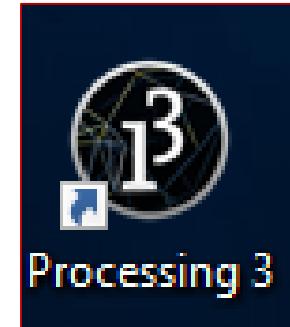
- Programmers are working to create language programs that can convert natural language to computer language.
- [Query](#) is one example.
- Still in the early stages, it will be some time in the future before they can be used to make useful programs.

Create Python Programs Using Processing

- Processing is a beginner friendly program for creating software.
- Think of it as word processing program for writing computer software.
- Very powerful graphics features.

Starting Our First Program

- Start Processing
- Make sure you are in **Python mode**.

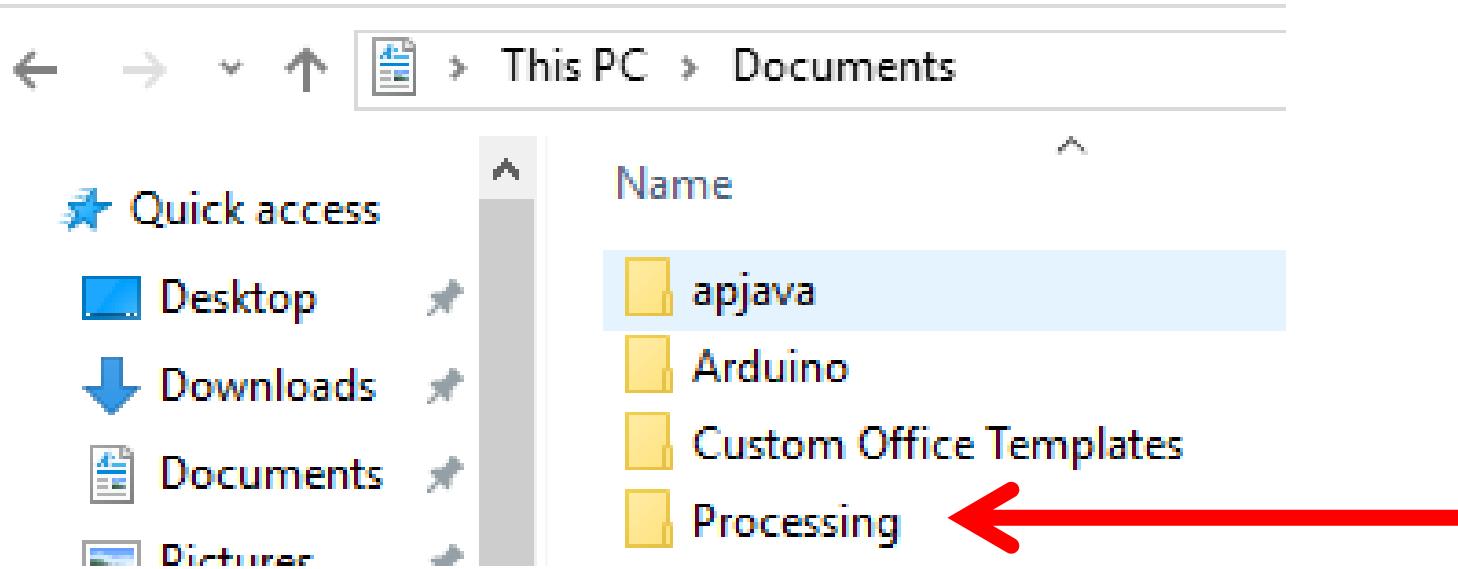


Starting Our First Program

- Choose *File / Save*
- Save your program as **OlympicRings**

Processing Programs are Saved in your Documents Folder

- By default, all your Processing programs will be saved in the Processing folder in your Documents folder.



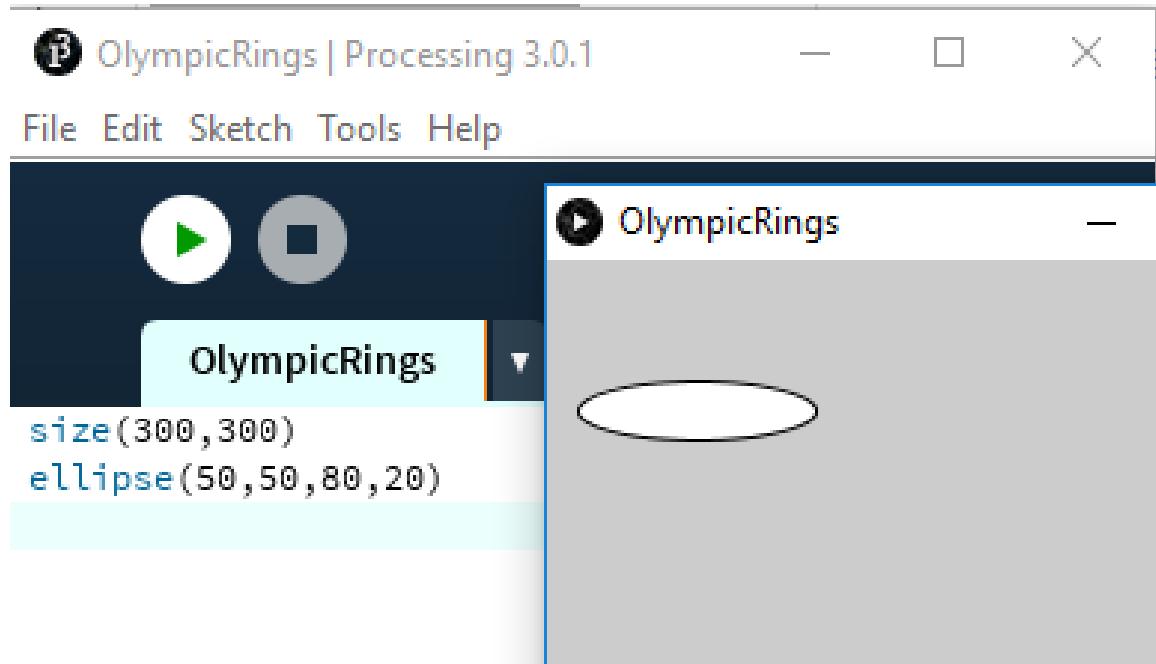
How to Make Your Own Computer Software

Type the following code:

```
size(300,300)
```

```
ellipse(50,50,80,20)
```

Then press the run button.



How to Make Your Own Computer Software

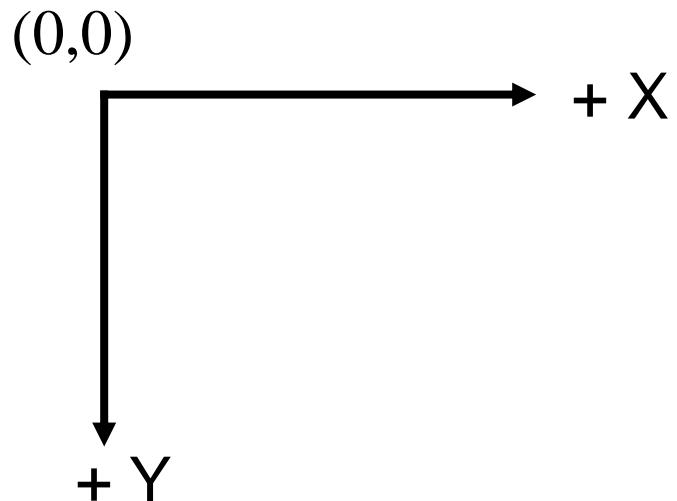
`size(300,300)`

`ellipse(50,50,80,20)`

- Two examples of a *function call*
- Functions are identified with `()`
- **this function call** “sets the size of the drawing canvas.”
- **This function call** “draws an ellipse with this size and position.”
- The numbers in the parenthesis are called *arguments or parameters*.

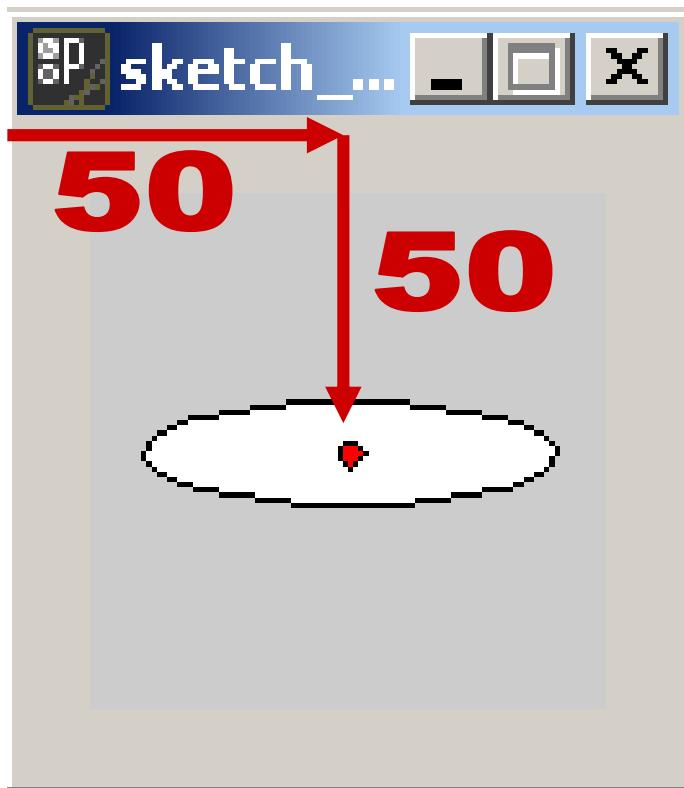
Coordinates in Computer Graphics

- In computer graphics, the coordinate system is only positive.
- The origin is the top left corner of the window



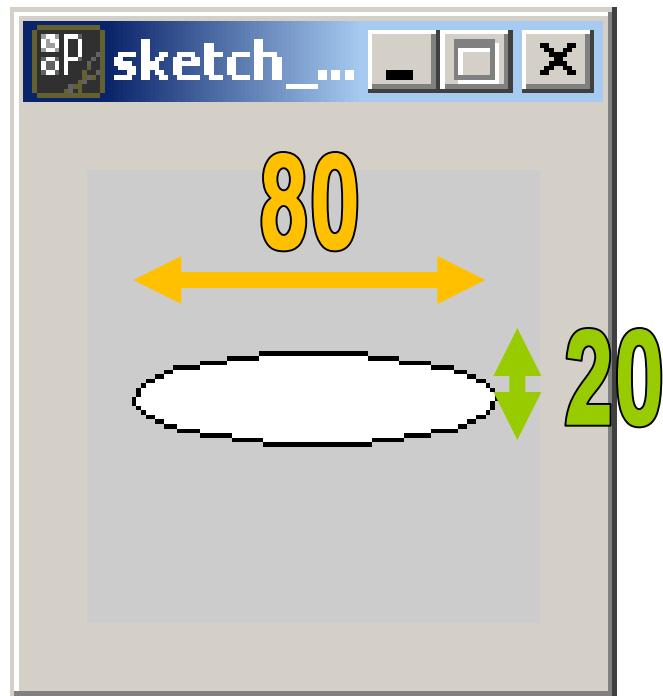
`ellipse(50,50,80,20)`

- The first two arguments (in Red) are the x and y coordinates of the center of the ellipse



ellipse(50, 50, 80, 20)

- The third argument (in Orange) is the width of the ellipse.
- The fourth argument (in Green) is the height
- If the width and height are the same, the result is a circle.



size(300, 200)

- The size function, should always be called first at the top of your program.
- This one increases (or decreases) the screen area (“canvas”).
- The two arguments are **width** and **height**.



How to Make Your Own Computer Software

```
noFill()
```

```
ellipse(50,50,80,20)
```

- A program can use any number of functions
- The noFill() function changes how ellipses are drawn
- Notice that noFill() has no arguments, it's parenthesis are empty



How to Make Your Own Computer Software

noFill()

smooth()

ellipse (50,50,80,20)

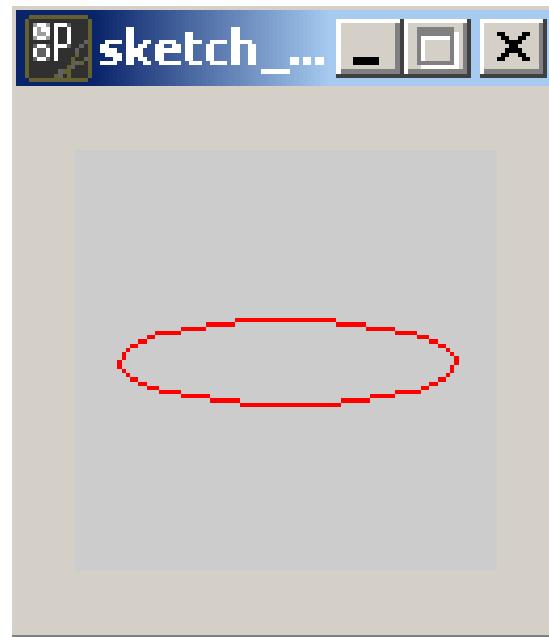
- **smooth()** also has no arguments.
- It smooths out the curves of the ellipse.

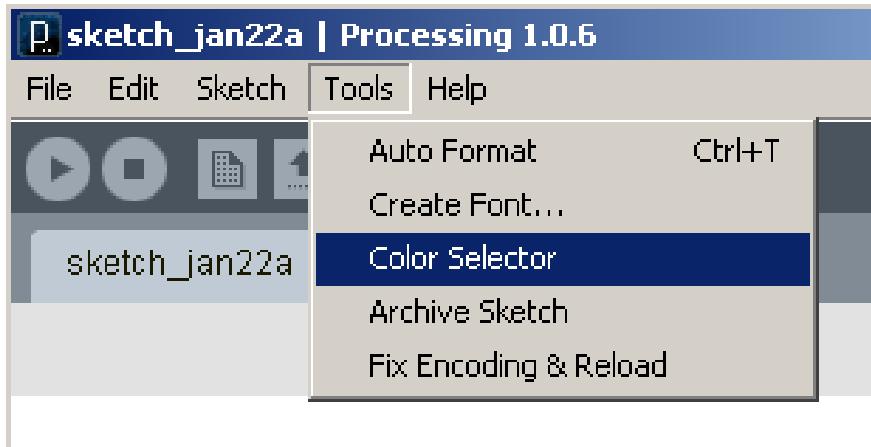


How to Make Your Own Computer Software

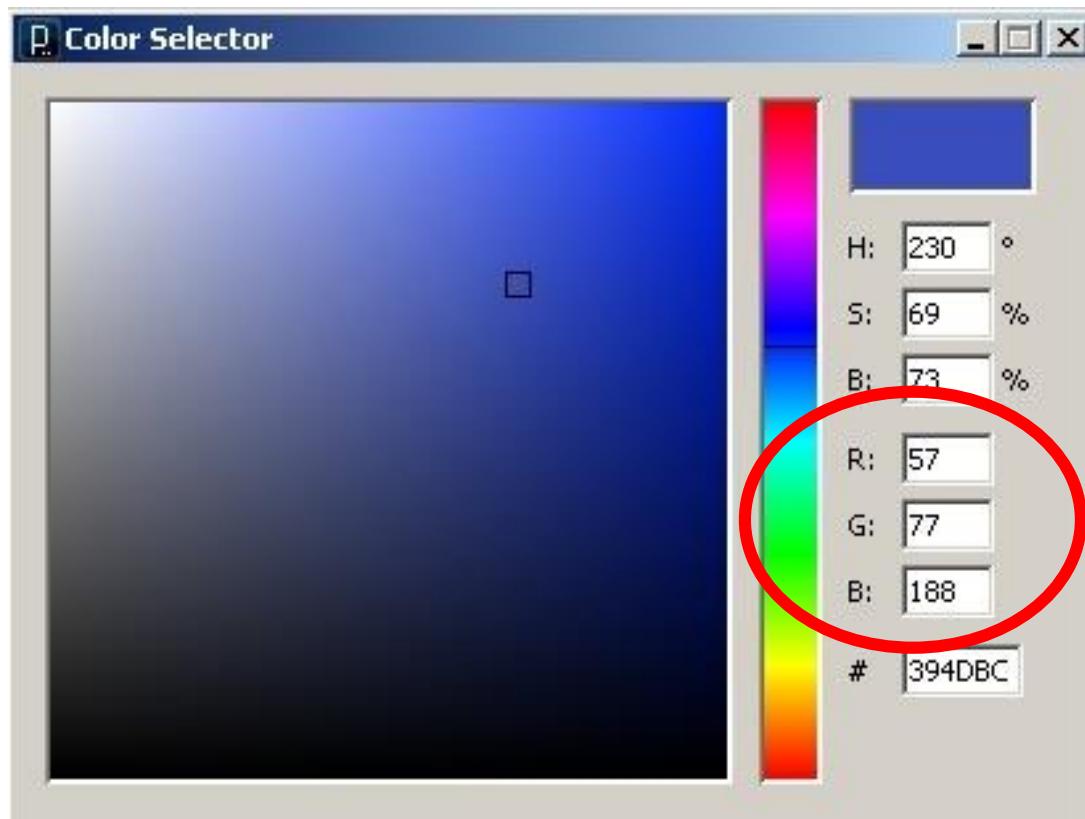
```
noFill()  
stroke(255,0,0)  
ellipse(50,50,80,20)
```

- The `stroke()` function changes the color of the outline.
- It's 3 arguments are the amount of Red, Green and Blue in the range 0 – 255.





The Color Selector:
click on the color you
want and read the RGB
values.



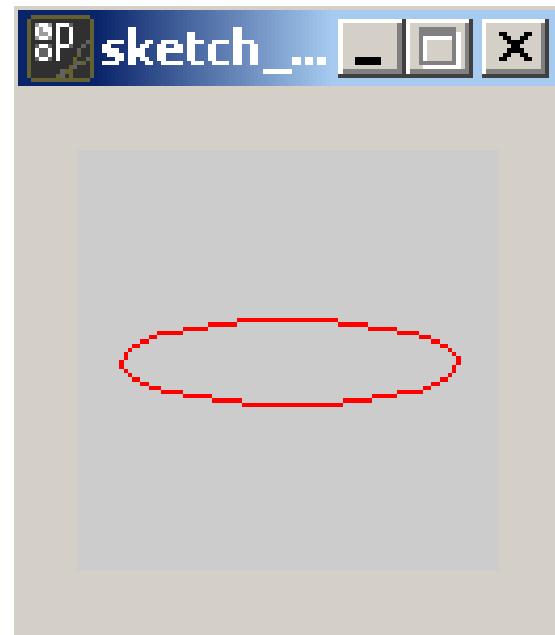


You can use a handheld microscope to see the RGB pixels in your computer monitor

How to Make Your Own Computer Software

```
noFill()  
stroke(255,0,0)  
strokeWeight(5)  
ellipse(50,50,80,20)
```

The **strokeWeight()** function
changes the thickness of the
outline



Sequence: Order is Important

```
ellipse(50,50,80,20)
```

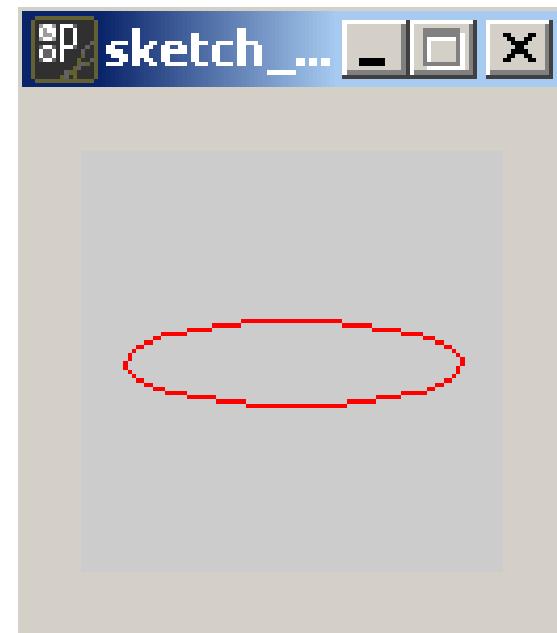
```
noFill()
```

```
stroke(255,0,0)
```

```
strokeWeight(5)
```

“Sequence” is the order of computer commands.

Here changing the stroke and fill has no effect, because it's done after the ellipse had already been drawn.

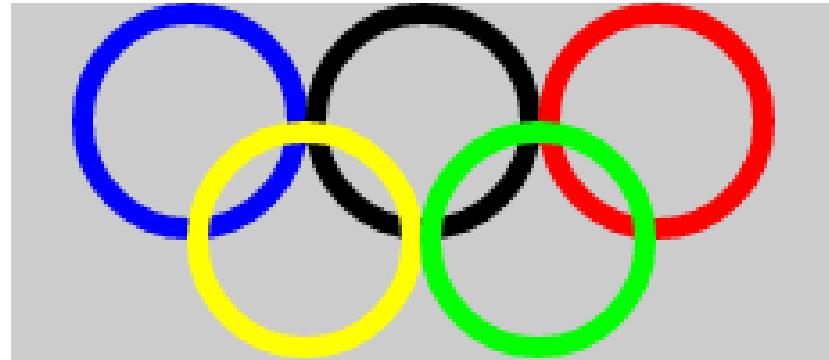


Assignment #1: Olympic Rings

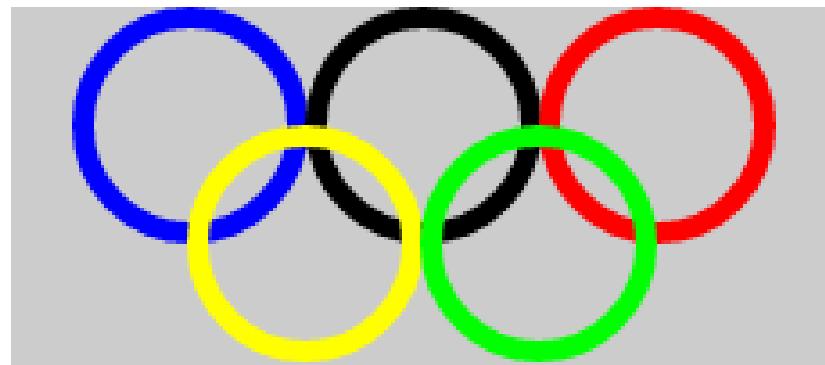
Write a program that produces a design *similar* to the Olympic Rings with 5 differently colored circles.

Your program will use the following functions (some more than once):

```
strokeWeight()  
stroke()  
ellipse()  
size()  
noFill()
```



Optional Challenge: Feel free to read about [other Processing functions](#) and add other elements to your Olympic design.

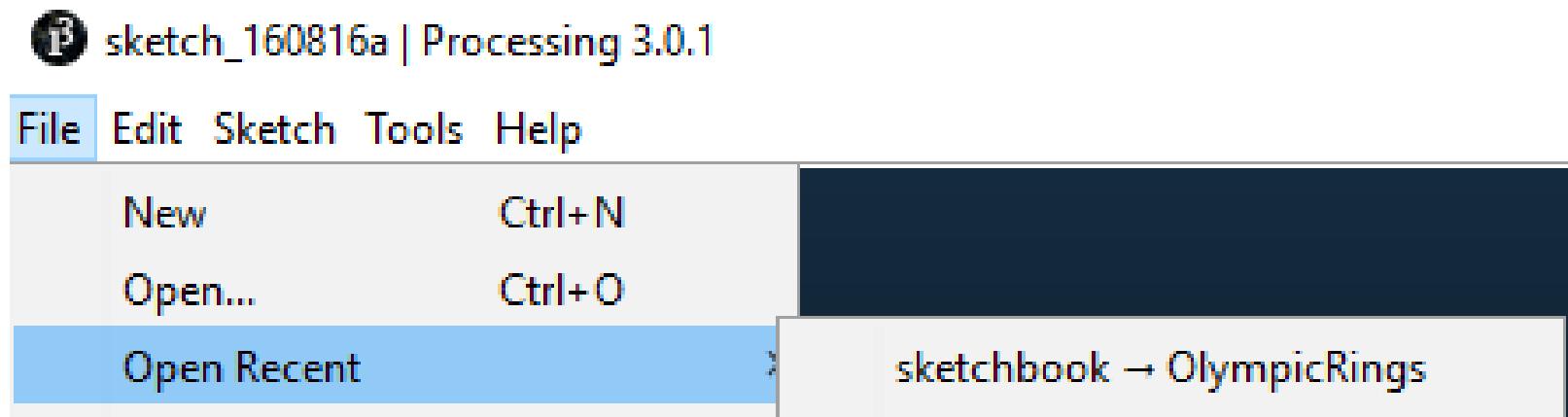


Olympic Rings: Common Mistakes

- Put each statement on a separate line.
- Once you call **noFill()**, from then on, all ellipses will be unfilled. Don't call it more than once.
- Same with **smooth()** and **strokeWeight()**. Don't call them more than once.

Opening Your Work from Yesterday

- One way is to start Processing and choose *File | Open Recent | sketchbook -> OlympicRings*



Submitting Your Assignment

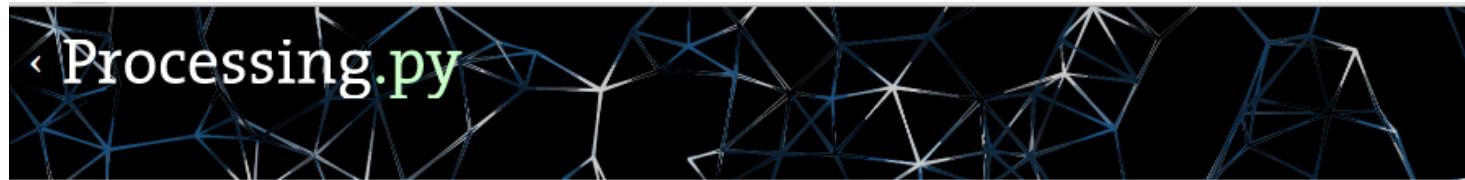
1. Name your program something like *OlympicRings* and save it.
2. Submit the program (the *.pyde* Python file).

Processing's Python "Dictionary"

<http://py.processing.org/reference/>

Sometimes this is called an **API***

An API is like a dictionary of a computer language



[Cover](#)
[Reference](#)
[Tutorials](#)
[Examples](#)
[Bugs](#)

Processing.py Reference. Processing is not a single programming language, but an arts-centric system for learning, teaching, and making visual form with code. This Reference documents its Python Mode.

Structure

(comment)
''' '' (multiline comment)
() (parentheses)
, (comma)
. (dot)

Shape

createShape()
2D Primitives
arc()
ellipse()

Color

Setting
background()
clear()
colorMode()

* API stands for *Application Programming Interface*, but you don't really care and it won't be on the test

The “Definition” of `ellipse()`



[Cover](#)

[Reference](#)

[Tutorials](#)

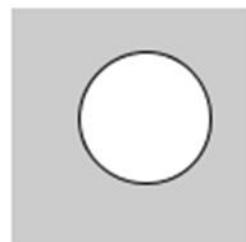
[Examples](#)

[Bugs](#)

Name

`ellipse()`

Examples



`ellipse(56, 46, 55, 55)`

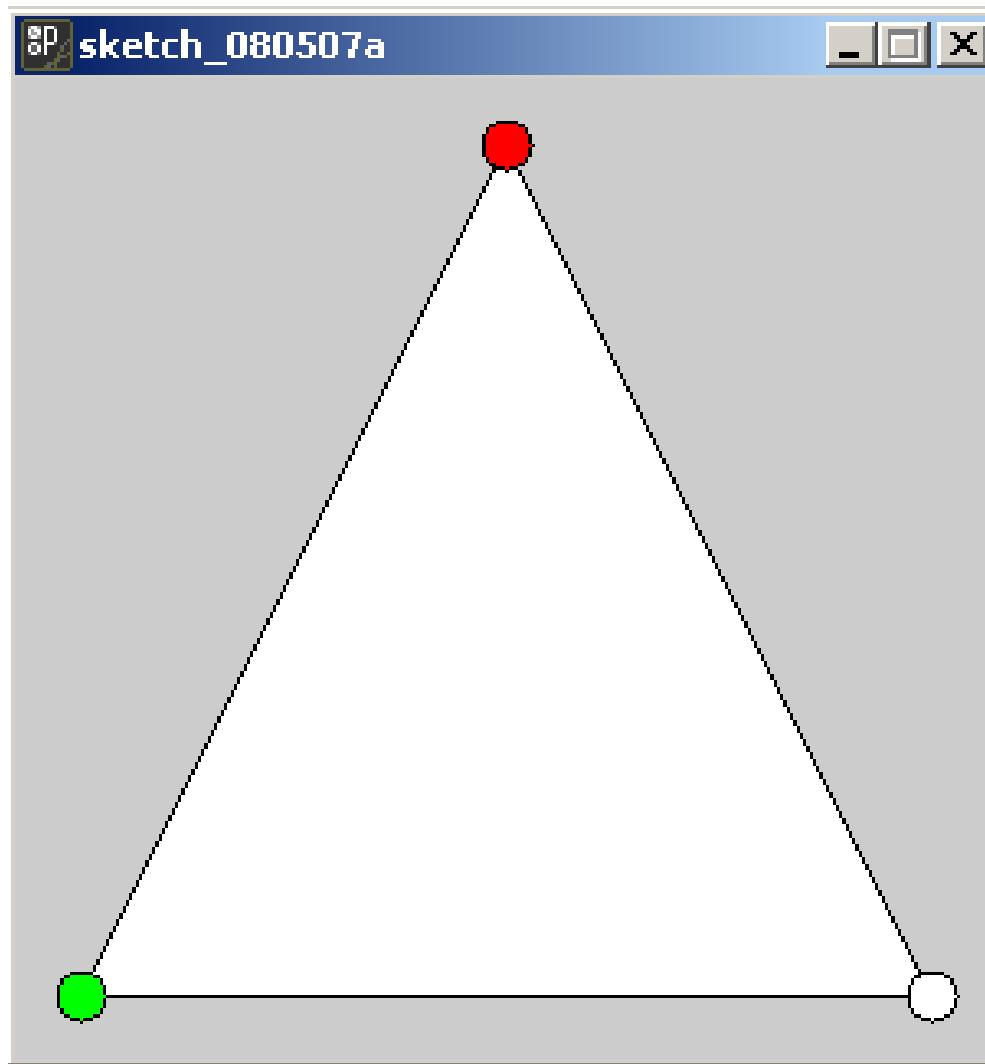
Description

Syntax `ellipse(a, b, c, d)`

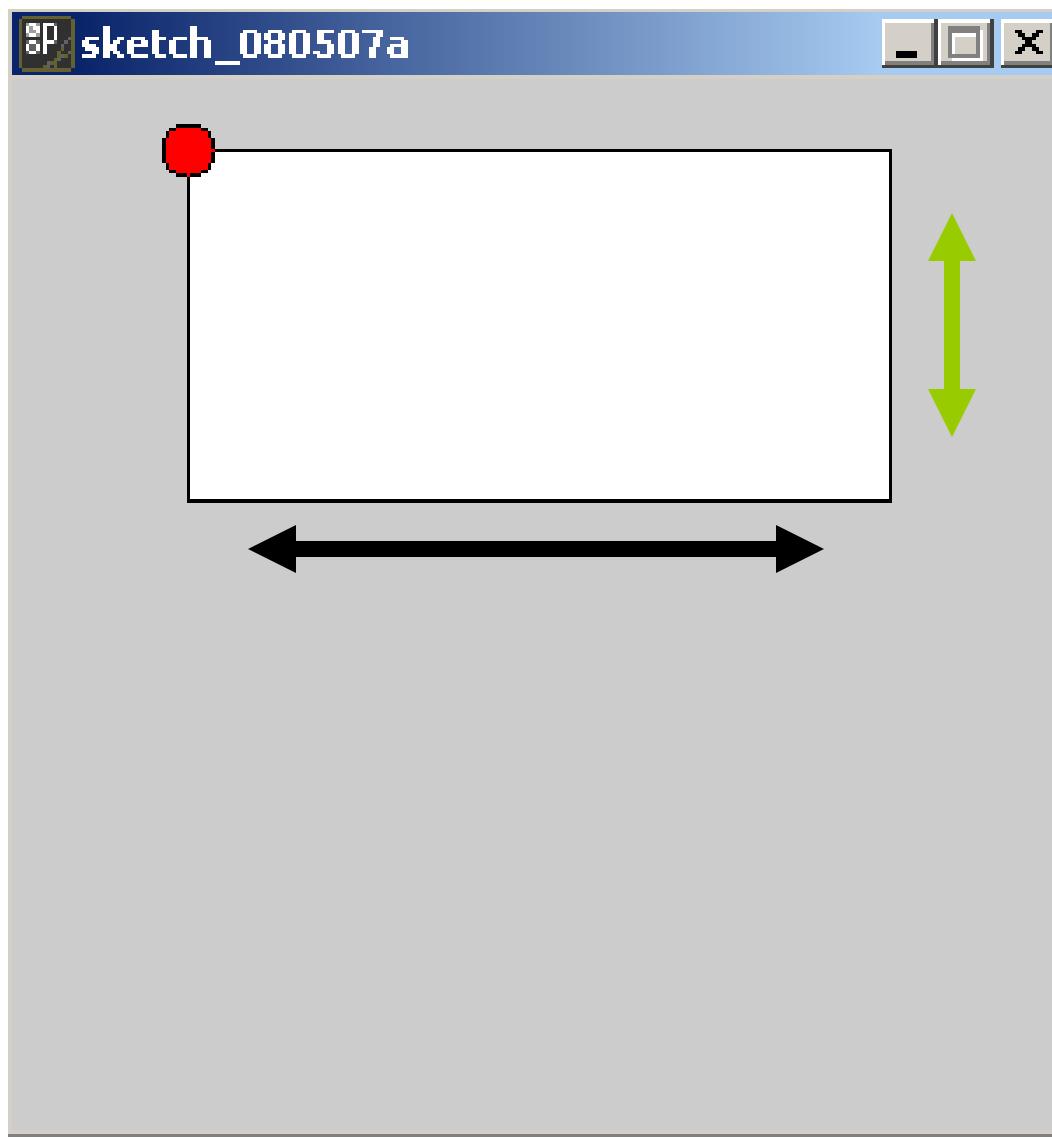
Drawing Functions

- `triangle()`
- `rect()` (also squares)
- `quad()`
- `ellipse()` (also circles)
- `point()`
- `line()`
- `bezier()` and `arc()` (for curves)
- `beginShape()` and `endShape()` (for polygons)

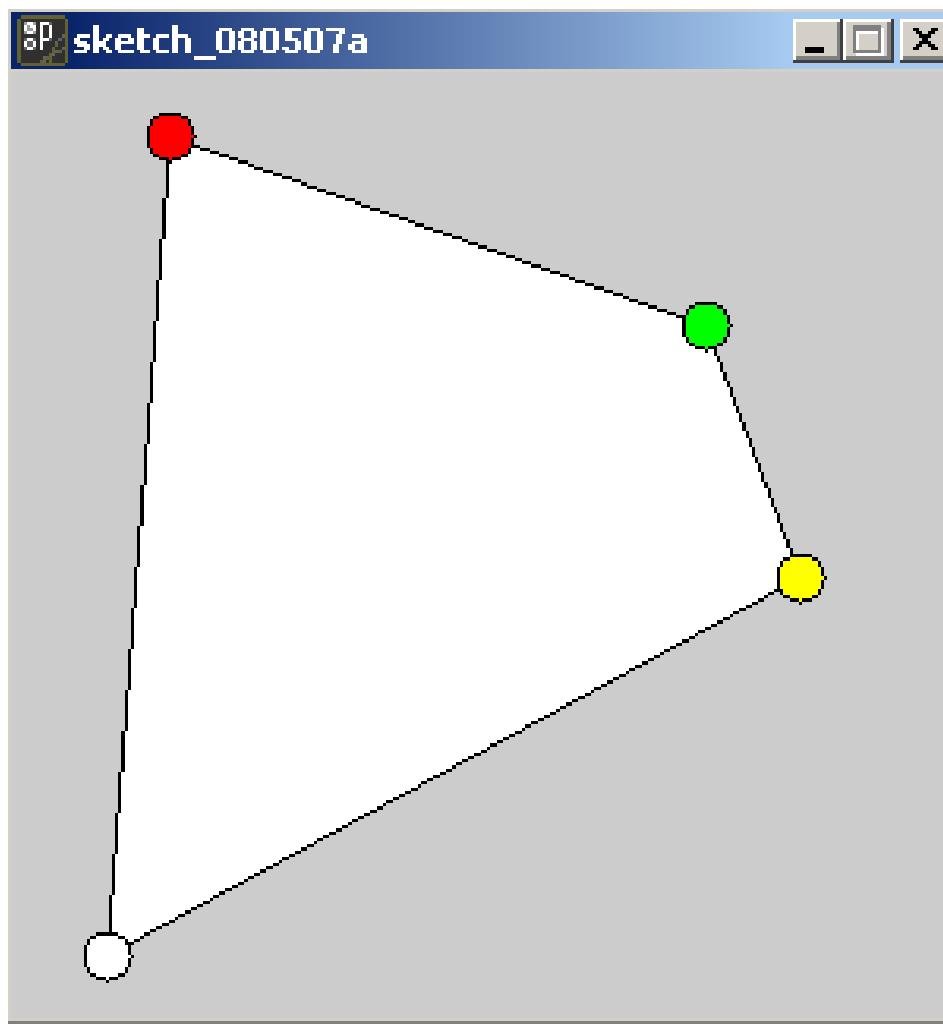
```
triangle(150,20,20,280,280,280)
```



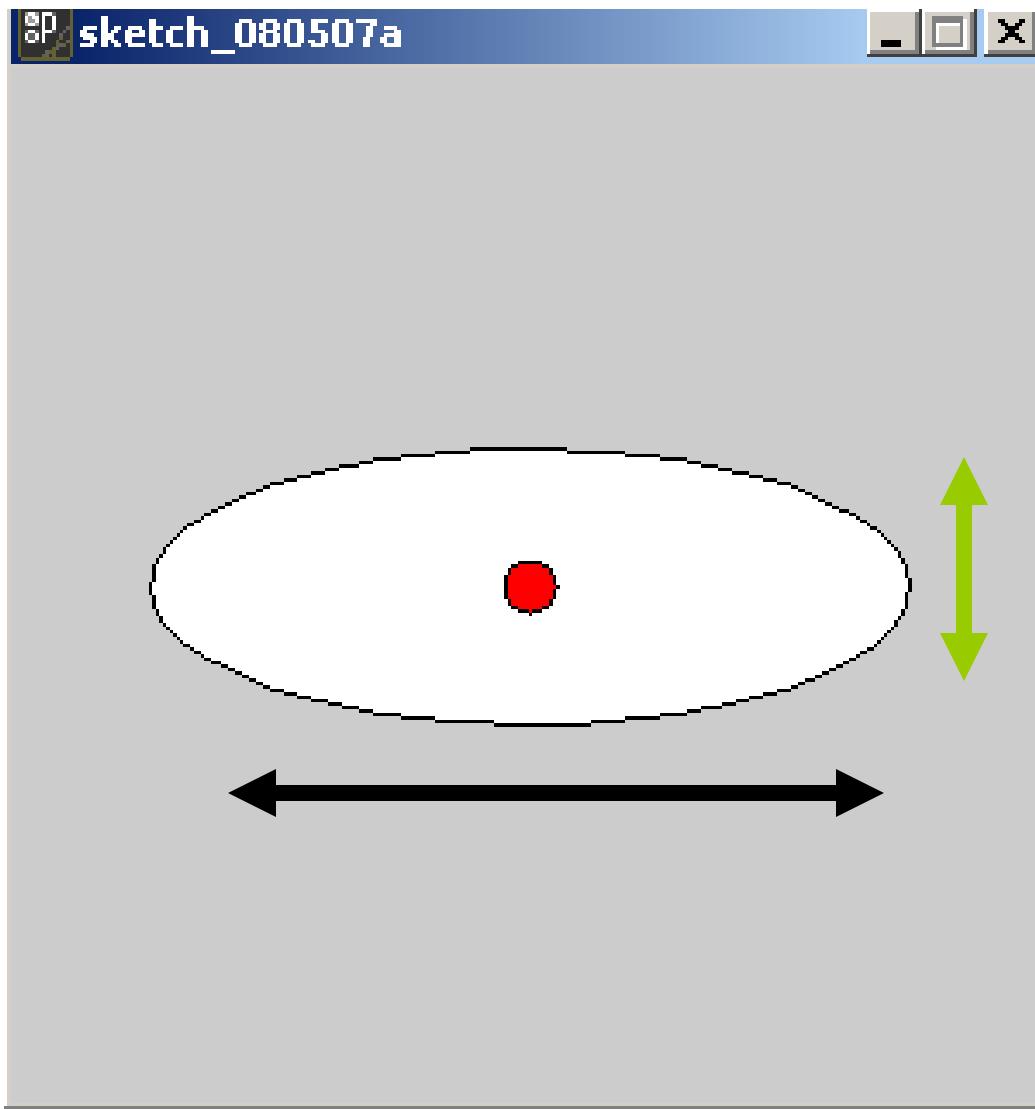
```
rect(50,20,200,100)
```



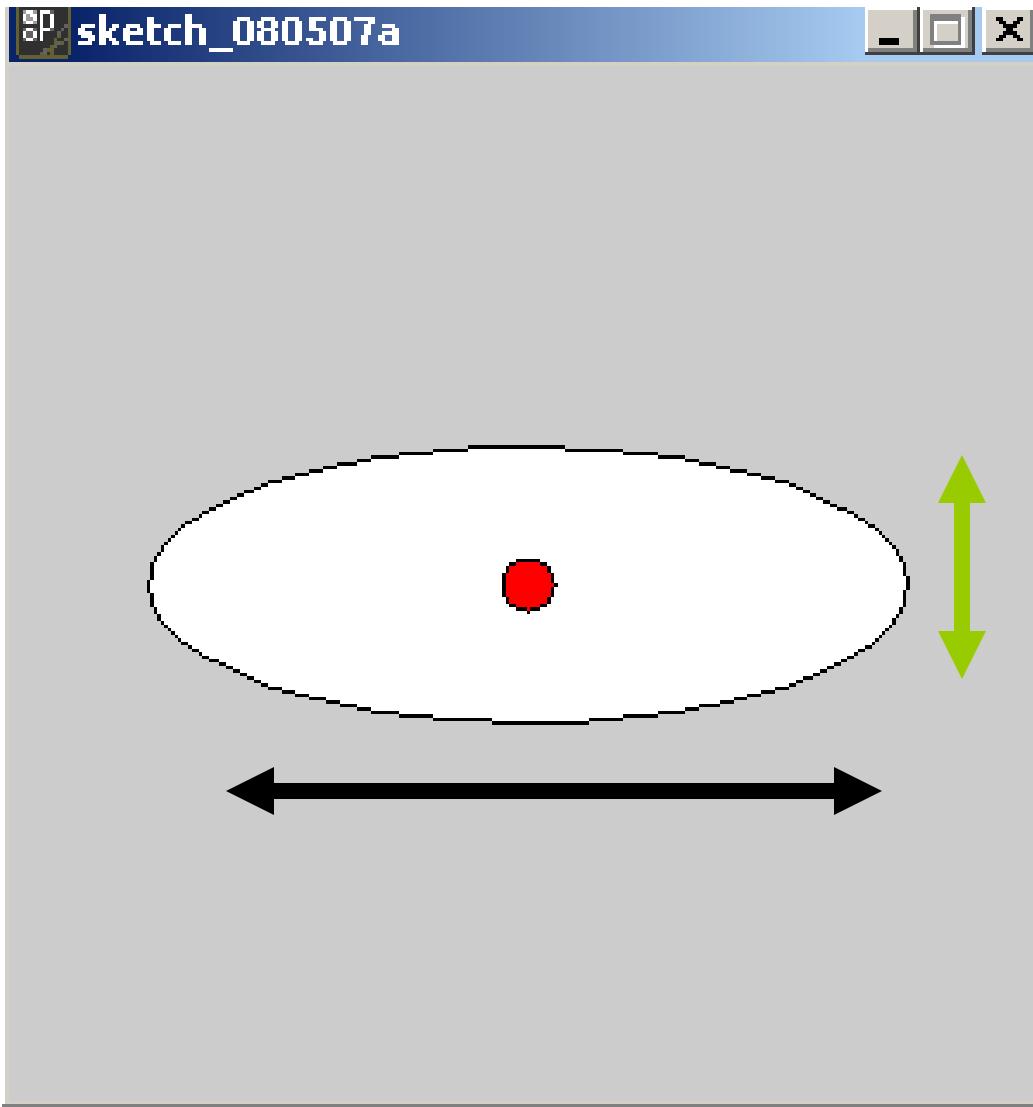
```
quad(50,20,220,80,250,160,30,28  
0)
```



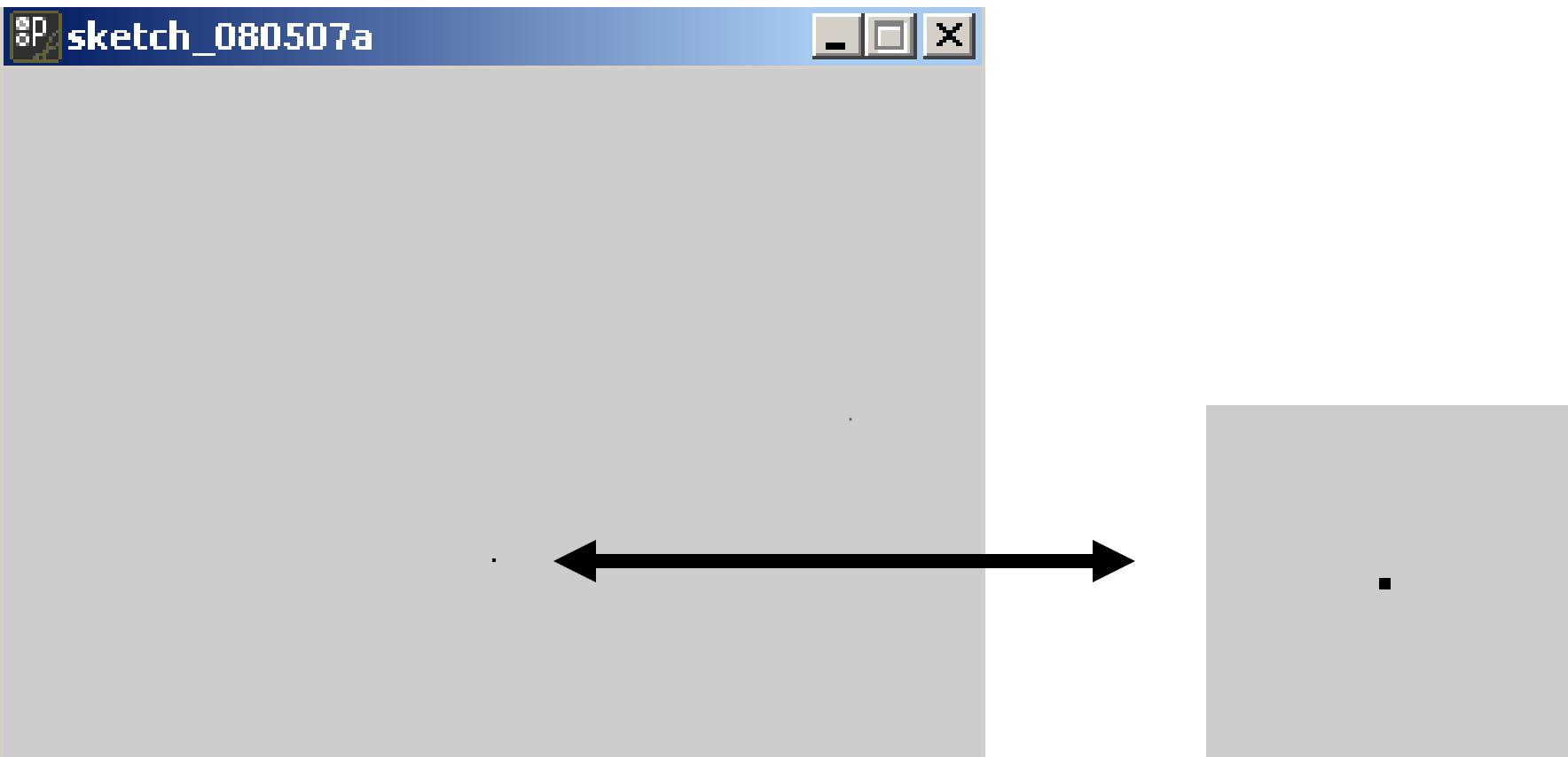
ellipse(150, 150, 220, 80)



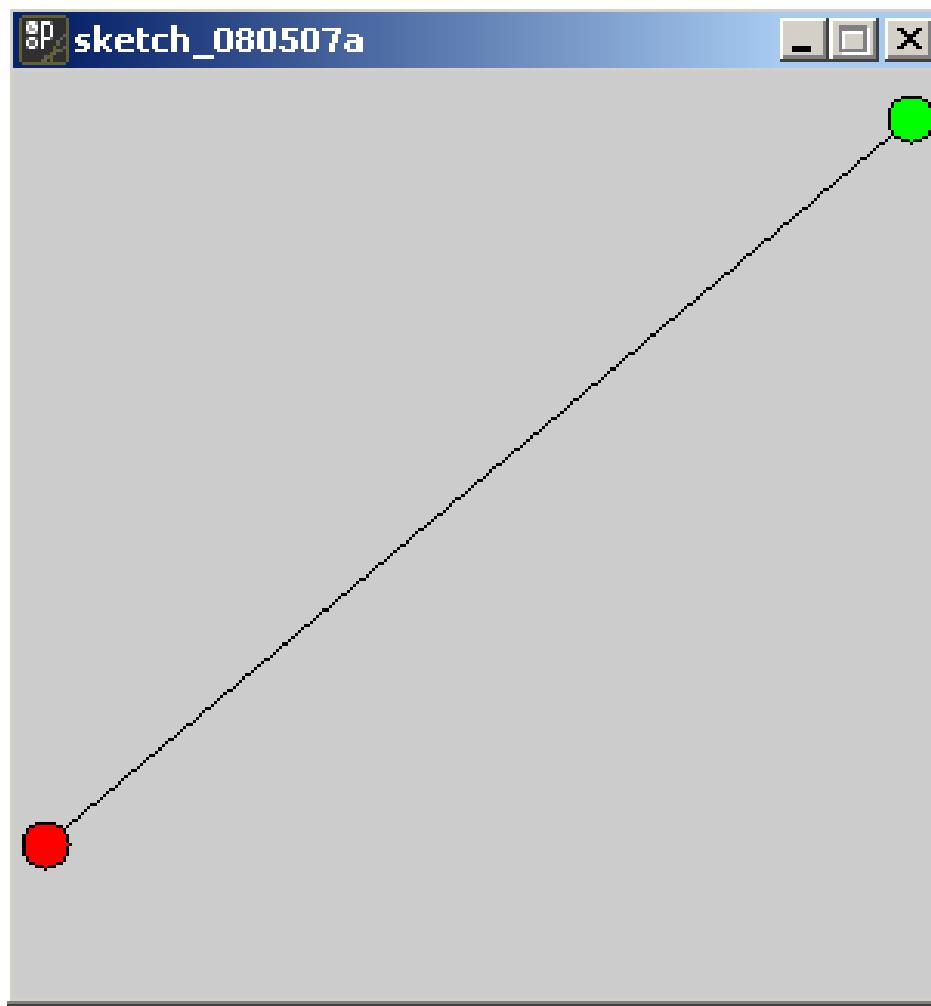
ellipse(150, 150, 220, 80)



`point(150,150)`

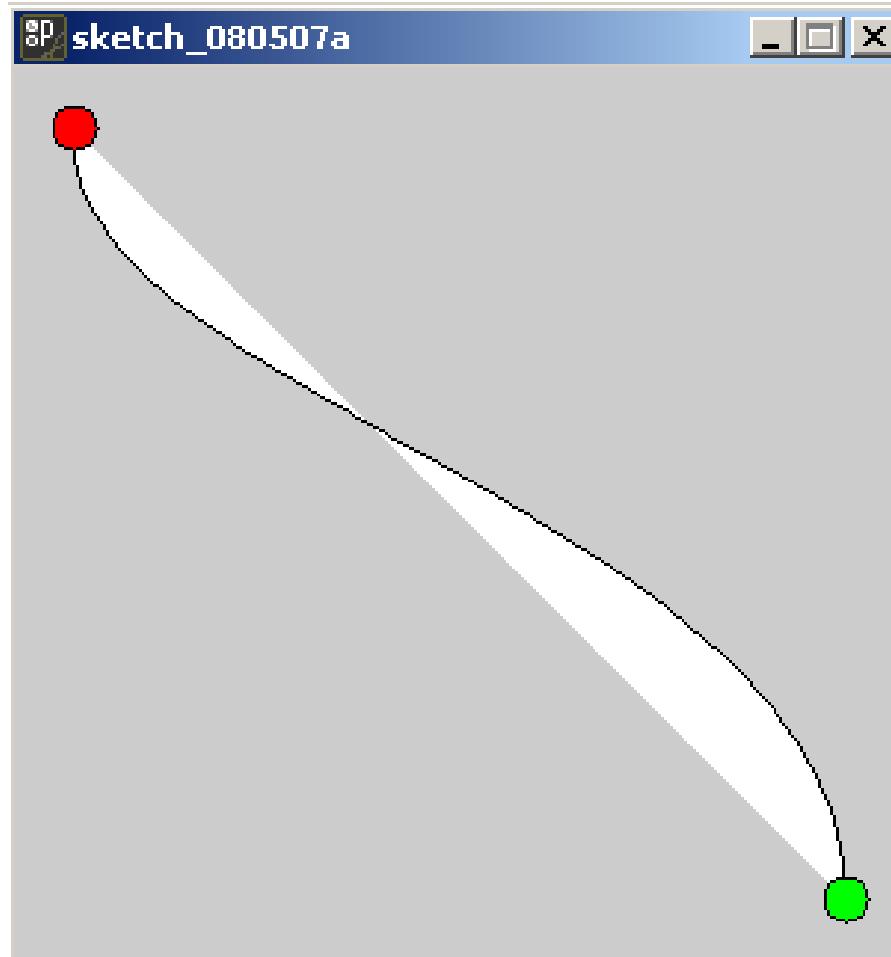


```
line(10,250,290,15)
```



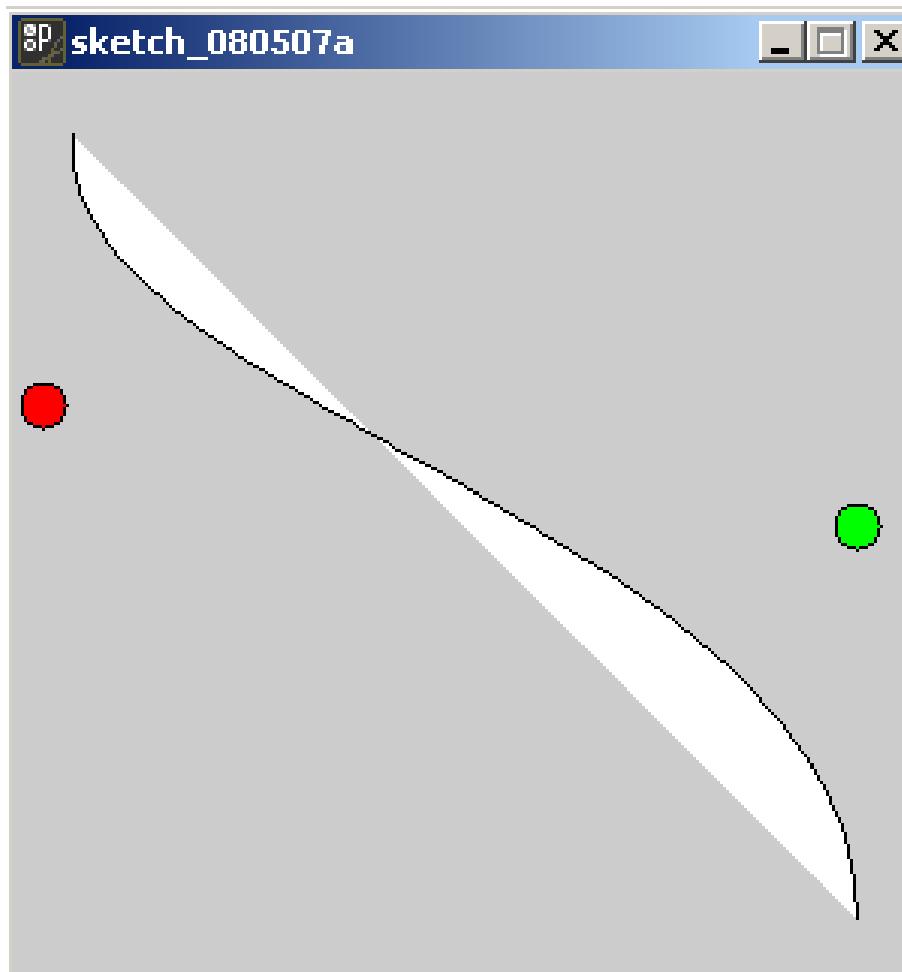
The Endpoints

```
bezier(20,20,10,110,280,150,280,280)
```



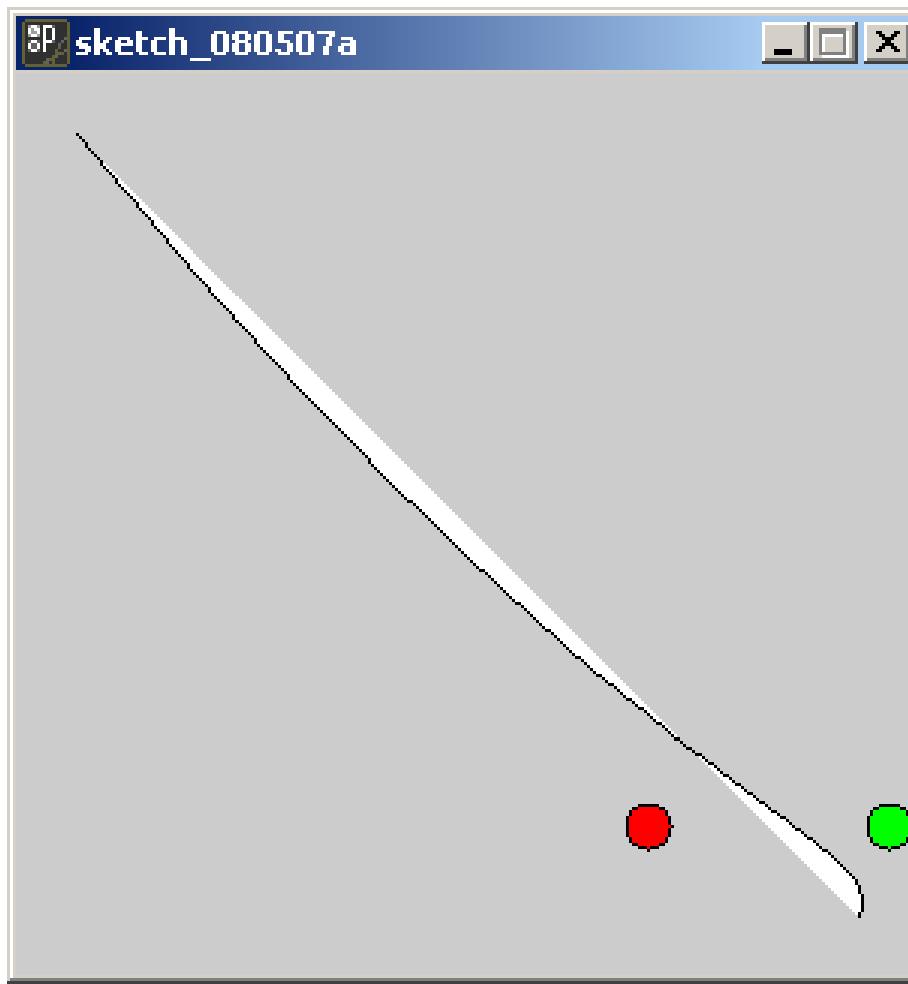
These Two Invisible Points “Pull” the Curve

```
bezier(20,20,10,110,280,150,280,280)
```



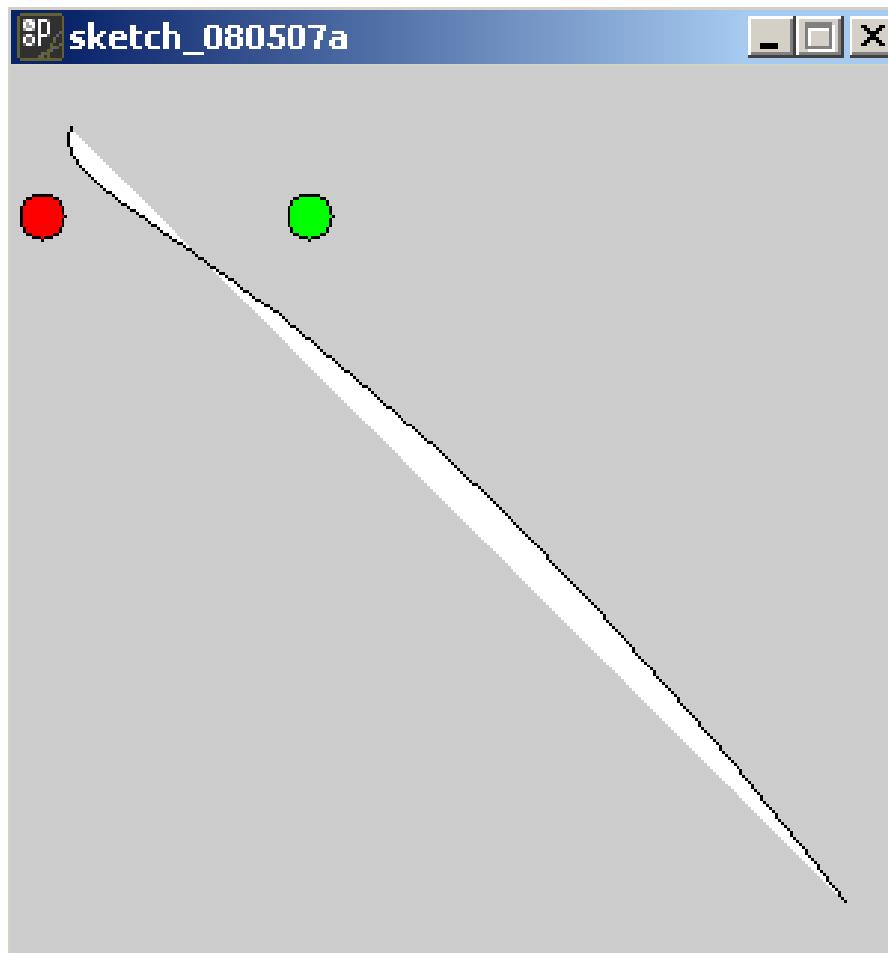
The Effect at the Ends is Weaker

```
bezier(20,20,210,250,290,250,280,280)
```



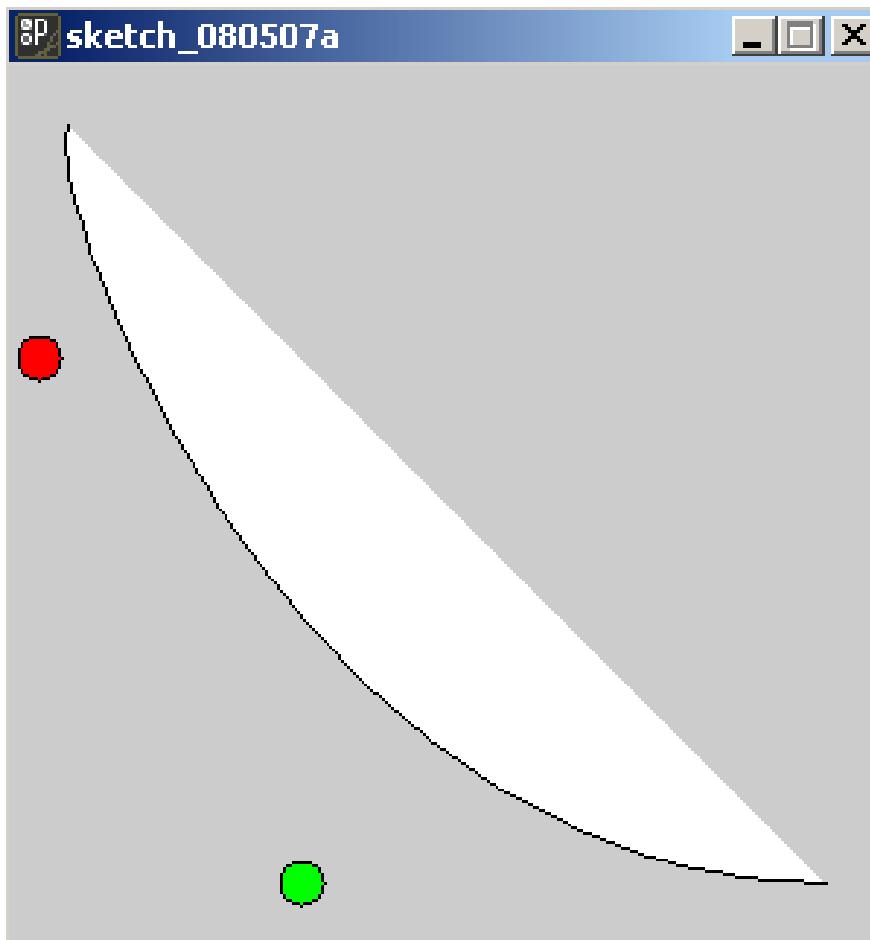
The Effect at the Ends is Weaker

```
bezier(20,20,10,50,100,50,280,280)
```



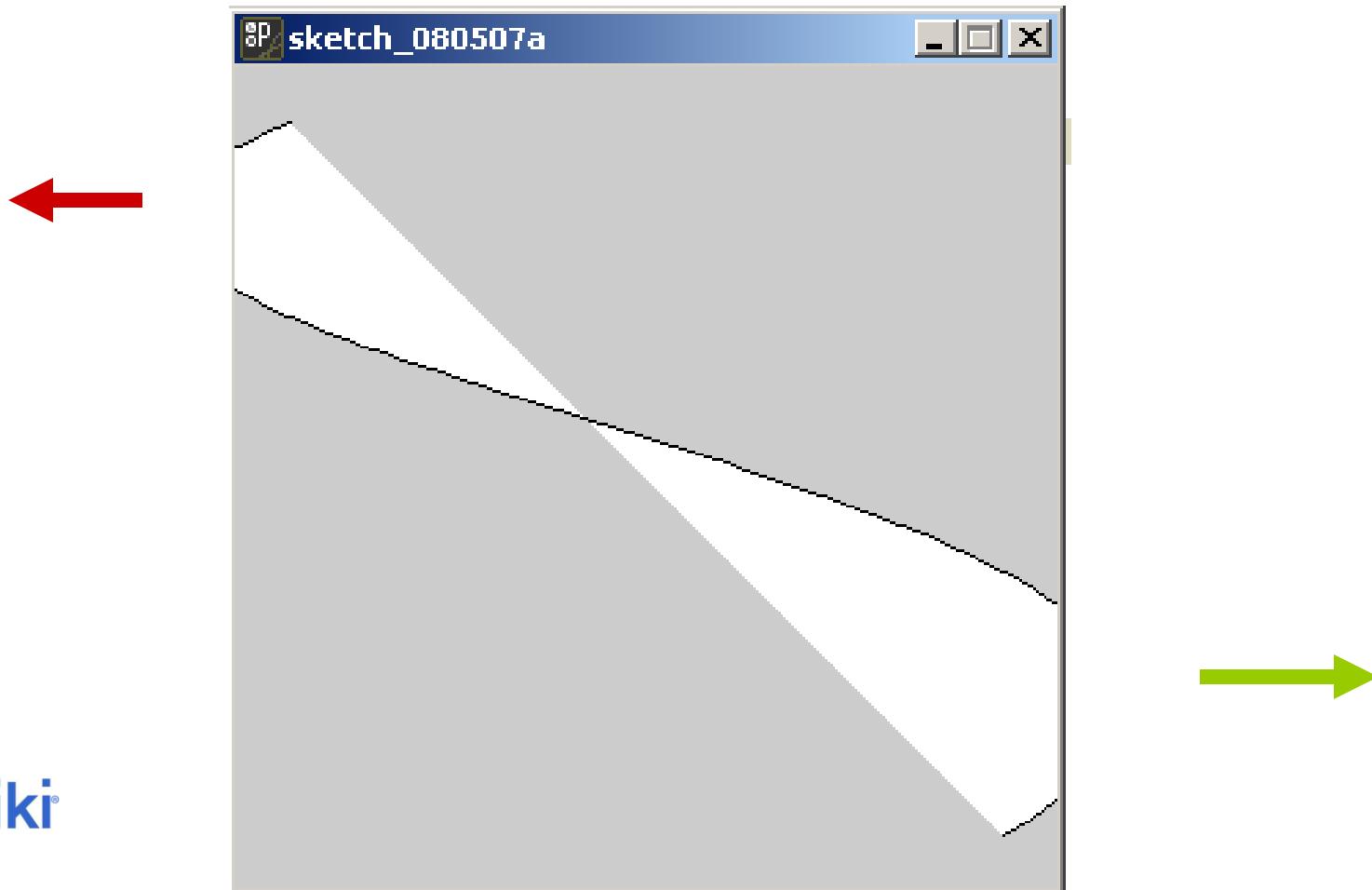
Both Points on the Same Side

```
bezier(20,20,10,50,100,280,280,280)
```



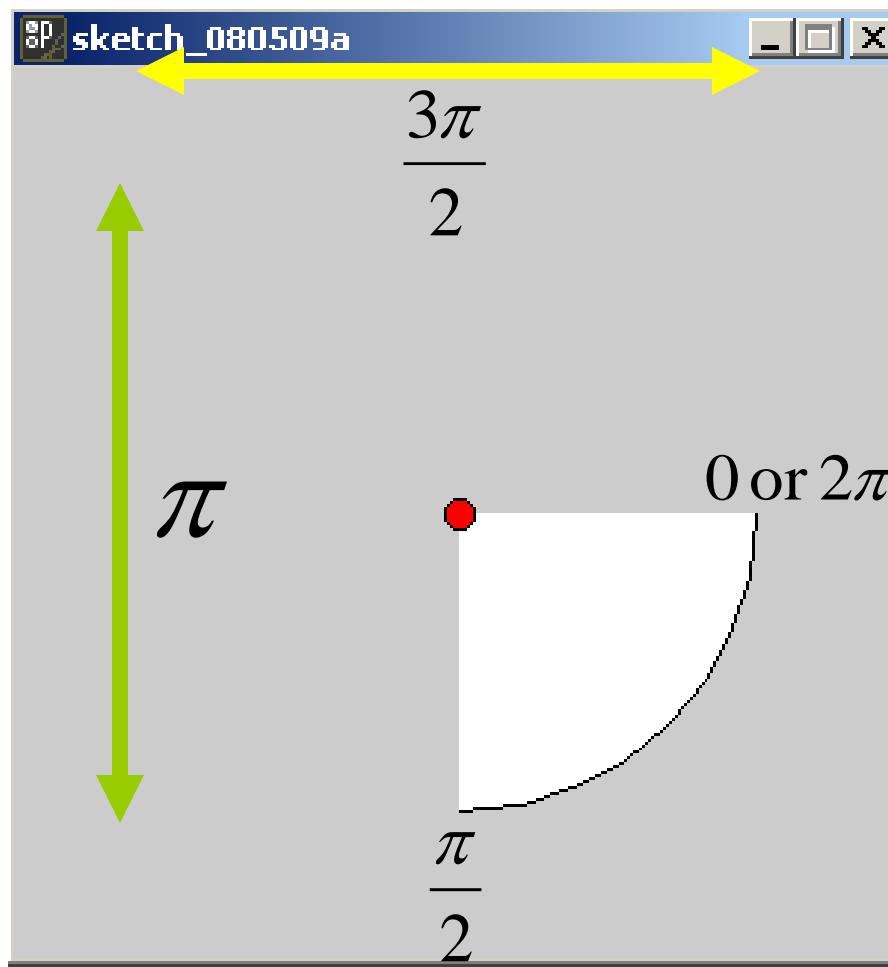
The Points Can be off the Screen!

```
bezier(20,20,-200,110,500,150,280,280)
```



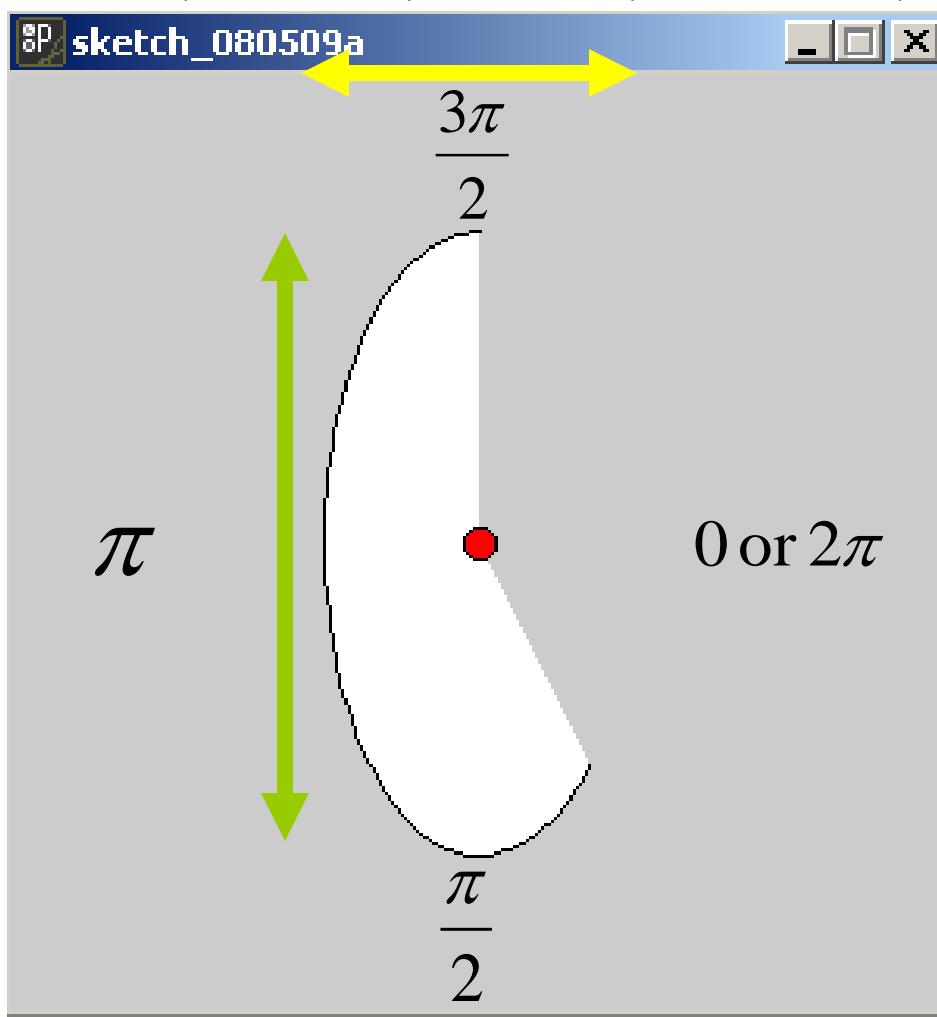
`arc()` draws part of an `ellipse()`

`arc(150, 150, 200, 200, 0, PI/2)`



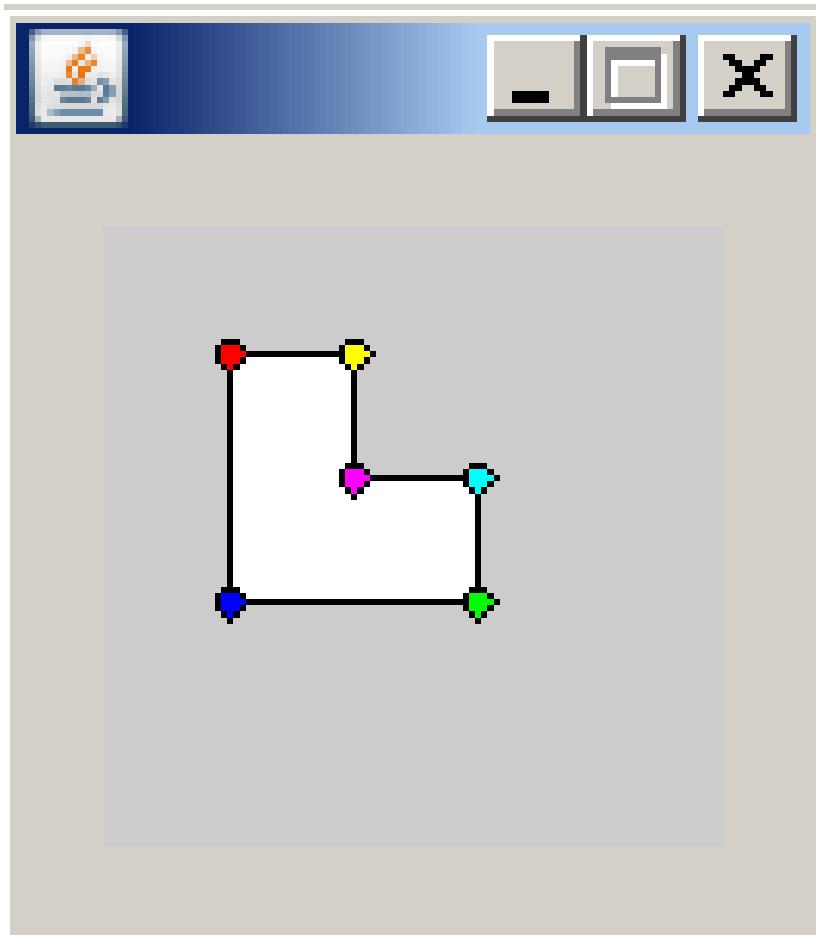
`arc()` draws part of an `ellipse()`

`arc(150, 150, 100, 200, PI/4, 3*PI/2)`



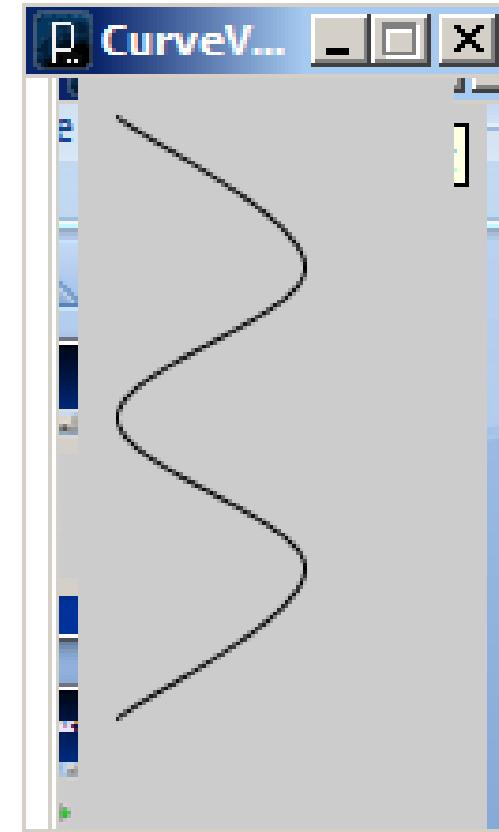
Polygons

```
beginShape()  
vertex(20, 20)  
vertex(40, 20)  
vertex(40, 40)  
vertex(60, 40)  
vertex(60, 60)  
vertex(20, 60)  
endShape(CLOSE)
```



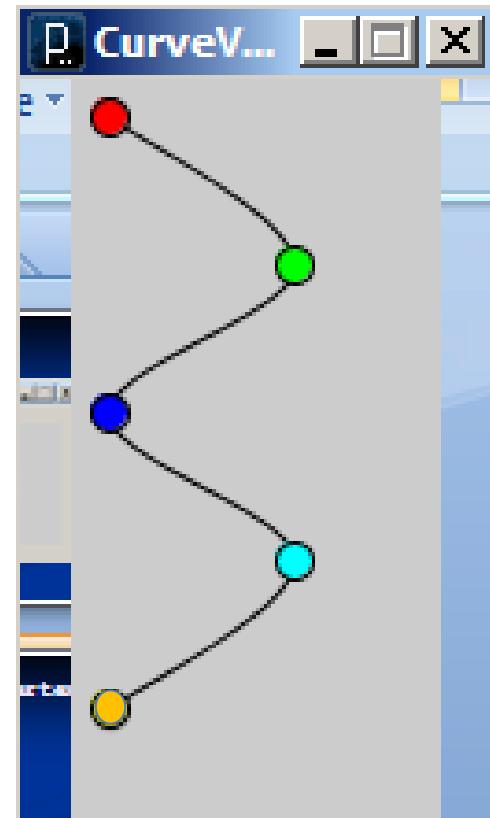
```
size(100,200)
noFill()
beginShape()
curveVertex(10,10)
curveVertex(10,10)
curveVertex(60,50)
curveVertex(10,90)
curveVertex(60,130)
curveVertex(10,170)
curveVertex(10,170)
endShape()
```

curveVertex

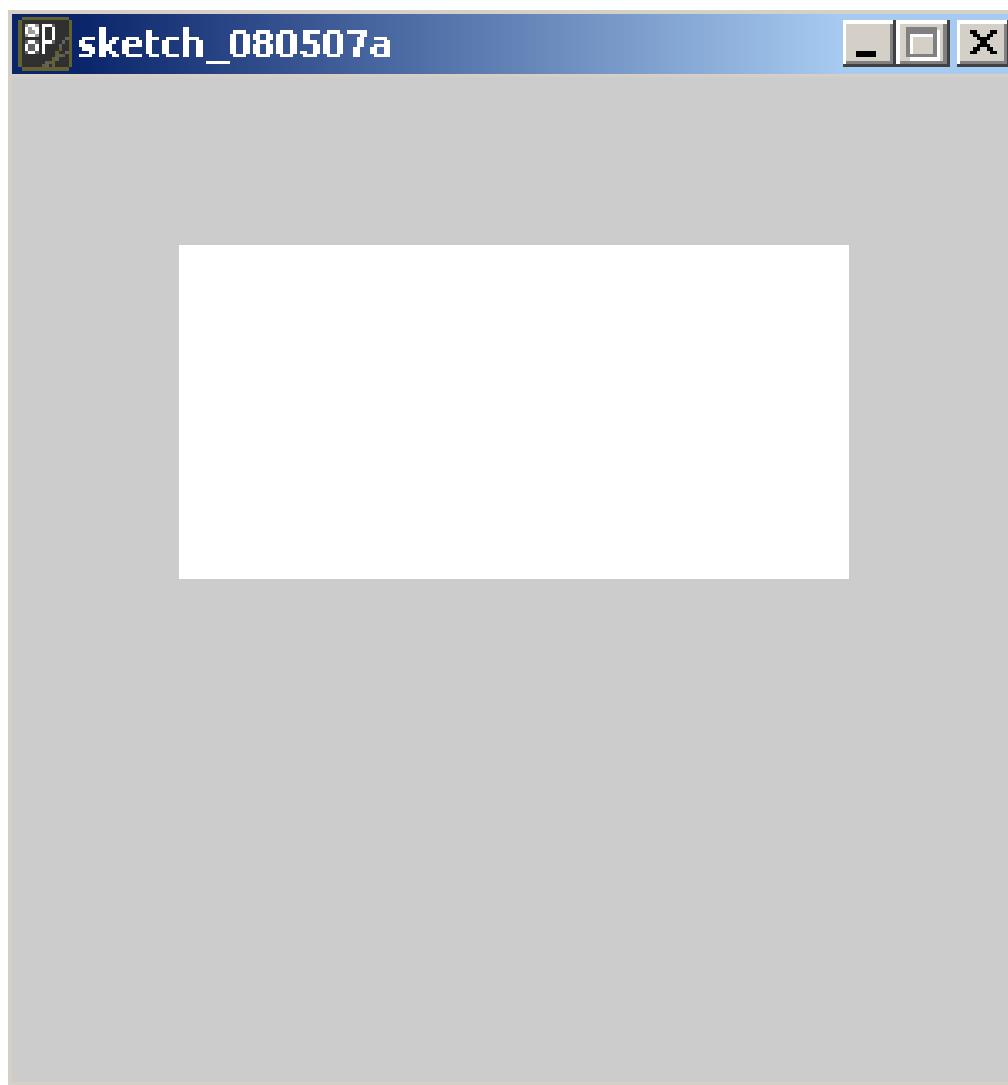


```
size(100,200)  
noFill()  
beginShape()  
curveVertex(10,10)  
curveVertex(10,10)  
curveVertex(60,50)  
curveVertex(10,90)  
curveVertex(60,130)  
curveVertex(10,170)  
curveVertex(10,170)  
endShape()
```

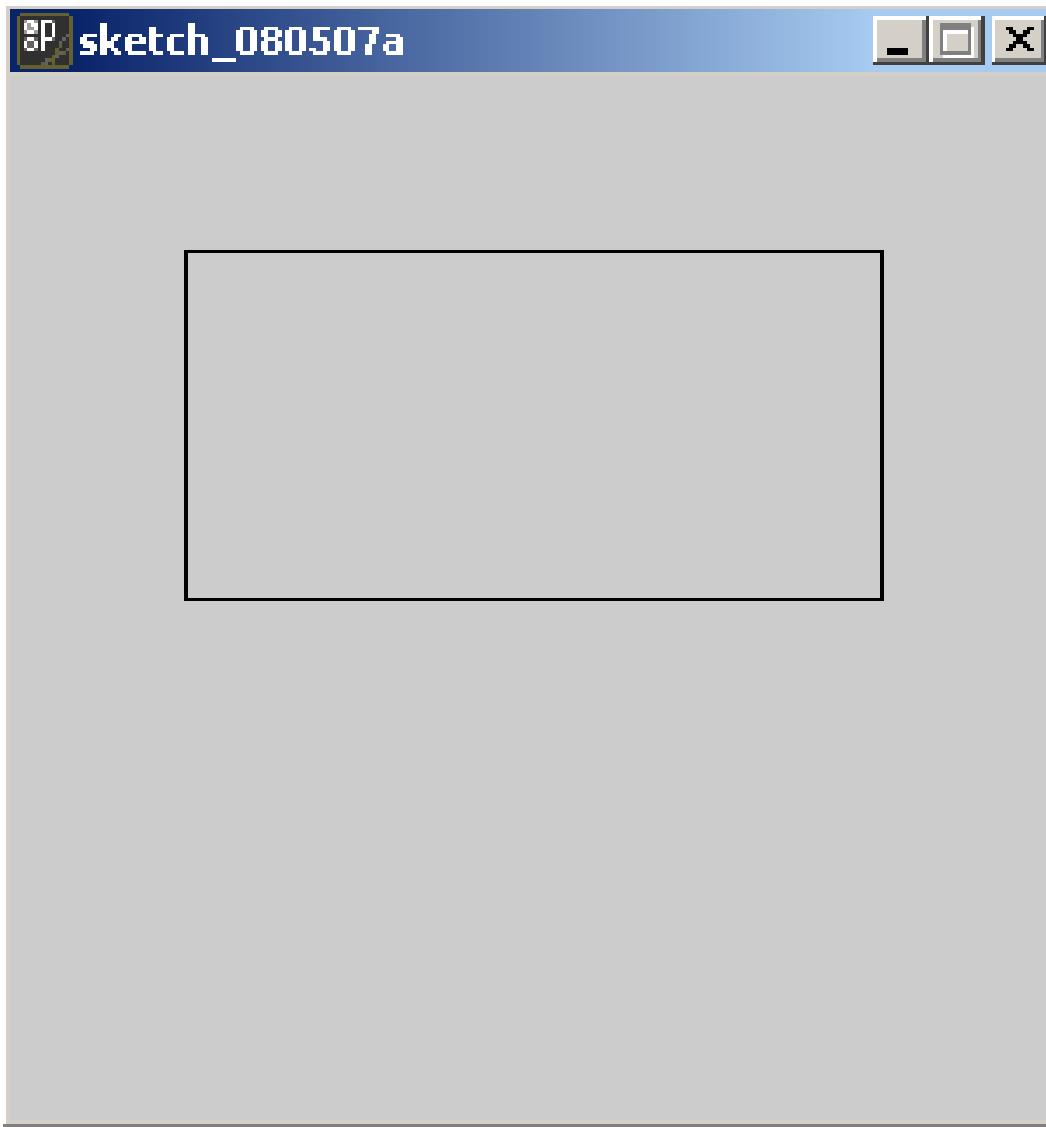
curveVertex



```
noStroke()  
rect(50,50,200,100)
```



```
noFill()  
rect(50,50,200,100)
```



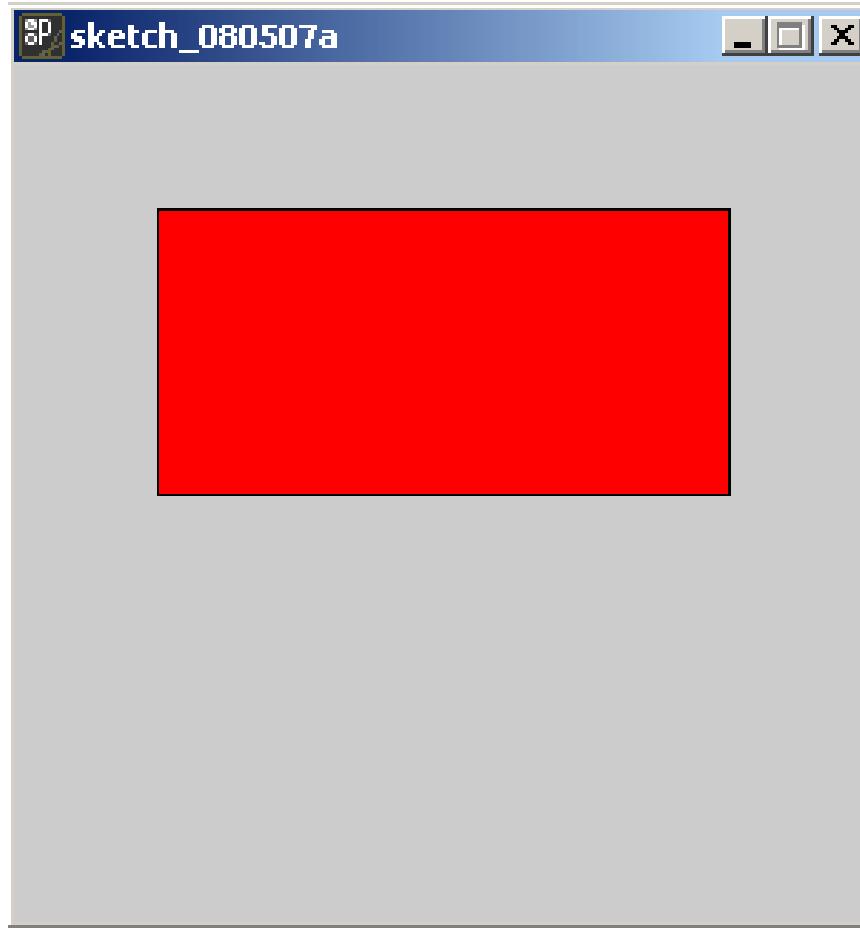
```
noFill()  
noStroke()  
rect(50,50,200,100)
```



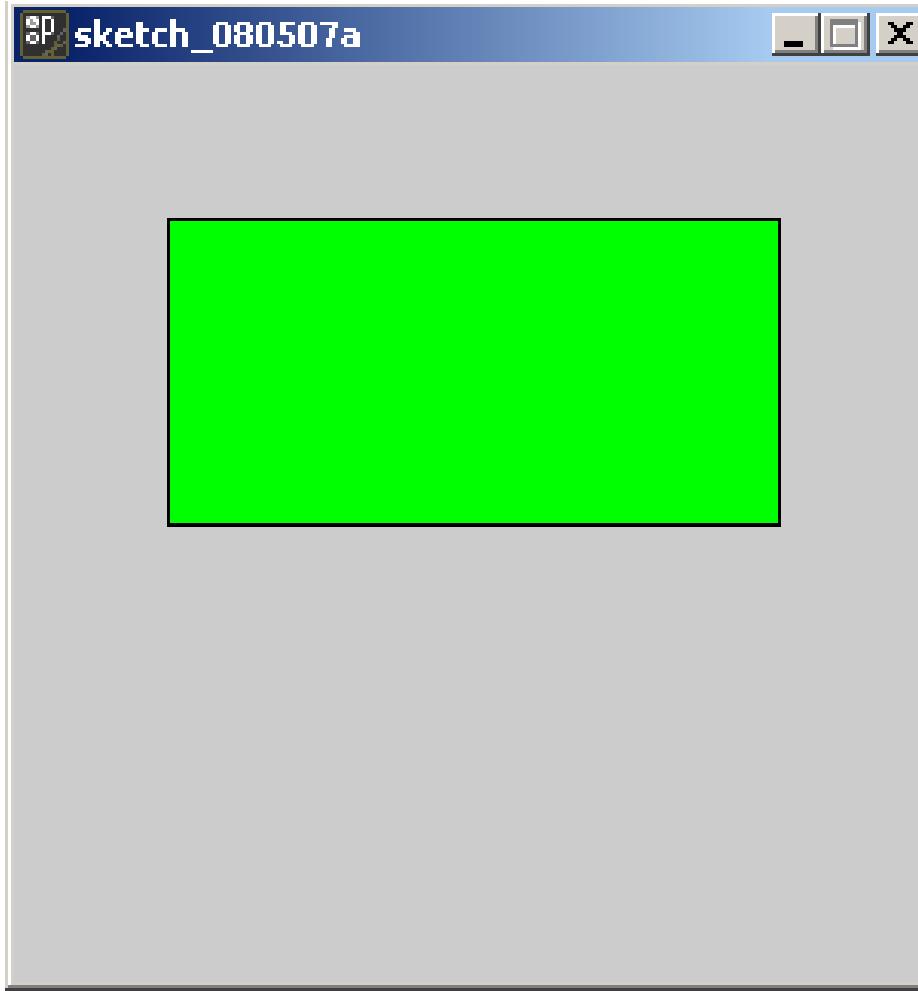
Empty!

```
fill(255,0,0)
```

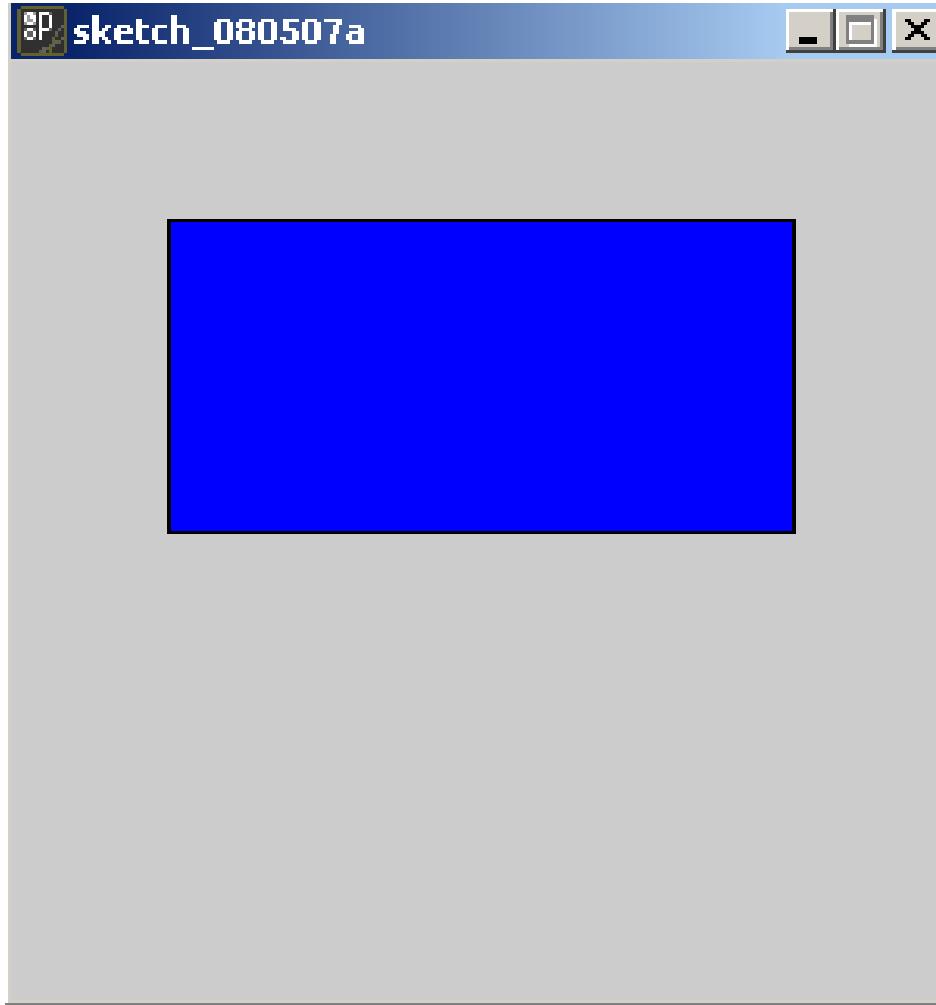
```
rect(50,50,200,100)
```



```
fill(0,255,0)  
rect(50,50,200,100)
```



```
fill(0,0,255)  
rect(50,50,200,100)
```

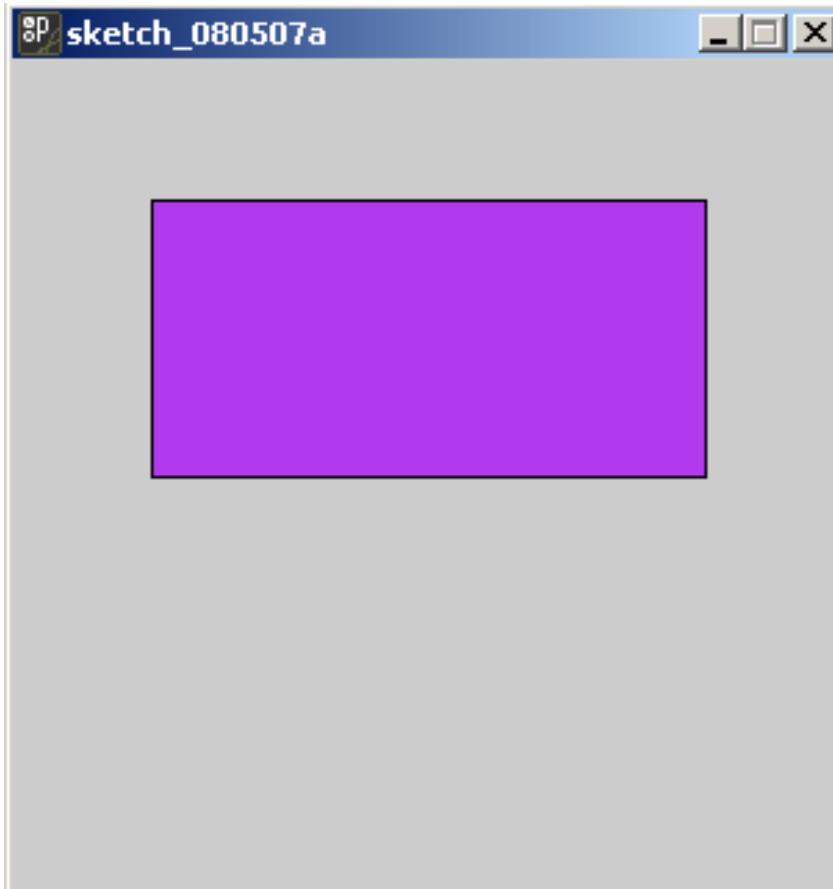


```
fill(178,58,238)
```

```
rect(50,50,200,100)
```

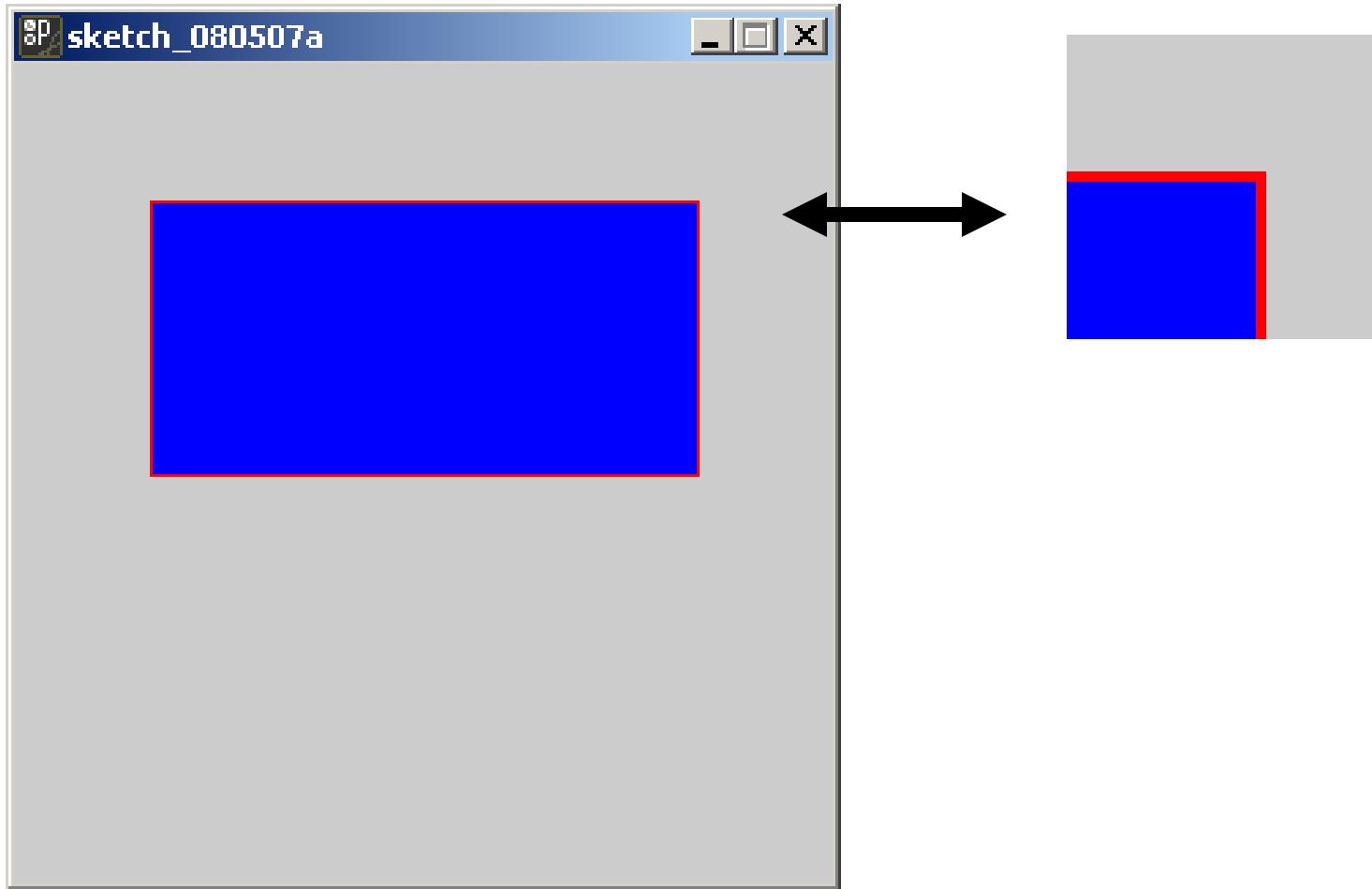
RGB Color Codes:

<http://www.tayloredmktg.com/rgb/>

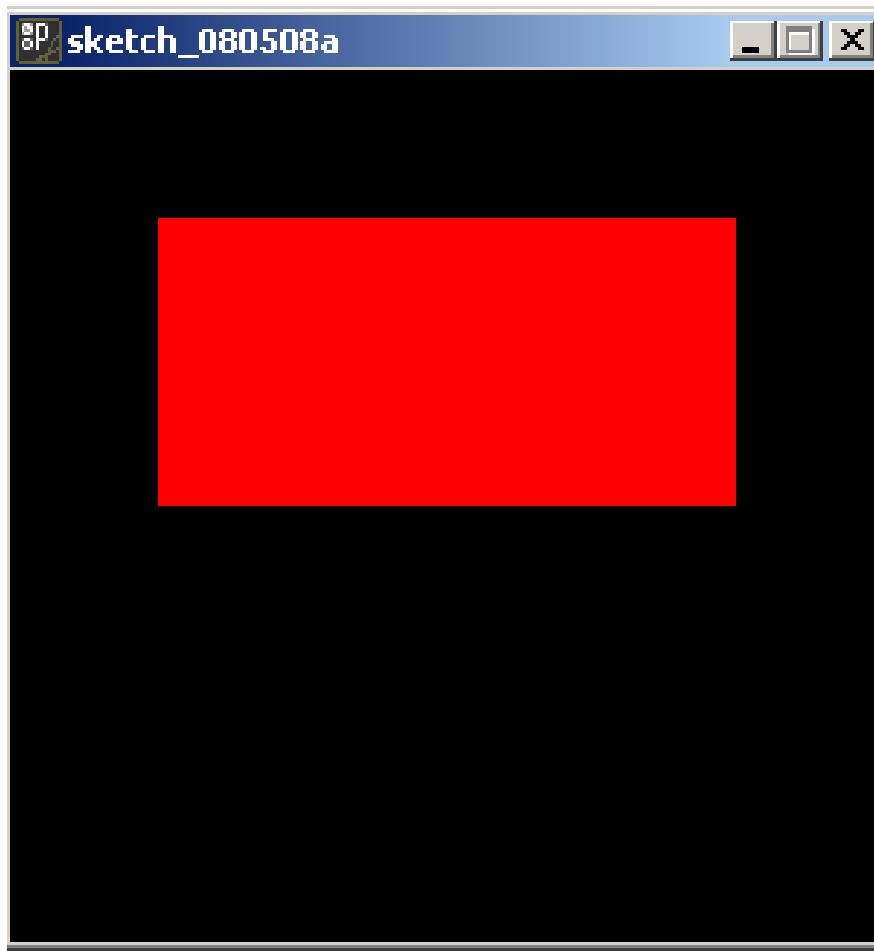


Gainsboro	220-220-220	cccccc	
Floral White	255-250-240	ffffaf	
Old Lace	253-245-230	fdf5e6	
Linen	240-240-230	faf0e6	
Antique White	250-235-215	faebd7	
Antique White 2	238-223-204	eedfcc	
Antique White 3	205-192-176	cdcb0b	

```
fill (0 ,0 ,255)  
stroke (255 ,0 ,0)  
rect (50 ,50 ,200 ,100)
```



```
background(0,0,0)
fill(255,0,0)
rect(50,50,200,100)
```



Grayscale

- There are one argument versions of **fill()**, **background()** and **stroke()** that give grayscale colors.

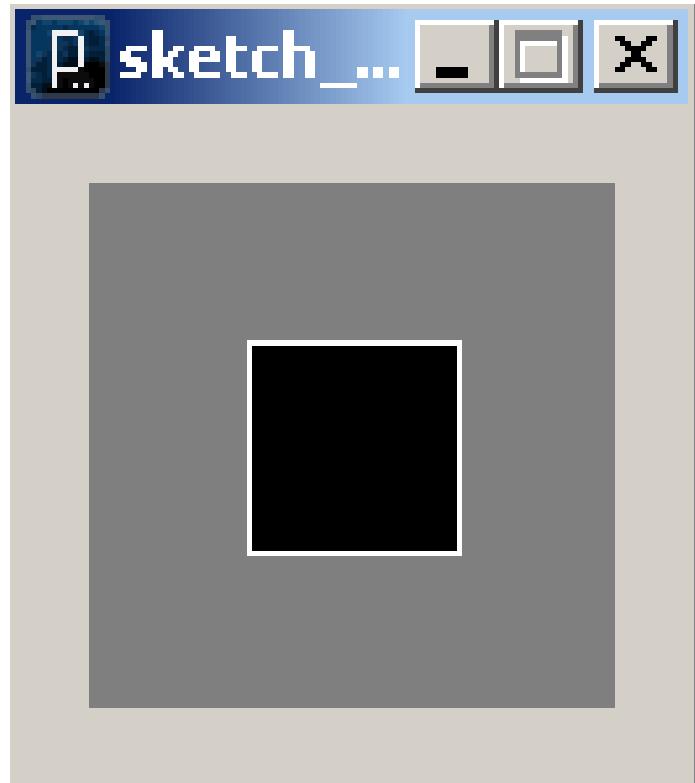
```
background(127)
```

```
fill(0)
```

```
stroke(255)
```

```
rect(30,30,40,40)
```

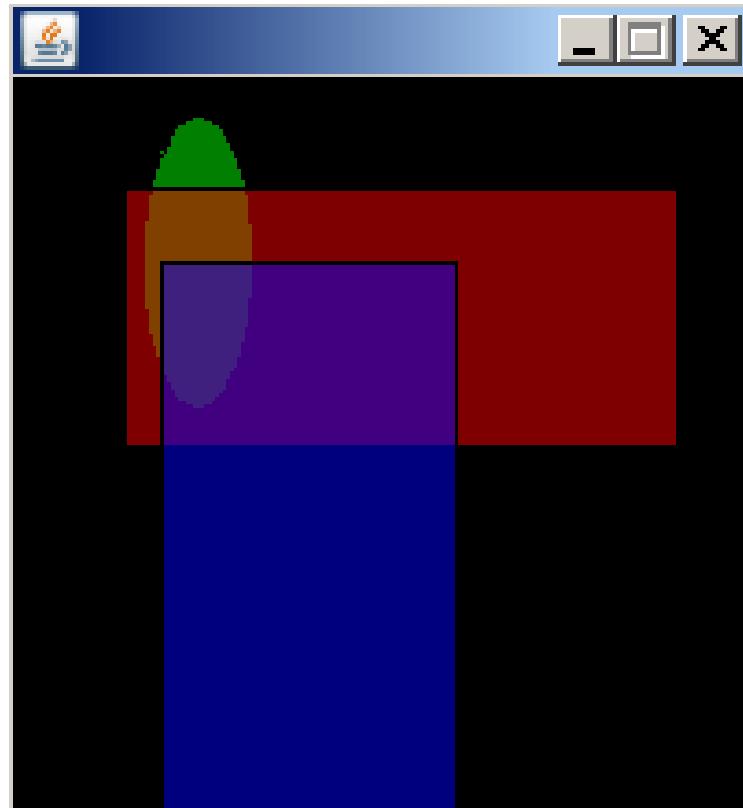
- The default background is 204.



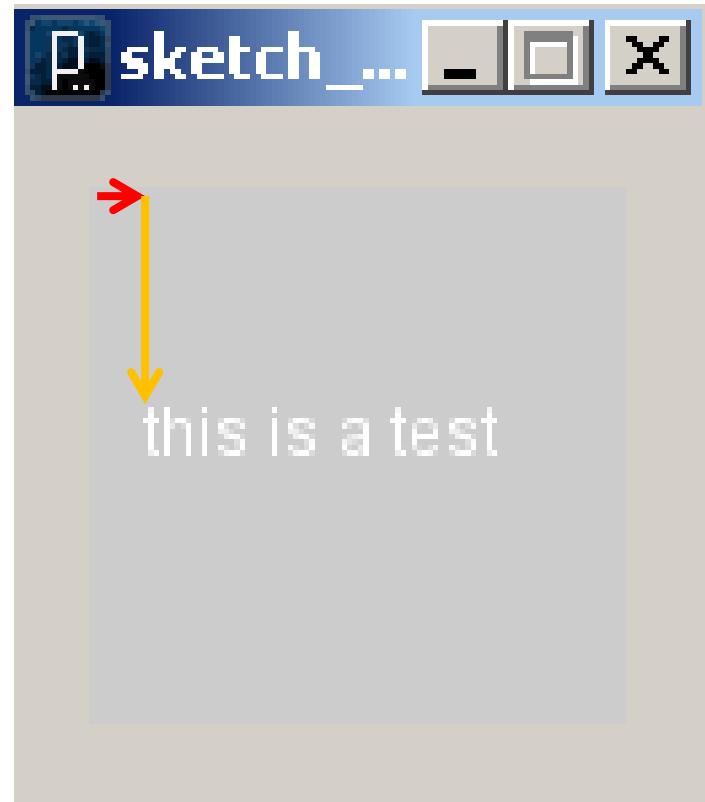
- You can make shapes transparent by using the four argument version of `fill()` .
- The **fourth argument** is opacity.

```
size(200,200)
background(0)
fill(0,255,0,127)
ellipse(50,50,30,80)
fill(255,0,0,127)
rect(30,30,150,70)
fill(0,0,255,127)
rect(40,50,80,170)
```

Opacity



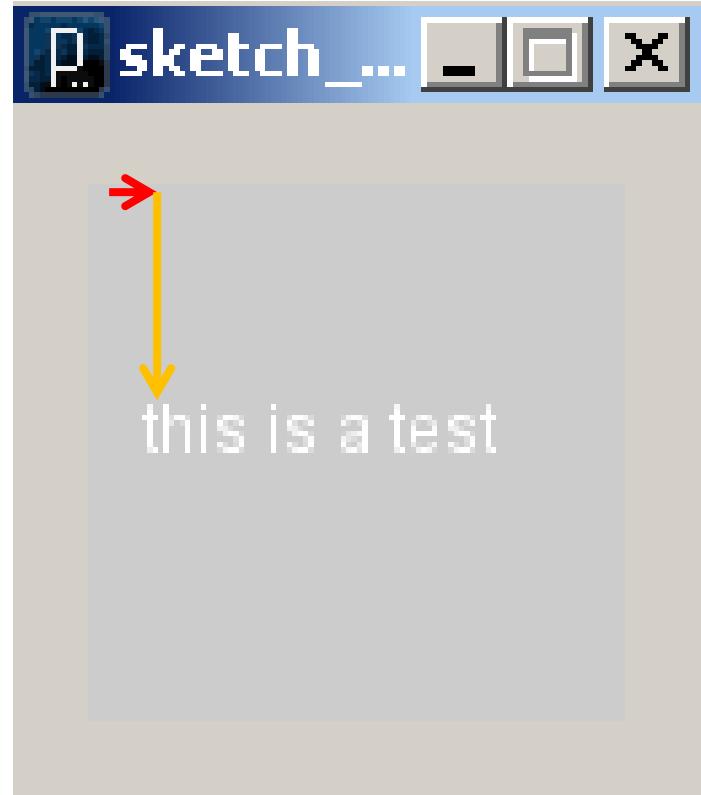
```
text("this is a test",10,50)
```



stroke() has no effect on text()

stroke(0)

text("this is a test",10,50)



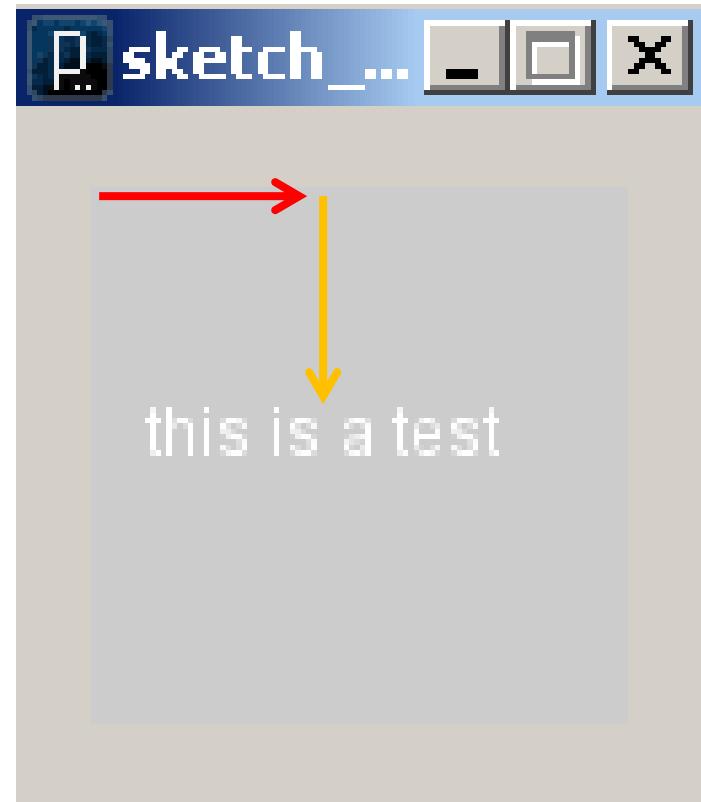
fill() changes text() color

fill(0)

text("this is a test",10,50)



```
textAlign(CENTER)  
text("this is a test",50,50)
```

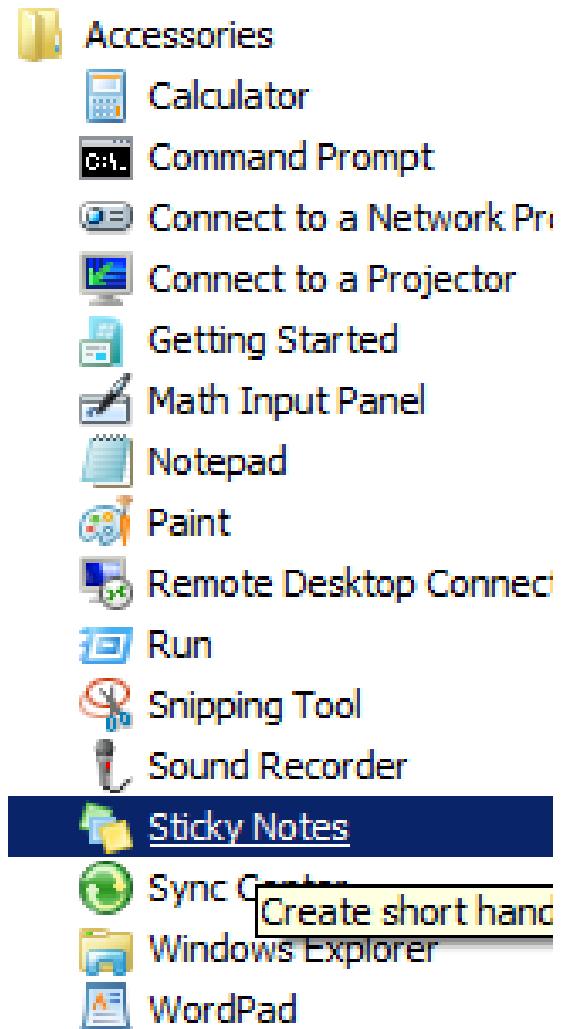


```
textSize(24)  
text("this is a test",10,50)
```



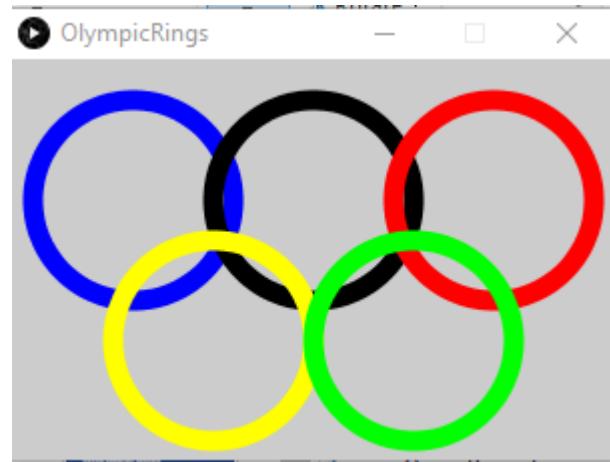
Practice Quiz Question

- A preview of the actual Quiz.
- Write your answer in something like notepad, WordPad or a sticky note.
- Make your font nice and big so your instructor can read it from a distance.
- In WordPad and sticky notes *Ctrl + Scroll Wheel* or *Ctrl + Shift + >* makes the font bigger.



Practice Quiz Question

1. Which **algorithm** best describes this Olympic Rings program?



A

1. Draw Yellow Ring
2. Draw Green Ring
3. Draw Blue Ring
4. Draw Black Ring
5. Draw Black Ring

B

1. Draw Red Ring
2. Draw Black Ring
3. Draw Blue Ring
4. Draw Green Ring
5. Draw Yellow Ring

C

1. Draw Blue Ring
2. Draw Black Ring
3. Draw Red Ring
4. Draw Yellow Ring
5. Draw Green Ring

Practice Quiz Questions

2. Fill in the blank: An _____ is like a dictionary of the words in a programming language
3. Which step is unnecessary in the following algorithm? (An algorithm can be thought of as a step by step process)
 - i. Set stroke to red
 - ii. Draw an ellipse at (30,30)
 - iii. Set stroke to red
 - iv. Draw an ellipse at (60,60)
 - v. Set stroke to green
 - vi. Draw an ellipse at (45,45)

Algorithms

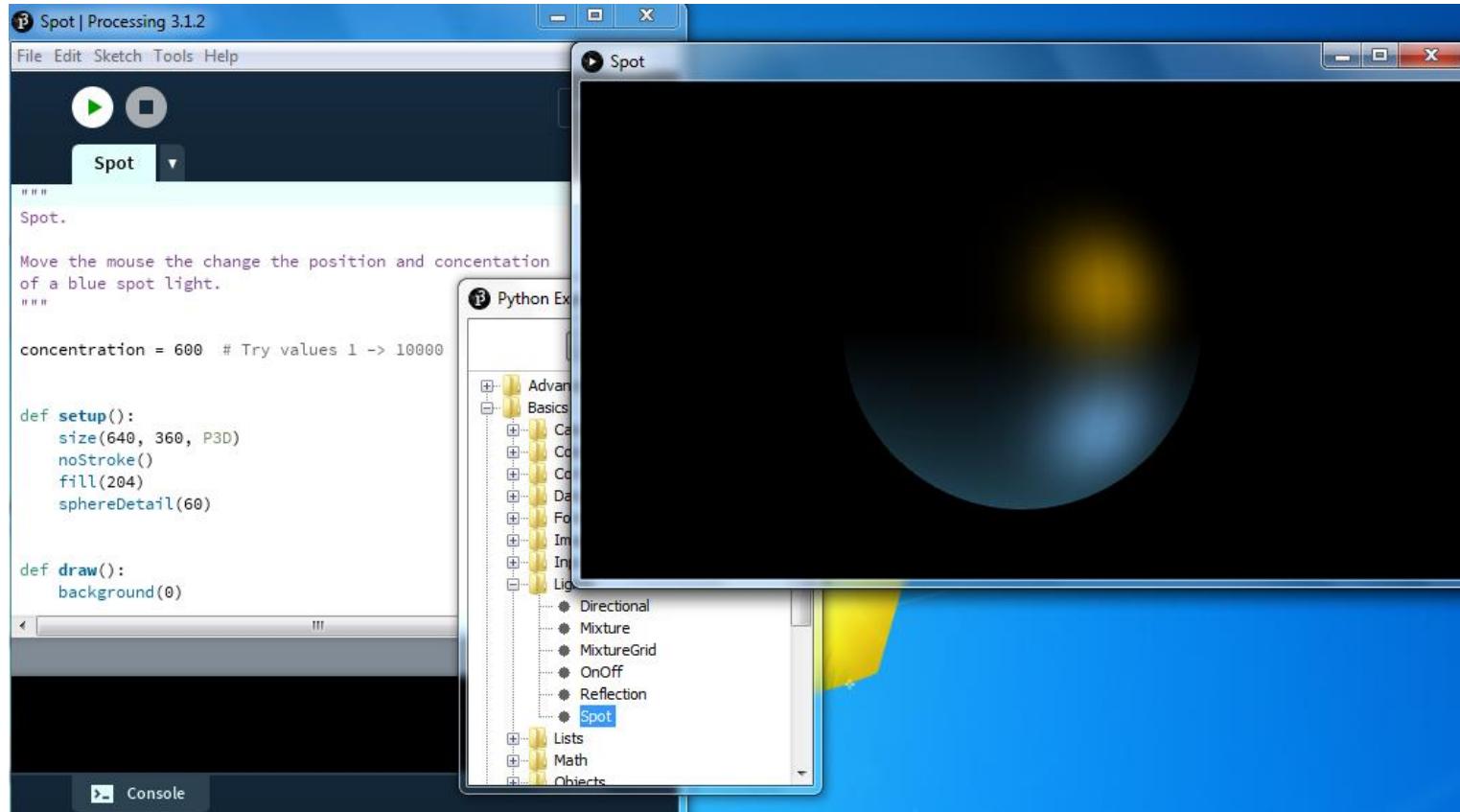
- Algorithms can be thought of as a *step by step process*.
- Before you can write the code for a computer program you need to figure out the algorithm.
- Then you can translate that algorithm in to the particular code for whatever programming language your are using.
- The AP exam won't test you on Python, but it will test you on Algorithms which are one of the most important principles in Computer Science.

CamelCase

- **CamelCase** is a style of writing without spaces.
- It's "good style" to use CamelCase for names in a programming language.
- Each separate word in the name is capitalized within the compound—as in ***beginShape()***, ***endShape()***, ***MacGyver***, or ***iPod***.
- It's named after the "humps" of the capitals.

Examples

Choose *File | Examples* to see many working Python programs.



Comments

Python ignores anything

after a pound sign

- Tells the computer "ignore this", this is for people to read.

Comments Can be Used to Identify the Programmer and/or Describe the Program.



```
# Mr Simon's happy face program
size(200,100)

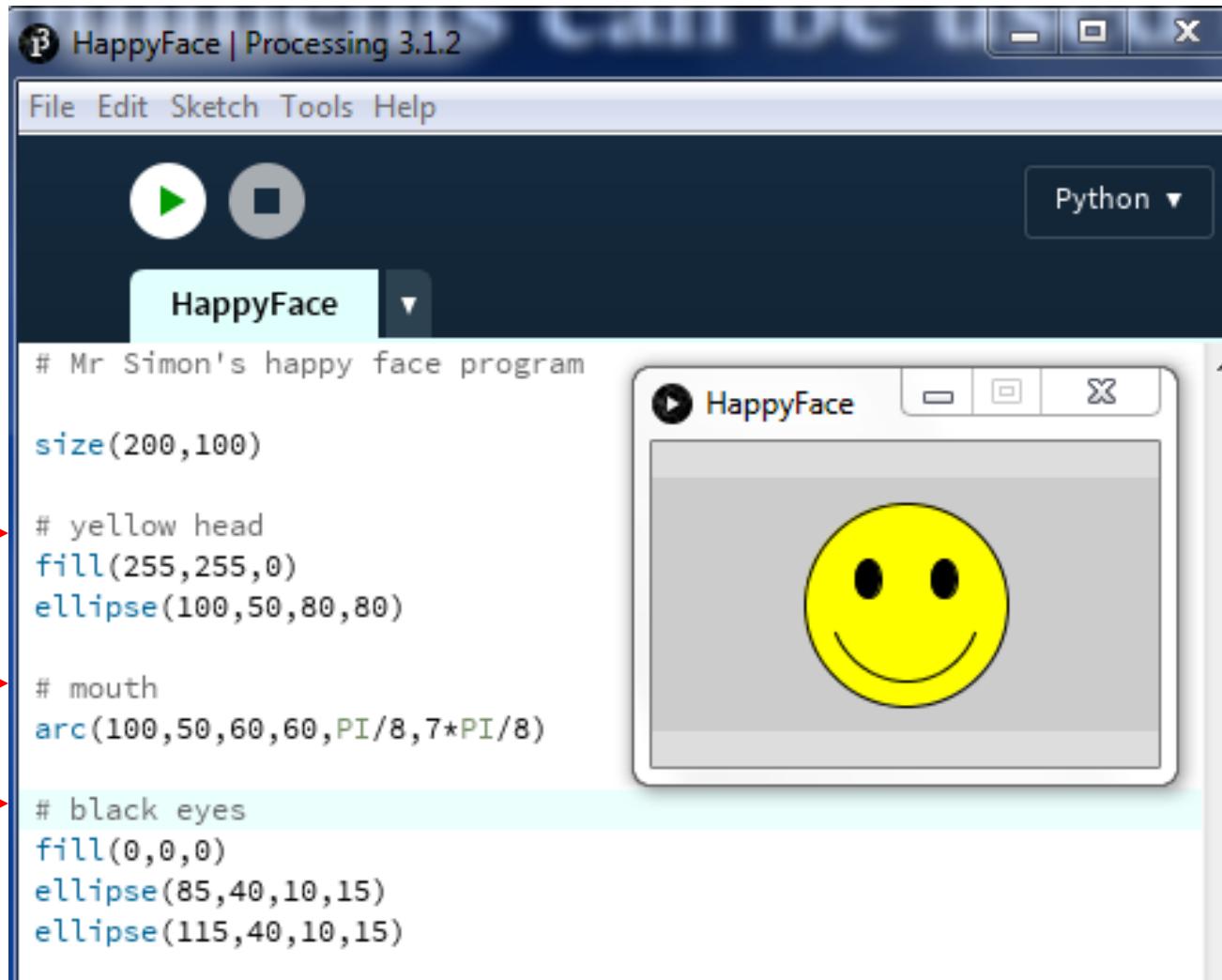
# yellow head
fill(255,255,0)
ellipse(100,50,80,80)

# mouth
arc(100,50,60,60,PI/8,7*PI/8)

# black eyes
fill(0,0,0)
ellipse(85,40,10,15)
ellipse(115,40,10,15)
```



Comments Can be Used to Label the Different Parts of the Program.



The image shows the Processing 3.1.2 software interface. The title bar says "HappyFace | Processing 3.1.2". The menu bar includes File, Edit, Sketch, Tools, and Help. A toolbar with a play button and a square button is visible. The sketch name "HappyFace" is selected in a dropdown. A dropdown menu shows "Python". The code editor contains the following code:

```
# Mr Simon's happy face program

size(200,100)

# yellow head
fill(255,255,0)
ellipse(100,50,80,80)

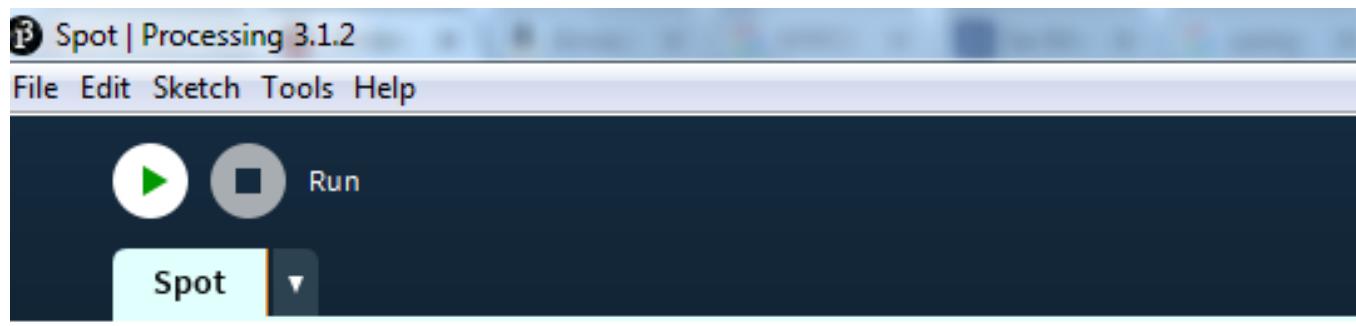
# mouth
arc(100,50,60,60,PI/8,7*PI/8)

# black eyes
fill(0,0,0)
ellipse(85,40,10,15)
ellipse(115,40,10,15)
```

Three red arrows point from the text labels "# yellow head", "# mouth", and "# black eyes" in the code to their corresponding graphical elements in the preview window. The preview window shows a yellow circle with two black dots and a curved smile line.

Comments

For AP Computer Science Principles you are welcome to copy and modify code that other people have written as long as you **use comments to label the adapted code.**



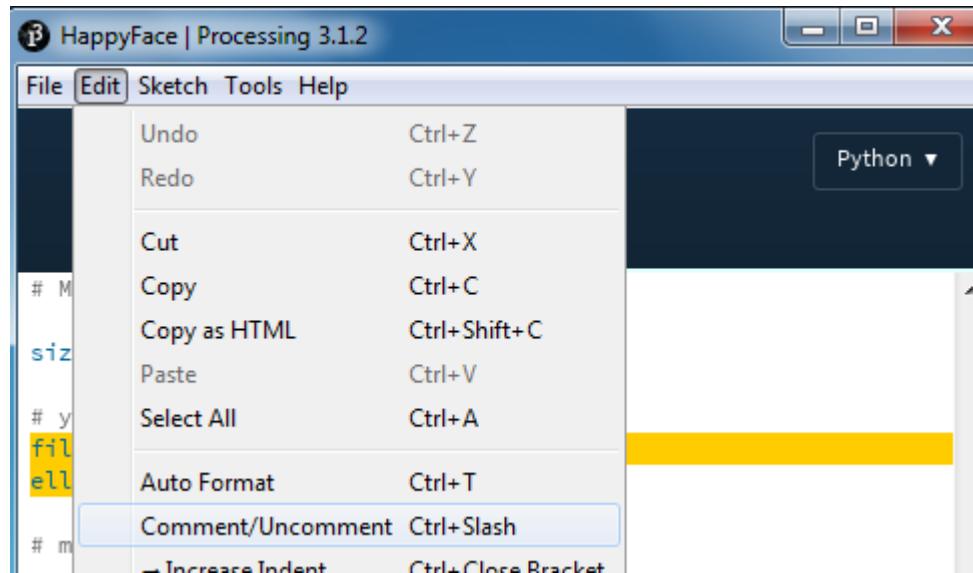
```
# The following code was adapted from the Processing Python Spot example
```

```
concentration = 600 # Try values 1 -> 10000
```

```
def setup():
    size(640, 360, P3D)
    noStroke()
    fill(204)
```

Commenting Out Code

- It's easy to forget what some code is doing.
- It's best to focus on one thing at a time.
- You can temporarily comment out code by selecting *Edit / Comment/Uncomment* or *Ctrl+{/*



Arithmetic

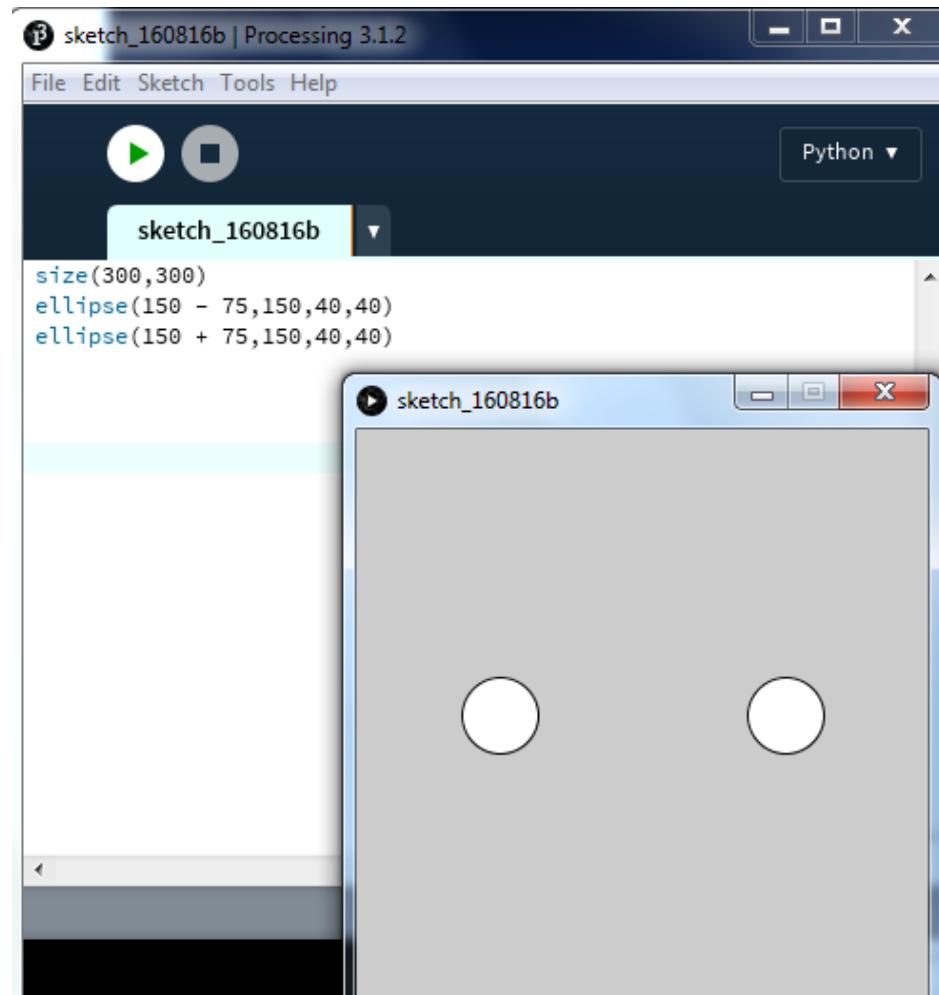
- Calculations use the following arithmetic operators: + - * / %.
- The operators can be combined with numbers to make *expressions*.
- The expression $5 + 3$ evaluates to 8.
- Parenthesis can be used to create more complicated expressions
- $(5 * (2 - 3))$ evaluates to -5.

Expressions Can be Arguments

- What will be the height and width of the following ellipse?
- `ellipse(150,150,2*(22+78),40*(8-2))`

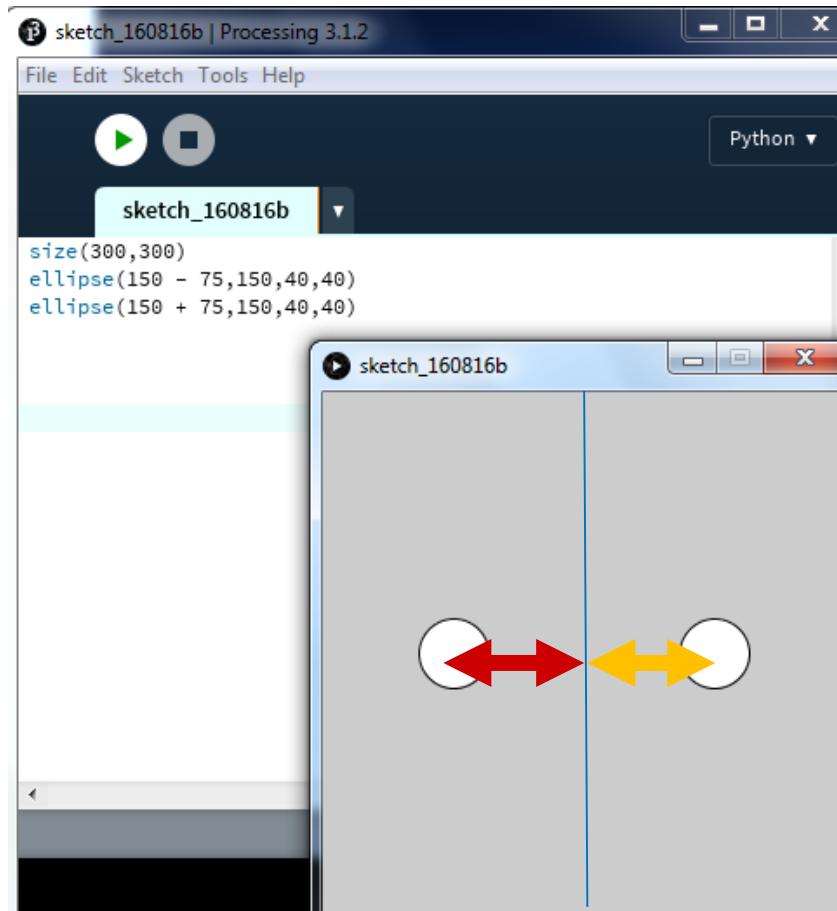
Symmetry

Expressions can be used to make placing symmetrical arrangements easier.



Symmetry

```
size(300,300)  
ellipse(150-75,150,40,40)  
ellipse(150+75,150,40,40)
```



Integer Arithmetic

- Division is done differently with integers than with decimals:
 - $5.0/4.0$ evaluates to 1.25
 - $5/4.0$ evaluates to 1.25
 - $5.0/4$ evaluates to 1.25
 - $5/4$ evaluates to 1
- If one or more numbers in an expression is a decimal, the result will be a decimal.
- If all the numbers in an expression are integers, the result will be an integer.

Integer Arithmetic

If you see an integer division expression like $5/4$, ask yourself "How many times does 4 go into 5?"

Modulus and Integer Division

Remember how you did math in grade school?

$$5 \overline{)8}$$

Modulus and Integer Division

Remember how you did math in grade school?

$$5 \overline{)8}$$

Modulus and Integer Division

Remember how you did math in grade school?

$$\begin{array}{r} 1 \\ 5 \overline{)8} \\ \underline{5} \end{array}$$

Modulus and Integer Division

Remember how you did math in grade school?

$$\begin{array}{r} 1 \\ 5 \overline{)8} \\ 5 \\ \hline 3 \end{array}$$

Modulus and Integer Division

The modulus operator gives the remainder of an integer division expression.

The diagram illustrates the relationship between integer division and modulus operations. It features two blue circles on the left. The top circle contains the expression $8/5$, and the bottom circle contains the expression $8\%5$. Two red arrows point from these circles to the right, each leading to a long division diagram. The top arrow points to a diagram where 5 goes into 8 once, leaving a remainder of 3, with the quotient 1 written above the division bar. The bottom arrow points to a similar diagram where 5 goes into 8 once, leaving a remainder of 3, with the quotient 1 written above the division bar. This visualizes how the modulus operator $\%$ captures the remainder of the division process.

$$5 \overline{)8} \quad \begin{array}{r} 1 \\ \hline 3 \end{array}$$

Practice Quiz Question:

Evaluate the following expressions

1. $9/4$

2. $9 \text{ MOD } 4$ ($9\%4$ in Python)

3. $21/2$

4. $21 \text{ MOD } 2$ ($21\%2$)

5. $5/8$

6. $5 \text{ MOD } 8$ ($5\%8$)

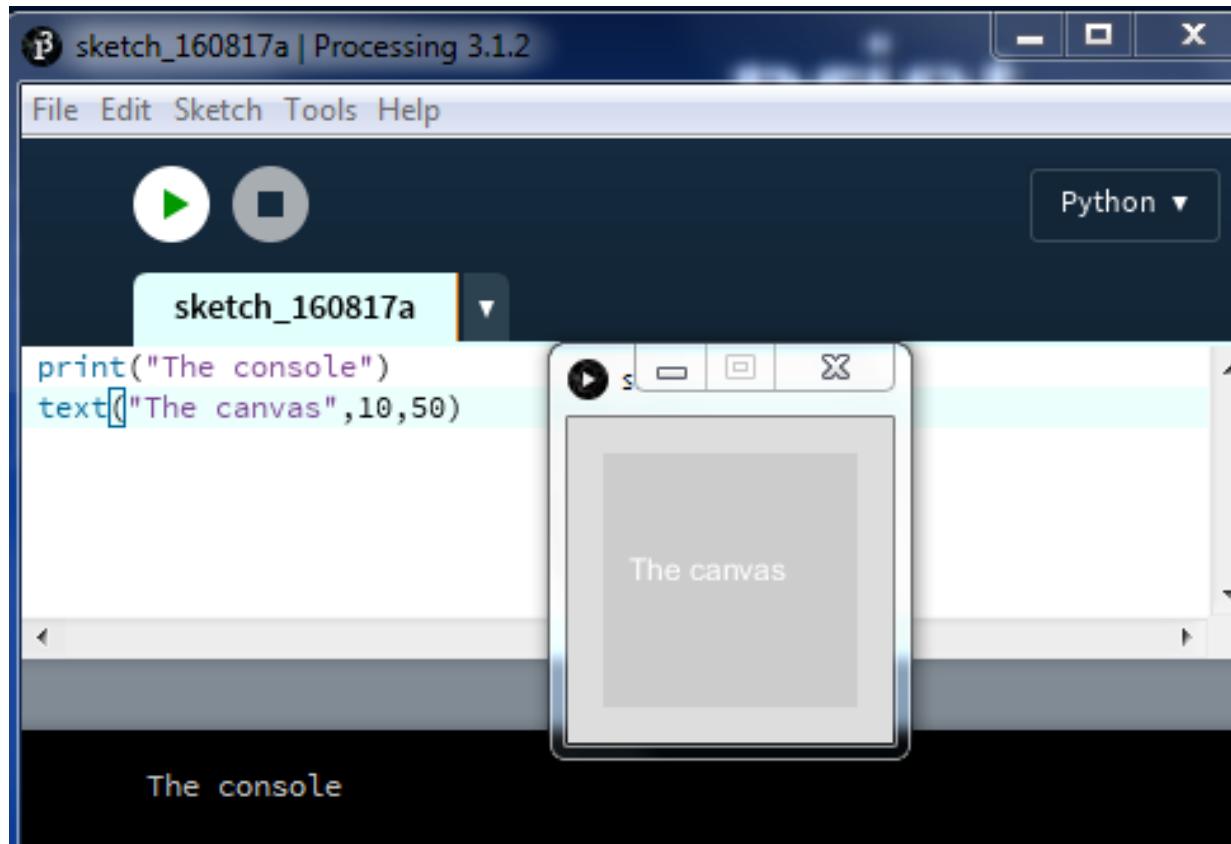
Finished your assignment?

Extra Time?

- You are expected to use your lab time productively
- Try these:
 - Take the practice multiple choice exam on pages 84-100 of the AP CSP course framework book
 - Learn more Python at codecademy
 - Practice your Python at codingbat
 - Read a Python book like *Hacking Secret Ciphers with Python*

Print

- **print()** displays to the *console*
- **text()** displays in the *drawing canvas*



Double Quotes Make a Difference

- If you **print** something with double quotes, Python prints it exactly as written including spaces.
- If you **print** an expression without double quotes Python evaluates it first then prints the result.

The screenshot shows the Processing IDE interface. The title bar reads "sketch_160817a | Processing". The menu bar includes "File", "Edit", "Sketch", "Tools", and "Help". Below the menu is a toolbar with a play button and a stop button. The sketch window contains the code:

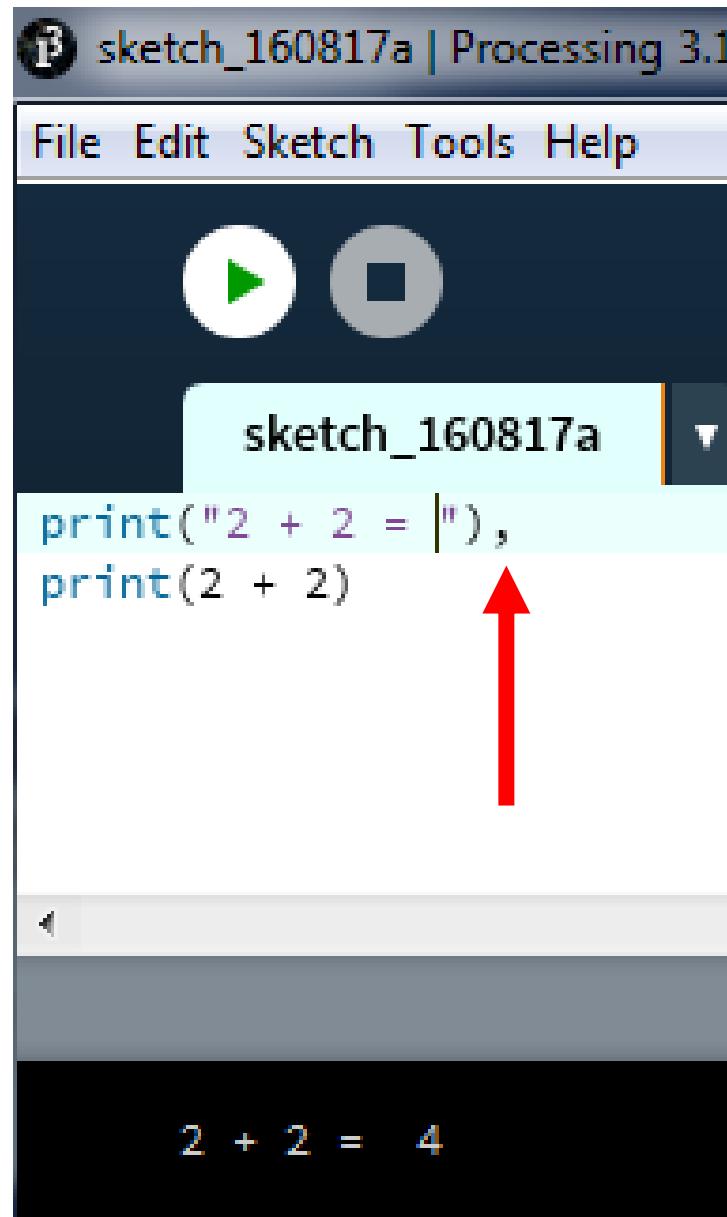
```
print("2 + 2")
print(2 + 2)
```

The output window shows the results of the code execution:

```
2 + 2
4
```

Print

- Ending a **print** statement in a **comma** allows output to go on the same line.



The screenshot shows the Processing 3.1 software interface. At the top is a menu bar with File, Edit, Sketch, Tools, and Help. Below the menu is a toolbar with a play button and a stop button. The main area is titled "sketch_160817a". In the code editor, there are two lines of code: `print("2 + 2 = "),` and `print(2 + 2)`. A red arrow points from the word "print" in the second line to the terminal window below. The terminal window displays the output: `2 + 2 = 4`.

```
print("2 + 2 = "),  
print(2 + 2)
```

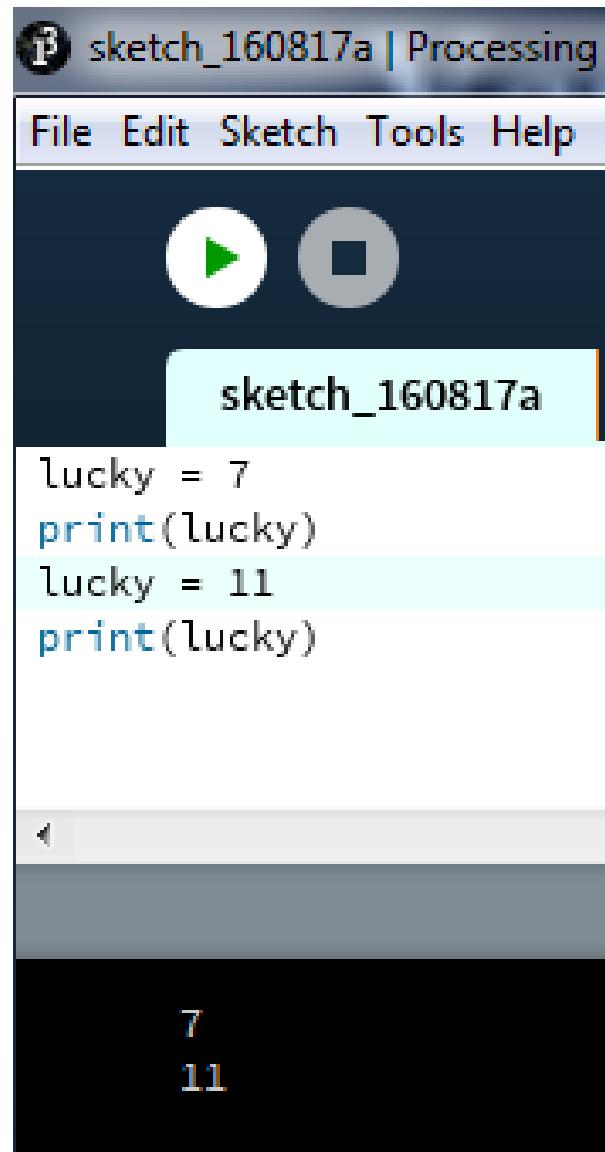
```
2 + 2 = 4
```

Variables

Vari-ables allow values to **change**.

Variables in Python

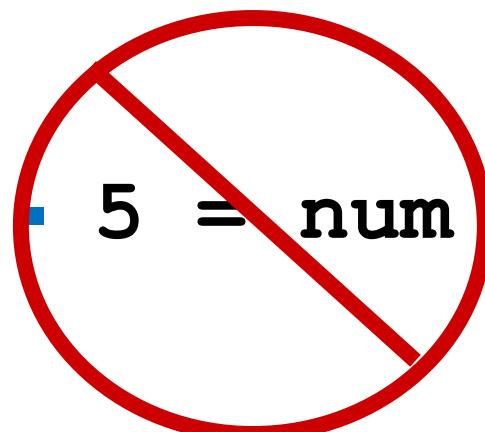
- You can imagine a variable as a box that you can put stuff into.
- A variable can only hold one value at a time.
- When **lucky** is assigned **11**, it replaces the **7**.



```
sketch_160817a | Processing  
File Edit Sketch Tools Help  
sketch_160817a  
lucky = 7  
print(lucky)  
lucky = 11  
print(lucky)  
  
7  
11
```

The Assignment Operator

- **=** is called the *assignment operator*
 - It takes the value on the *right*, and puts it in the variable on the *left*
 - **num = 5**
 - You can't do it the other way around!
- 5 = num**



The Assignment Operator

- You can't do an assignment as an argument
- ~~ellipse (x = 45, 55, 10, 10)~~
- You have to do in two separate lines
- `x = 45`
- `ellipse(x, 55, 10, 10)`

The Assignment Operator

- This looks weird if you are use to algebra, but it's very common in programming.
- `x = 3`
- `x = x + 1`
- What value is in the `x` variable after those two lines of code are executed?

Assignments in Python

- Remember that assignments go from right to left
- What would the output of the following program be?

```
a = 2  
b = 3  
b = a  
print(b)
```

Variables are an Example of an Abstraction

- What is abstraction?
- An example of abstraction is giving names to things, so that the name captures the core of what the thing represents.
- Abstraction is reducing detail to manage complexity.

Variable Declarations & Initializations

- Good Style: Give your variable a **meaningful name** in **CamelCase**.
- **numStudents = 47**
- **numStudents** suggests that the variable holds a number of students.
- Variable names can't have spaces or unusual characters.
- Be careful! If you type a name for a variable and it changes color, that name has already been taken!

Using Variables in Expression

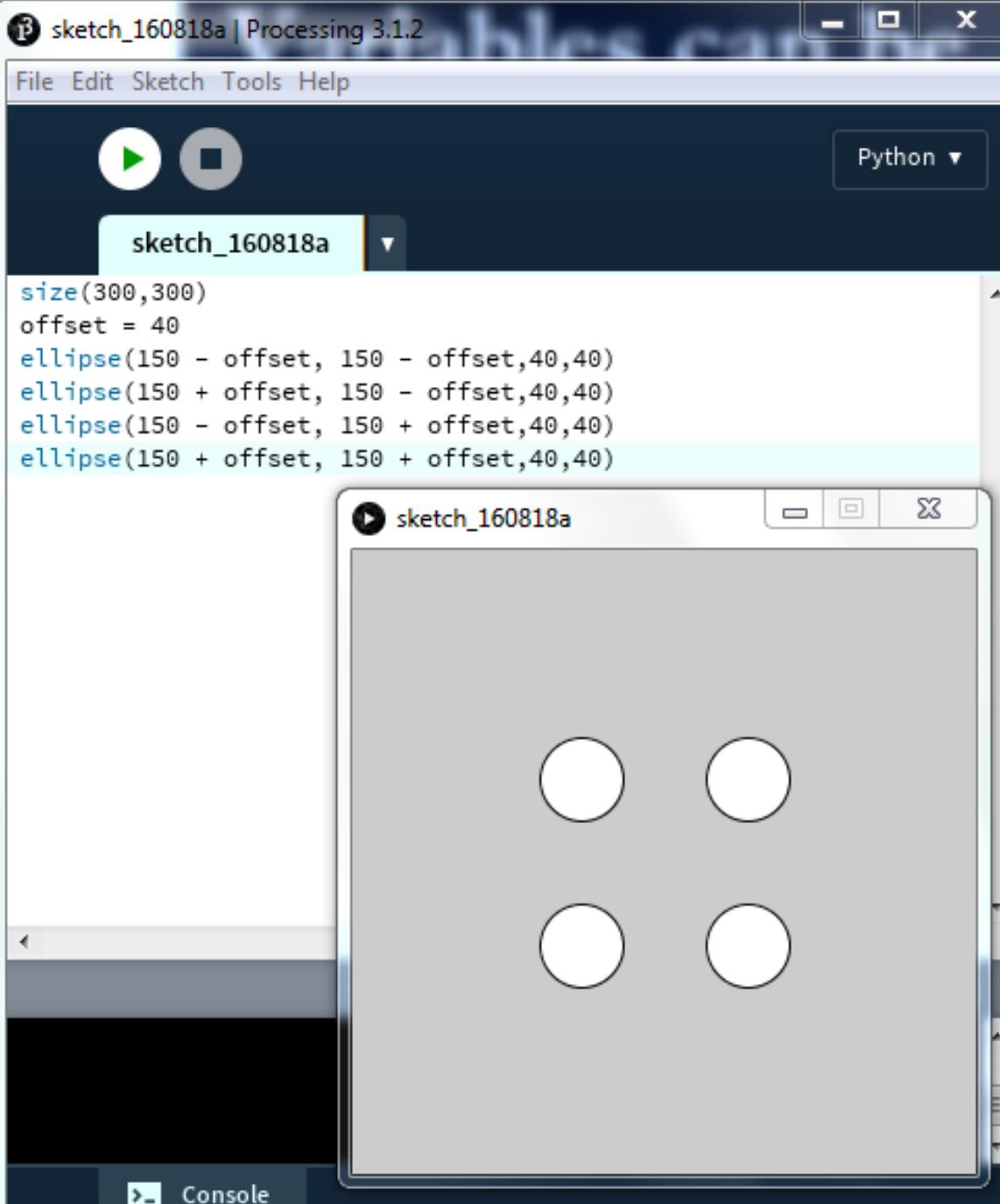
- Once a variable has been assigned a value, you can use it in an expression
- What will be the **x** and **y** coordinates of the center of the ellipse?

```
value1 = 17
```

```
value2 = 13
```

```
ellipse(50+value1, 50-
    value2, 12, 44)
```

Variables Can be Reused as Many Times as You Like.

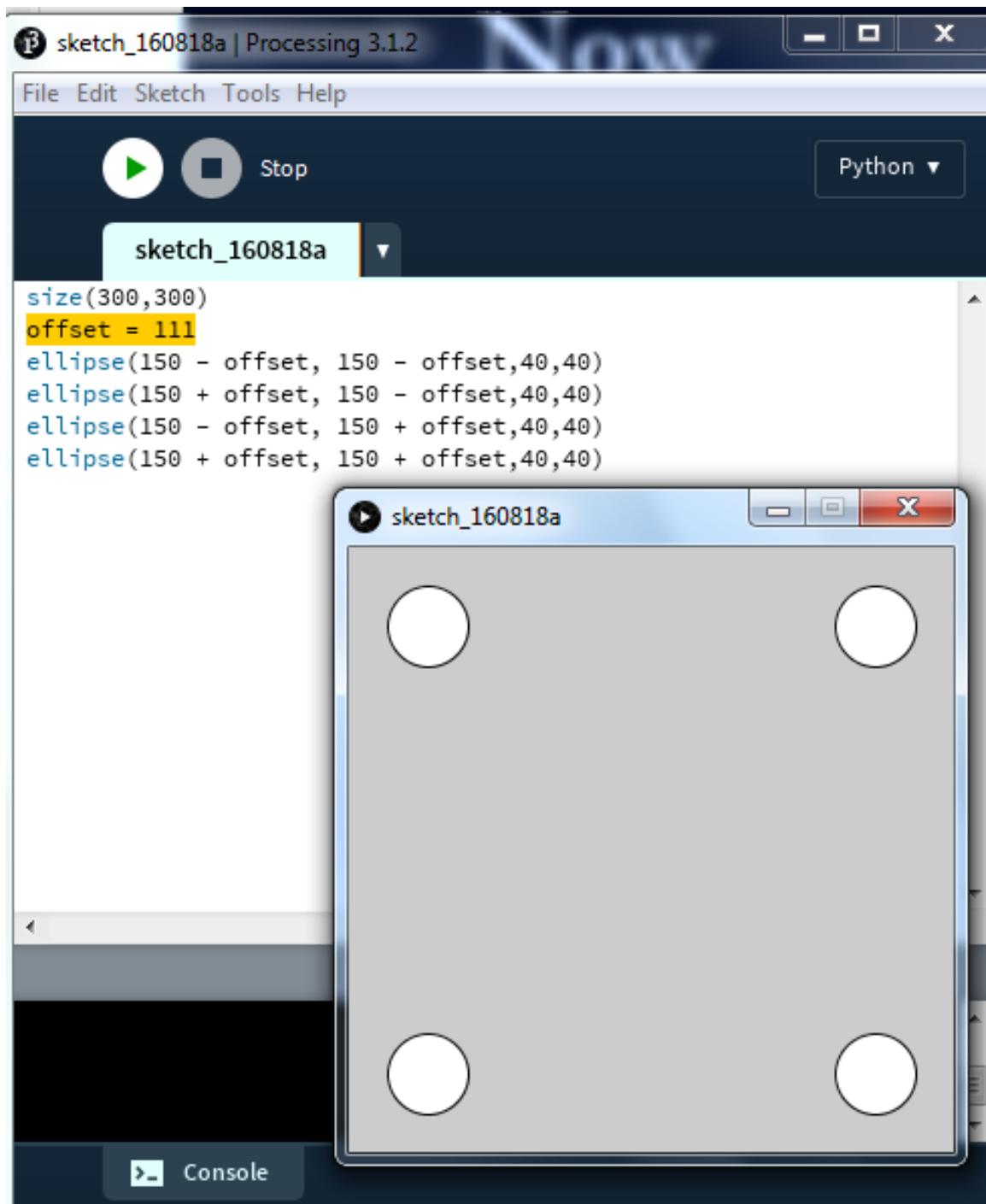


The screenshot shows the Processing IDE interface. The top bar displays "sketch_160818a | Processing 3.1.2". The menu bar includes "File", "Edit", "Sketch", "Tools", and "Help". A dropdown menu on the right says "Python ▾". Below the menu is a toolbar with a play button and a square button. The code editor window contains the following code:

```
size(300,300)
offset = 40
ellipse(150 - offset, 150 - offset,40,40)
ellipse(150 + offset, 150 - offset,40,40)
ellipse(150 - offset, 150 + offset,40,40)
ellipse(150 + offset, 150 + offset,40,40)
```

The preview window titled "sketch_160818a" shows a gray canvas with four white ellipses arranged in a 2x2 grid, centered at (150, 150) with an offset of 40 pixels.

Now,
With Just One
Change. . .



Variables Assignments and `print()` on the AP Exam

- Variable assignments on the AP exam are indicated with an **Arrow** instead of `=`
- Instead of `print()` the exam will use **DISPLAY()**

```
a <- 2
```

```
b <- 3
```

```
b <- a
```

```
DISPLAY(b)
```

Practice Quiz Question

What will the following algorithm print?

x <- 2

x <- x + 1

y <- x + 1

DISPLAY(y)

Practice Quiz Question

A student wrote a program that used the following algorithm:

```
lucky <- 7
DISPLAY(lucky)
luck <- 11
DISPLAY(lucky)
```

The student expected the program to print 7 and 11, but instead it printed 7 twice. Correct one step in the algorithm so the program works as intended.

Lists

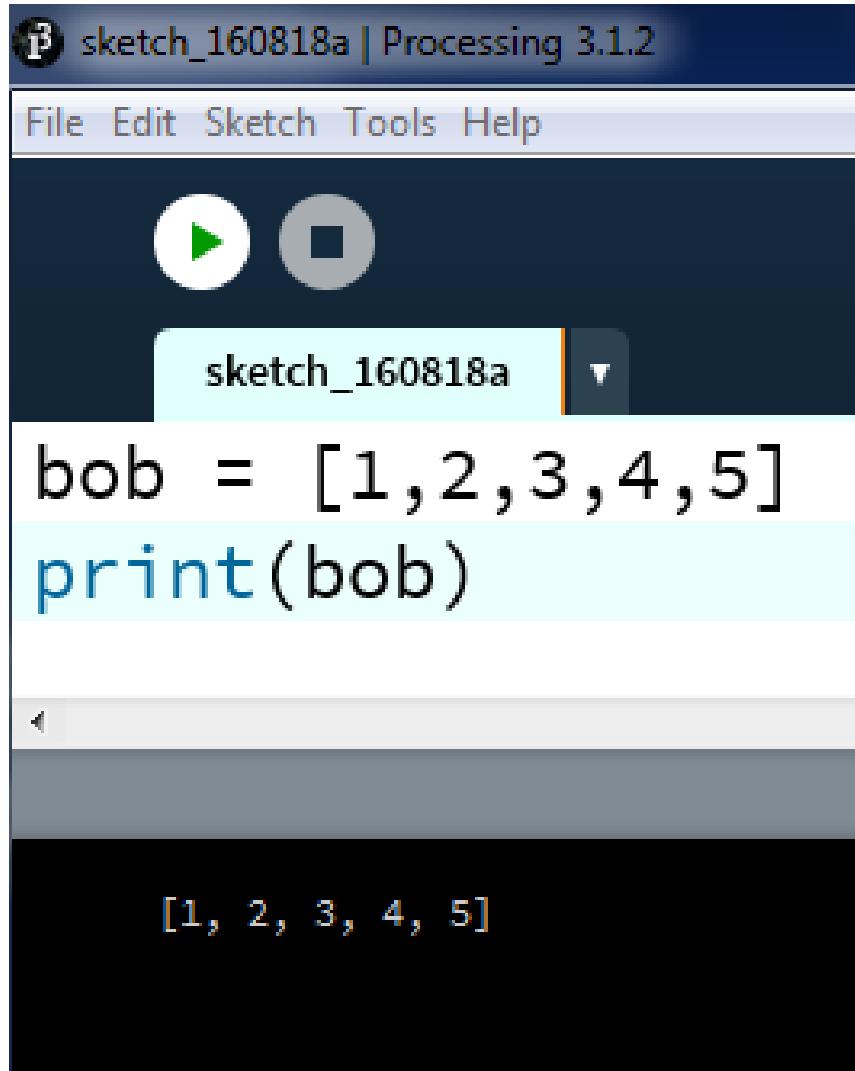
- A list is a group of values.
- One variable (**bob** in this example) can refer to a collection of values.
- When **bob** is printed we can see the entire collection.



```
P sketch_160818a | Processing 3.1.2
File Edit Sketch Tools Help
sketch_160818a
bob = [1,2,3,4,5]
print(bob)
[1, 2, 3, 4, 5]
```

Lists

- A list is an example of **Abstraction**.
- One simple name refers to a complex collection of elements.
- Abstraction is the process of reducing detail to **manage complexity**.



The screenshot shows the Processing 3.1.2 software interface. The title bar reads "sketch_160818a | Processing 3.1.2". The menu bar includes File, Edit, Sketch, Tools, and Help. Below the menu is a toolbar with a play button and a stop button. The code editor window contains the following code:

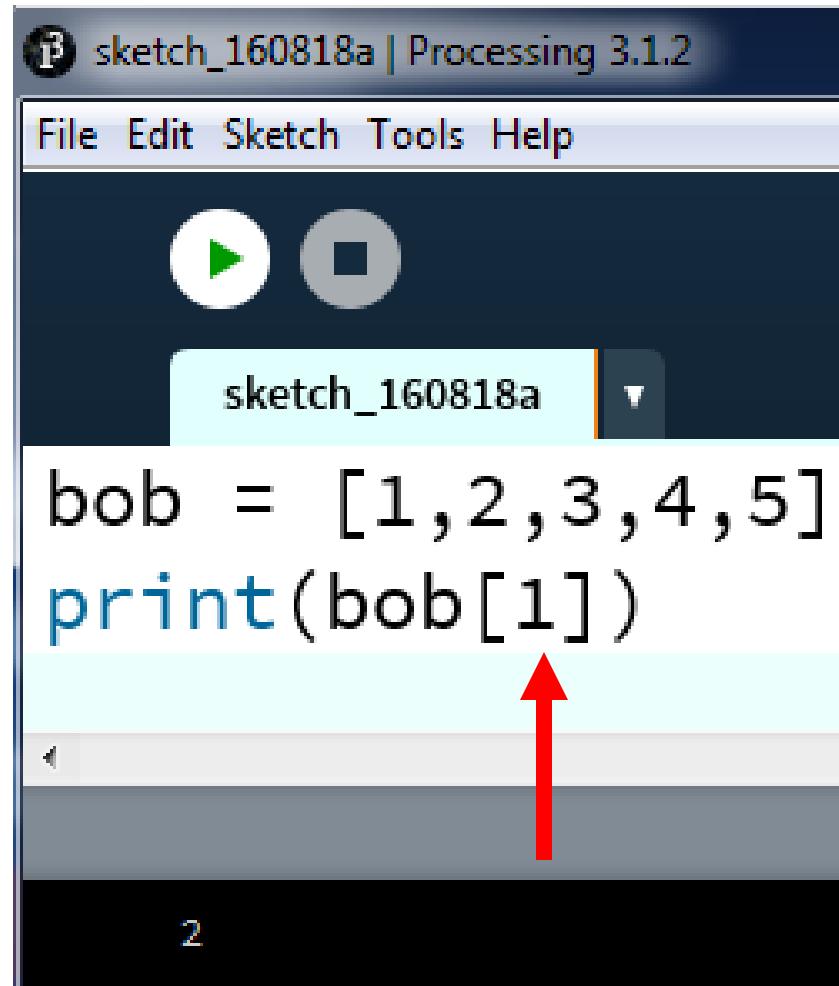
```
bob = [1,2,3,4,5]
print(bob)
```

The output window below the code editor displays the result of the print statement:

```
[1, 2, 3, 4, 5]
```

Lists

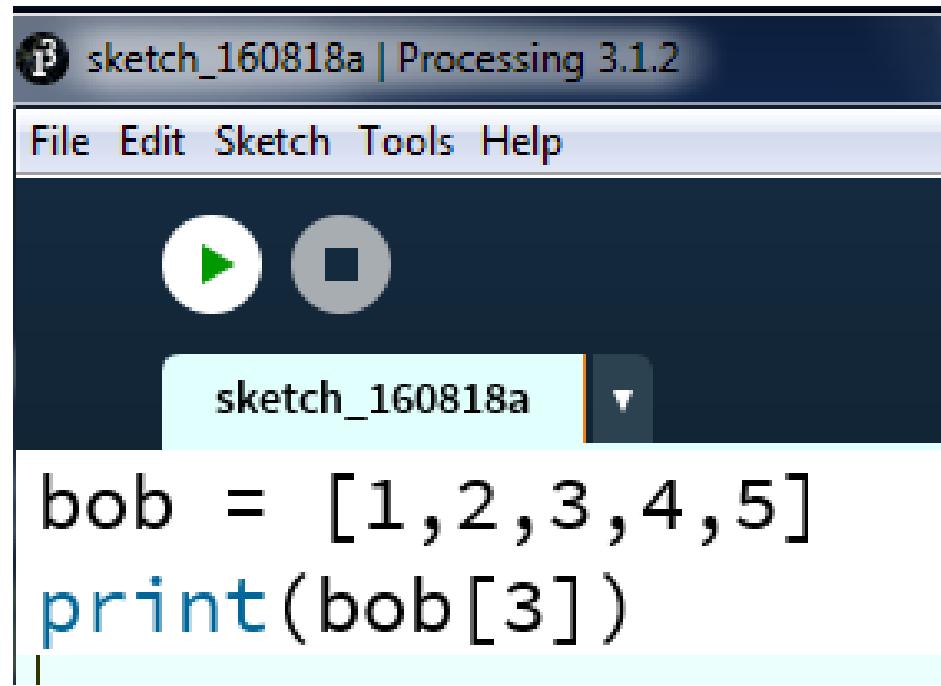
- If I just want to print just one element in **bob** I use **[]**.
- The number in the **[]** is called an *index*.
- It's the position on the one element in the list
- Note that the first element is in position 0!



```
sketch_160818a | Processing 3.1.2
File Edit Sketch Tools Help
sketch_160818a
bob = [1,2,3,4,5]
print(bob[1])
```

Lists

What will be printed?

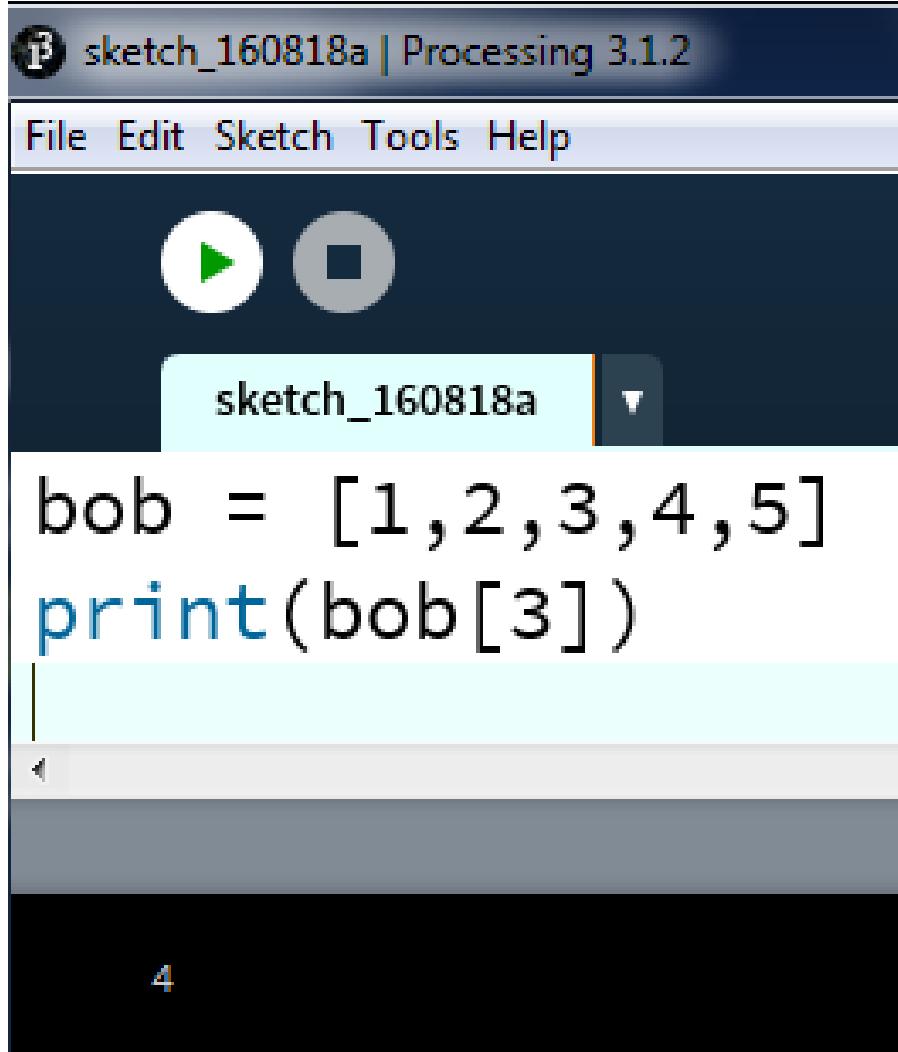


```
sketch_160818a | Processing 3.1.2
File Edit Sketch Tools Help
sketch_160818a ▾
bob = [1,2,3,4,5]
print(bob[3])
|
```

The screenshot shows the Processing IDE. The title bar says "sketch_160818a | Processing 3.1.2". The menu bar includes "File", "Edit", "Sketch", "Tools", and "Help". Below the menu is a toolbar with a play button and a stop button. The sketch name "sketch_160818a" is displayed in a dropdown menu. The code editor contains the following code:
`bob = [1,2,3,4,5]
print(bob[3])`

Lists

4!



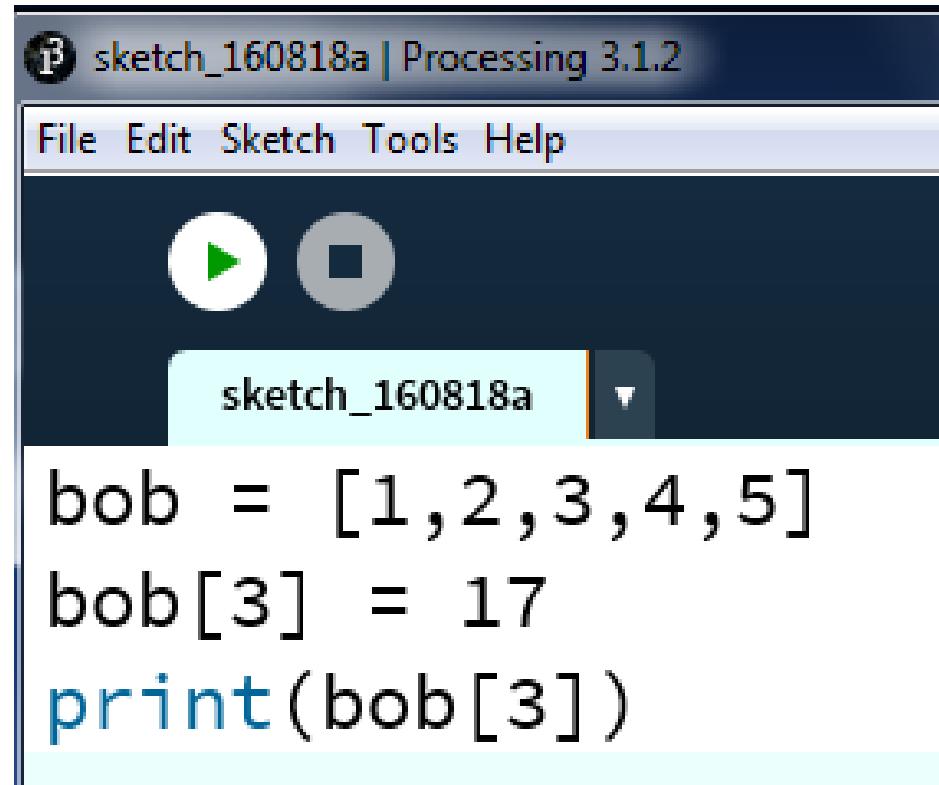
The screenshot shows the Processing 3.1.2 software interface. The title bar reads "sketch_160818a | Processing 3.1.2". The menu bar includes "File", "Edit", "Sketch", "Tools", and "Help". Below the menu are two large circular buttons: a white one with a green play triangle and a grey one with a white square. A dropdown menu is open, showing "sketch_160818a" and a downward arrow icon. The main code area contains the following code:

```
bob = [1,2,3,4,5]
print(bob[3])
```

The output window at the bottom displays the number "4".

Lists

What will be printed now?

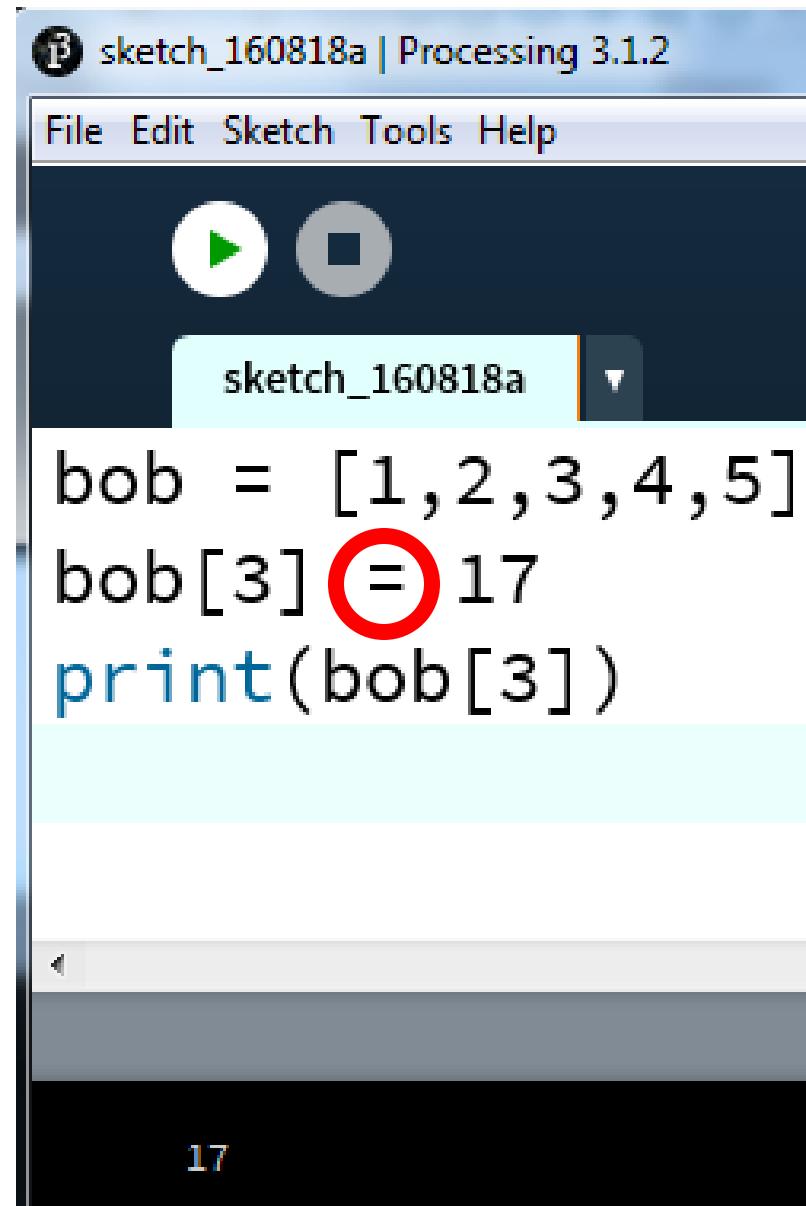


The screenshot shows the Processing 3.1.2 IDE. The title bar reads "sketch_160818a | Processing 3.1.2". The menu bar includes File, Edit, Sketch, Tools, and Help. Below the menu are two large circular buttons: a green play button on the left and a grey square button on the right. The code editor window has a dark header bar with the sketch name "sketch_160818a" and a dropdown arrow. The main code area contains the following:

```
bob = [1,2,3,4,5]
bob[3] = 17
print(bob[3])
```

Lists

- 17!
- Just like any other variable, the value can be changed with an assignment.



The screenshot shows the Processing 3.1.2 software interface. The title bar reads "sketch_160818a | Processing 3.1.2". The menu bar includes File, Edit, Sketch, Tools, and Help. Below the menu is a toolbar with a play button and a stop button. The code editor window contains the following code:

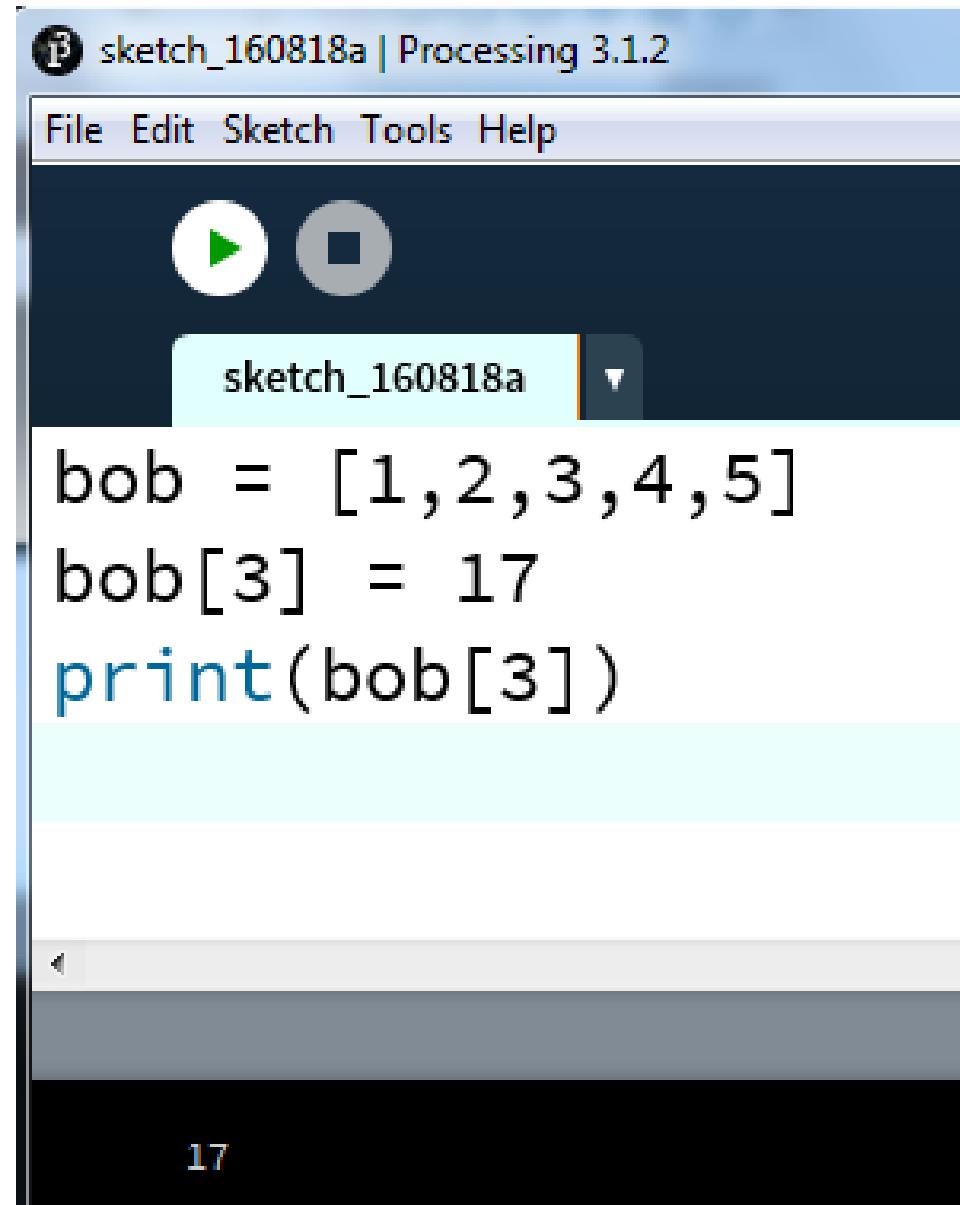
```
bob = [1,2,3,4,5]
bob[3] = 17
print(bob[3])
```

The assignment operator (=) in the line "bob[3] = 17" is highlighted with a red circle. The output window below the code editor is currently empty.

Lists

- On the AP exam an **arrow** indicates assignment.
- The code would be written:

```
bob <- [1,2,3,4,5]
bob[3] <- 17
DISPLAY(bob[3])
```



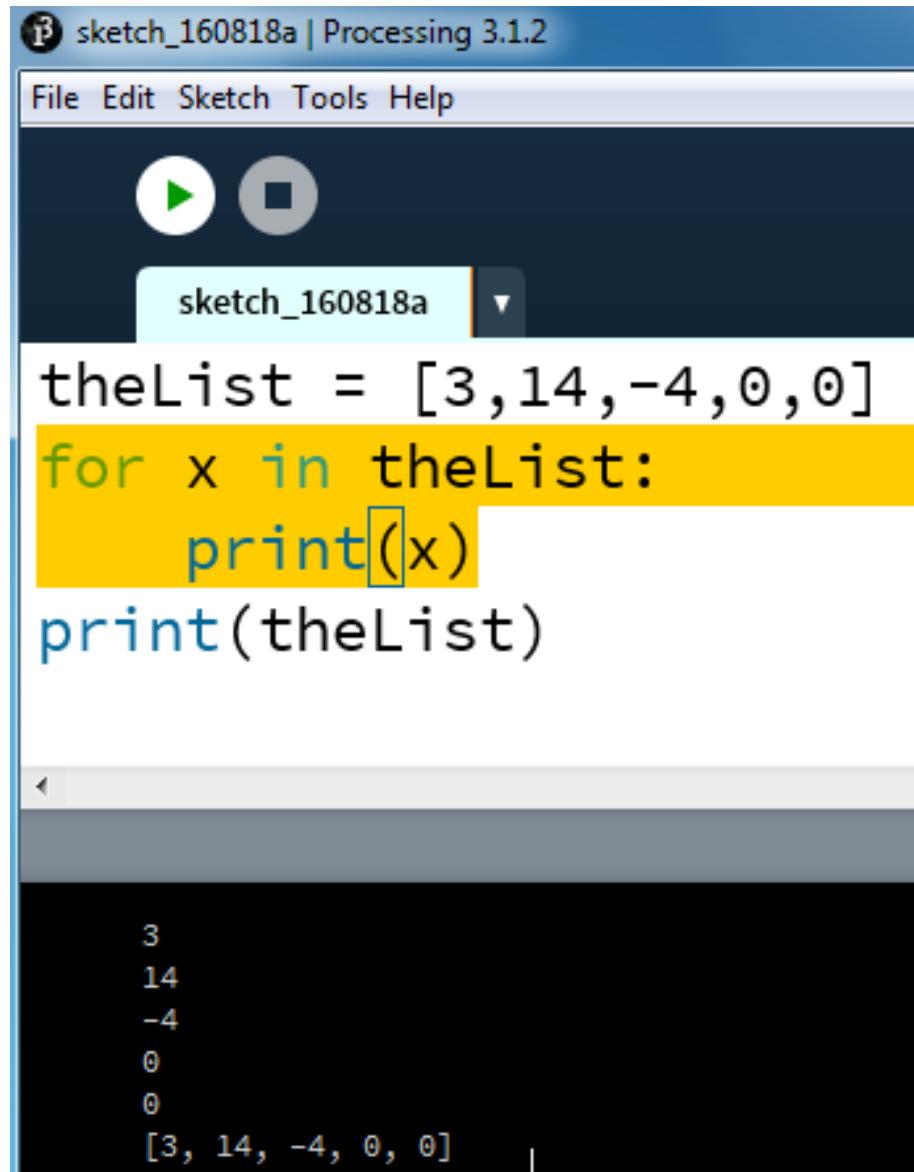
```
sketch_160818a | Processing 3.1.2
File Edit Sketch Tools Help
sketch_160818a
bob = [1,2,3,4,5]
bob[3] = 17
print(bob[3])
17
```

Practice Quiz Question

```
list <- [17,1,-45,37,14]
list[0] <- 12
list[4] <- list[0]/2
DISPLAY(list)
```

Iteration: Traversing a list

- The **highlighted code** prints each element of the list one at a time.
- Visiting all of the elements in the list one at a time is called “Traversing the list.”
- The easiest way to do this in Python is with a **for** each loop.
- One by one, each element in **theList** will be copied into **x**
- This is an example of *iteration*.



The screenshot shows the Processing 3.1.2 environment. The top bar displays the title "sketch_160818a | Processing 3.1.2" and a menu bar with File, Edit, Sketch, Tools, and Help. Below the menu is a toolbar with a play button and a stop button. The code editor window contains the following Python-like pseudocode:

```
theList = [3,14,-4,0,0]
for x in theList:
    print(x)
print(theList)
```

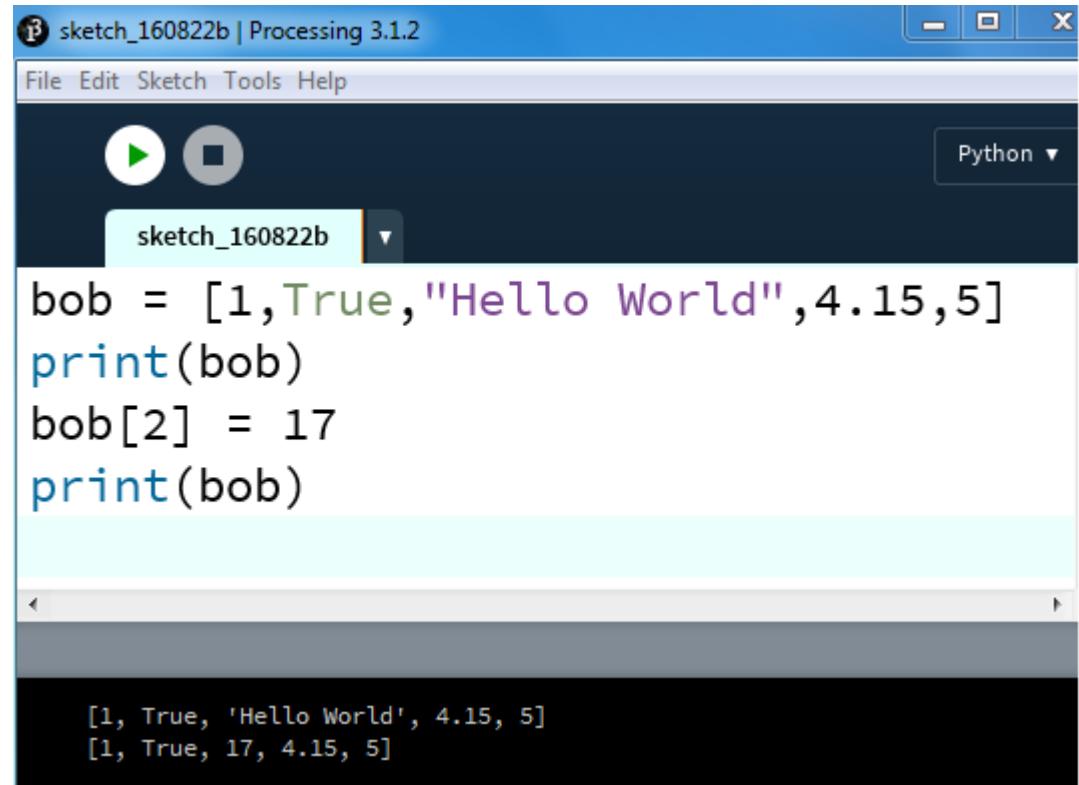
The line "for x in theList:" is highlighted in yellow. When the play button is pressed, the terminal window at the bottom shows the output:

```
3
14
-4
0
0
[3, 14, -4, 0, 0]
```

Lists

Lists can contain just about anything:

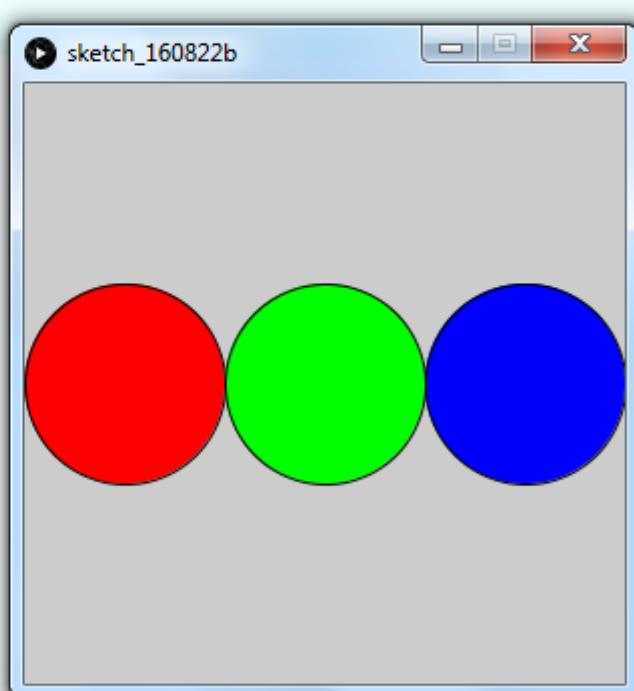
- Numbers
- Text
- Even colors!



```
sketch_160822b | Processing 3.1.2
File Edit Sketch Tools Help
Python ▾
sketch_160822b
bob = [1,True,"Hello World",4.15,5]
print(bob)
bob[2] = 17
print(bob)

[1, True, 'Hello World', 4.15, 5]
[1, True, 17, 4.15, 5]
```

A List of Colors



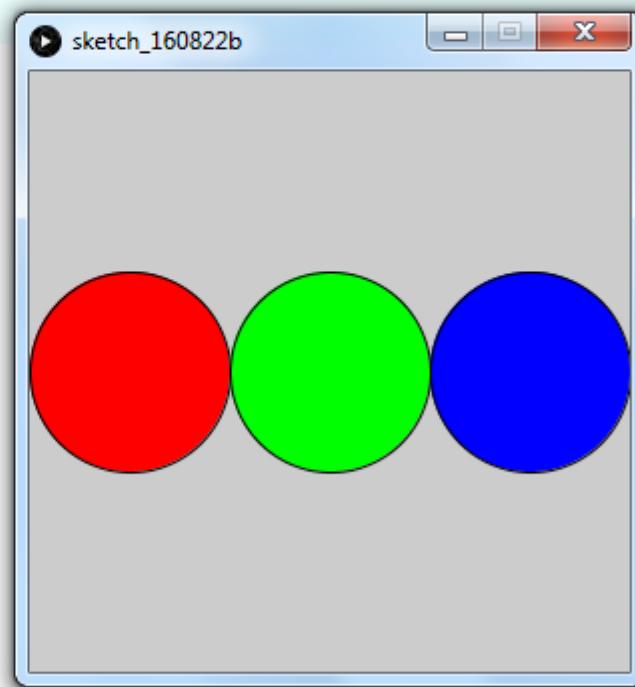
```
size(300,300)
colors = [color(255,0,0),color(0,255,0),color(0,0,255)]
x = 50
for col in colors:
    fill(col)
    ellipse(x,150,100,100)
    x = x + 100
```

Indentation and the Colon

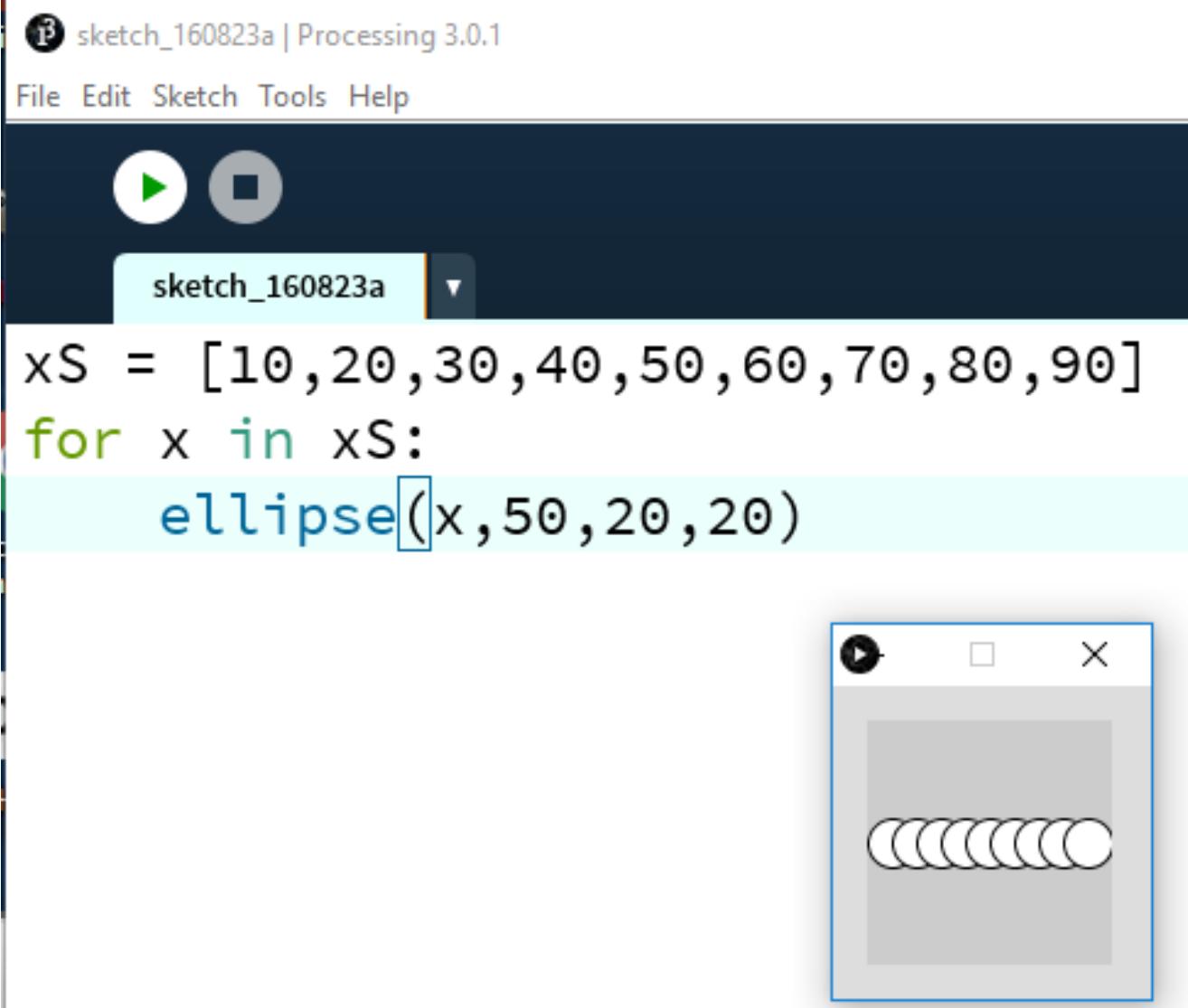


```
size(300,300)
colors = [color(255,0,0),color(0,255,0),color(0,0,255)]
x = 50
for col in colors:
    fill(col)
    ellipse(x,150,100,100)
    x = x + 100
```

The code in the Processing IDE creates a sketch titled "sketch_160822b". It starts with a call to size(300,300). A variable "colors" is defined as an array containing three color objects: red, green, and blue. The variable "x" is initialized to 50. A for loop iterates over the "colors" array, filling each ellipse with the current color and drawing it at position (x, 150) with a diameter of 100 pixels. After each iteration, "x" is incremented by 100. Four yellow arrows point from the text "for col in colors:" to the opening brace of the for loop, indicating the scope of the loop.



A List of x Coordinates



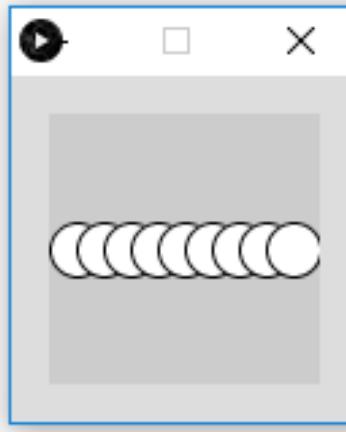
sketch_160823a | Processing 3.0.1

File Edit Sketch Tools Help

sketch_160823a

```
xs = [10,20,30,40,50,60,70,80,90]
for x in xs:
    ellipse(x,50,20,20)
```

The code defines a list of x-coordinates and uses a for loop to draw an ellipse at each position.



On the AP Exam, for Each Loops will be Written this Way.

It's pretty close to Python, except Python doesn't use the word "EACH."

Text:

```
FOR EACH item IN list
{
    <block of statements>
}
```

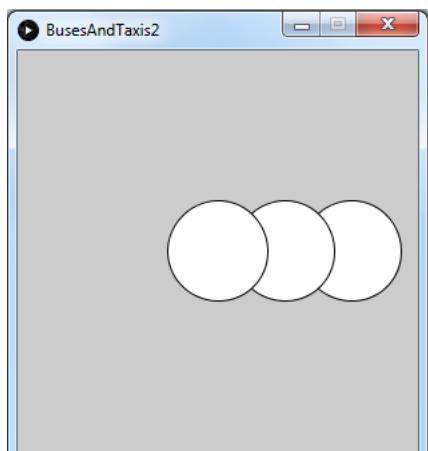
The variable `item` is assigned the value of each element of `list` sequentially, in order from the first element to the last element.

The code in `block of statements` is executed once for each assignment of `item`.

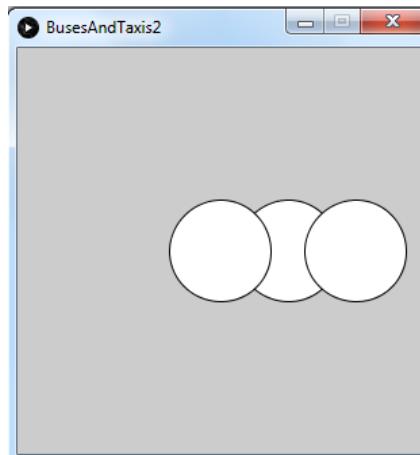
Find the Output that Best Matches the Algorithm.

1. Set the size of the drawing canvas to 300 by 300.
2. `xS <- [250,150,200]`
3. FOR EACH `x` IN `xS` draw an ellipse at (`x`,150)

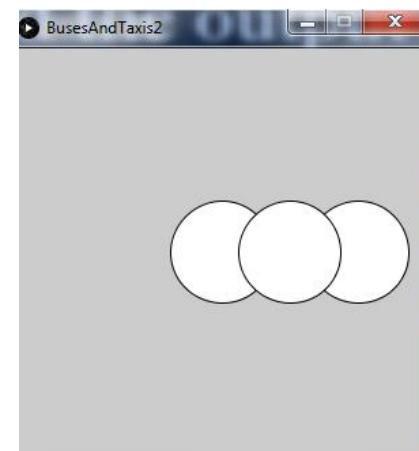
A



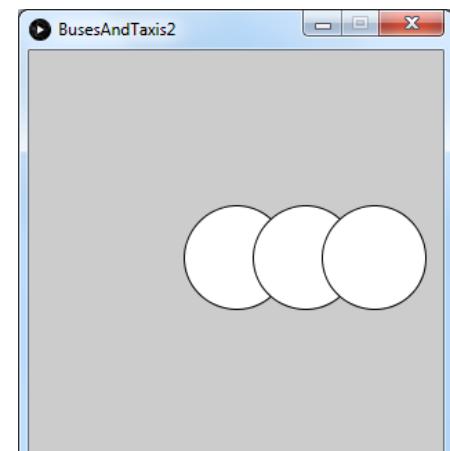
B



C



D



Lists: append () and len ()

The screenshot shows the Arduino IDE interface. The menu bar includes File, Edit, Sketch, Tools, and Help. Below the menu is a toolbar with a play button, a square button, and a "Run" label. The sketch window contains the following code:

```
bob = []
print(len(bob))
bob.append(6)
bob.append(2.69)
bob.append(3.14)
print(len(bob))
print(bob)
```

The screenshot shows the output window of the Arduino IDE. The output is:

```
0
3
[6, 2.69, 3.14]
```

Lists: pop ()

```
bob = ['a', b, 'c']
bob.append('d')
bob.pop(1) #remove element at
index 1
print(bob)
bob.pop() #remove last
element
print(bob)
```

```
[a, c, d]
[a, c]
```

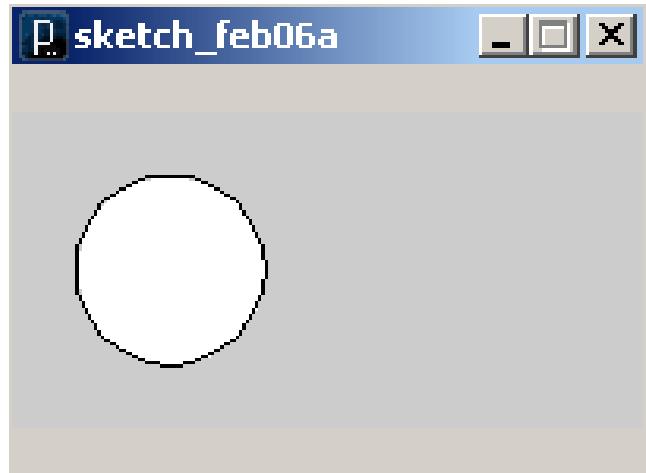
Practice Quiz Question:

Find the output

```
primates =  
["gorilla", "baboon", "chimp"]  
primates.append("mandrill")  
primates[1] = "gibbon"  
primates.pop(1)  
print(primates)  
primates.pop  
print(primates)
```

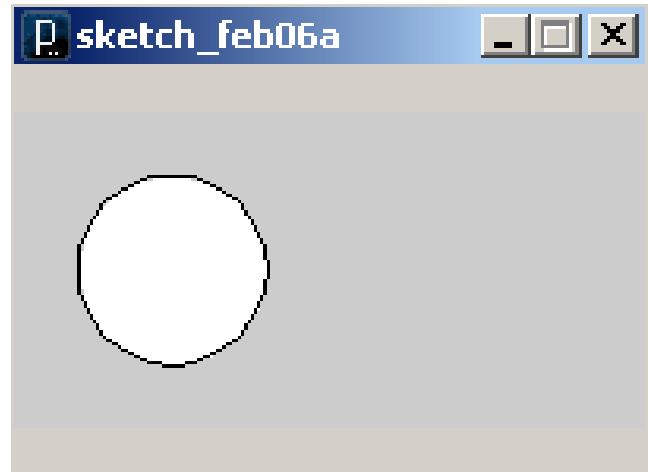
"Moving" a Circle

```
size(200,100)  
x = 50  
y = 50  
ellipse(x,y,60,60)
```



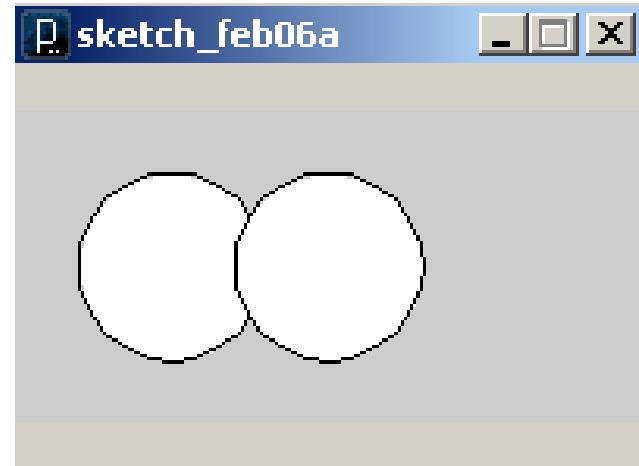
"Moving" a Circle

```
size(200,100)  
x = 50  
y = 50  
ellipse(x,y,60,60)  
x = x + 50
```



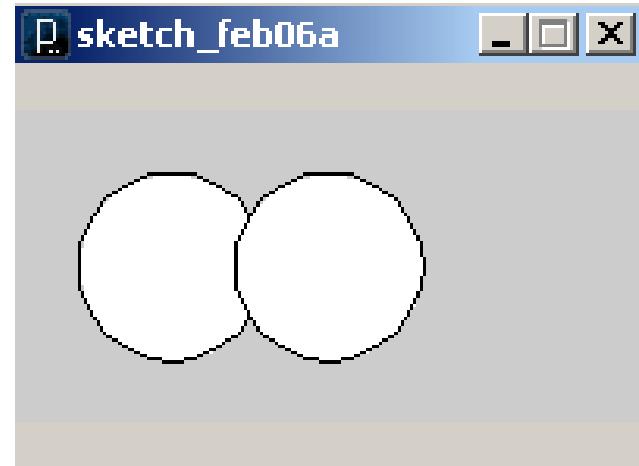
"Moving" a Circle

```
size(200,100)  
x = 50  
y = 50  
ellipse(x,y,60,60)  
x = x + 50  
ellipse(x,y,60,60)
```



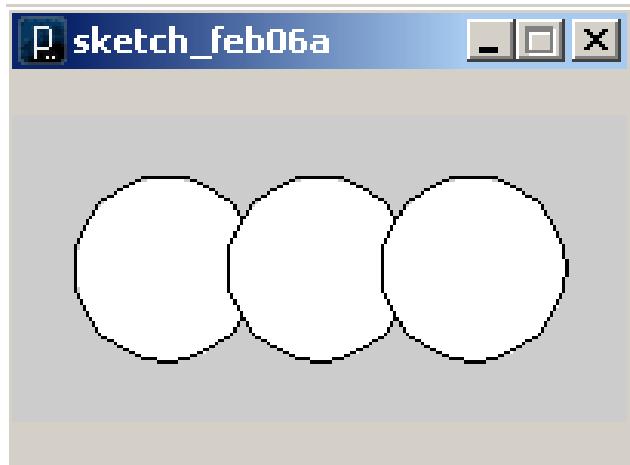
"Moving" a Circle

```
size(200,100)  
x = 50  
y = 50  
ellipse(x,y,60,60)  
x = x + 50  
ellipse(x,y,60,60)  
x = x + 50
```



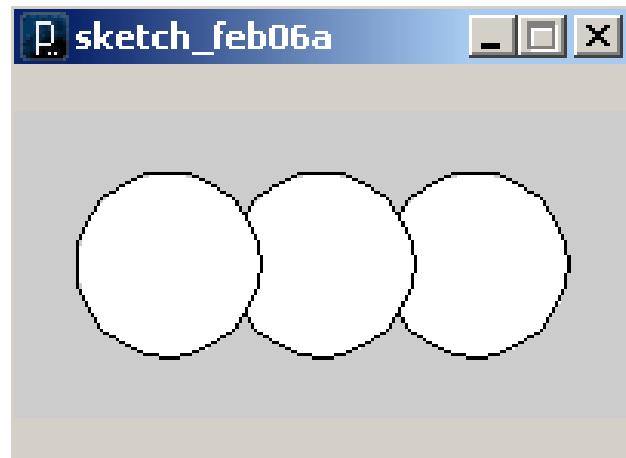
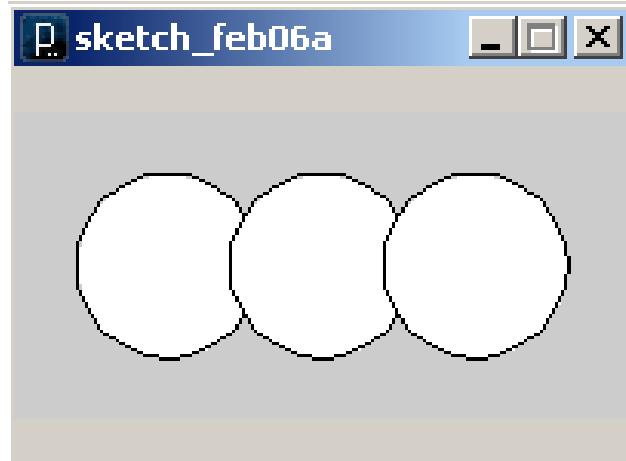
"Moving" a Circle

```
size(200,100)  
x = 50  
y = 50  
ellipse(x,y,60,60)  
x = x + 50  
ellipse(x,y,60,60)  
x = x + 50  
ellipse(x,y,60,60)
```



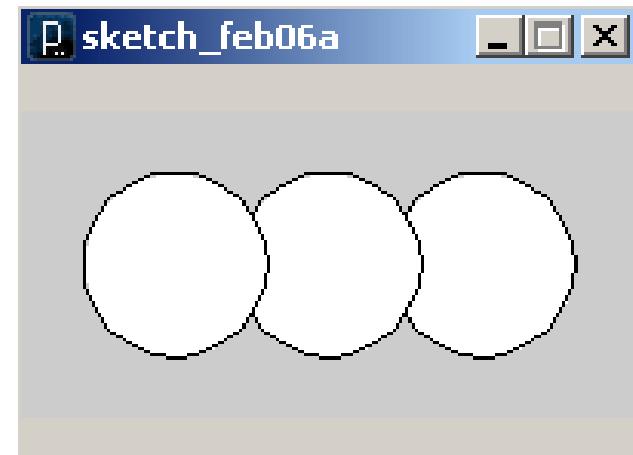
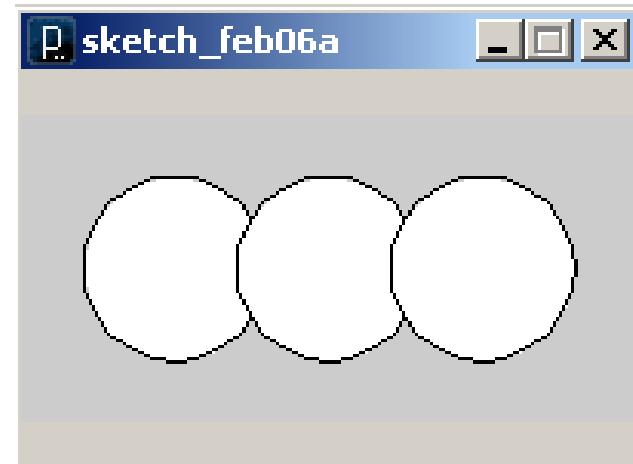
Notice the Difference?

```
size(200,100)  
x = 50  
y = 50  
ellipse(x,y,60,60)  
x = x + 50  
ellipse(x,y,60,60)  
x = x + 50  
ellipse(x,y,60,60)
```



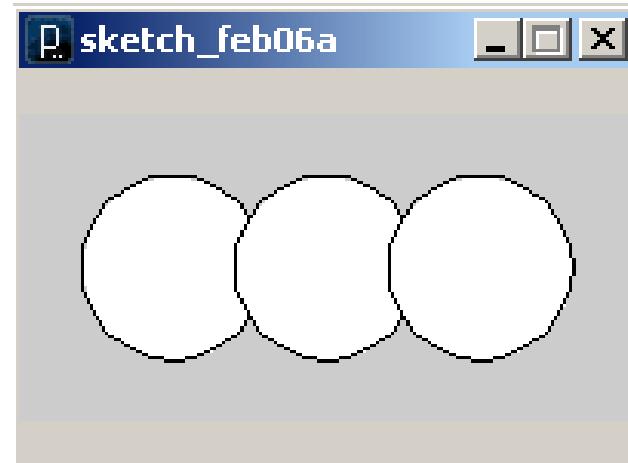
The top picture drew the left circle *first*, while the bottom drew the left circle *last*.

```
size(200,100)  
x = 50  
y = 50  
ellipse(x,y,60,60)  
x = x + 50  
ellipse(x,y,60,60)  
x = x + 50  
ellipse(x,y,60,60)
```



The bottom picture doesn't match the output of this program.

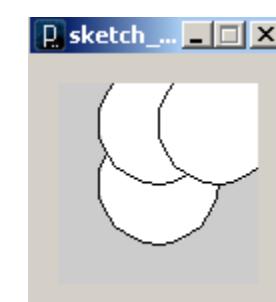
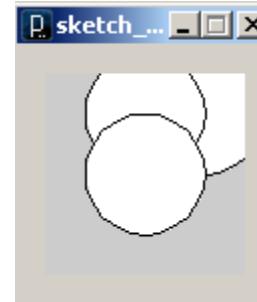
```
size(200,100)  
x = 50  
y = 50  
ellipse(x,y,60,60)  
x = x + 50  
ellipse(x,y,60,60)  
x = x + 50;  
ellipse(x,y,60,60)
```



Find the Output that Best Matches the Algorithm.

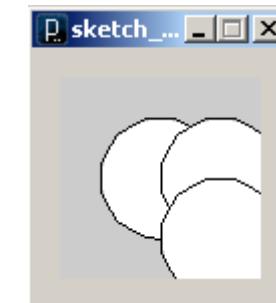
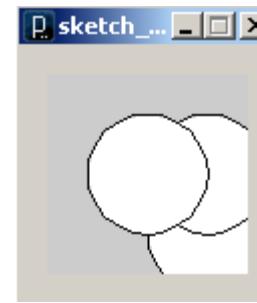
1. $x \leftarrow 50$
2. $y \leftarrow 50$
3. Draw an ellipse at (x, y)
4. $x \leftarrow x + 30$
5. Draw an ellipse at (x, y)
6. $y \leftarrow y - 30$
7. Draw an ellipse at (x, y)

A



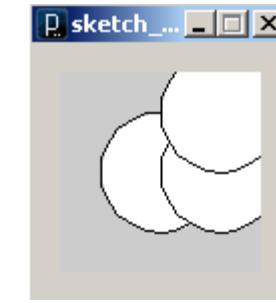
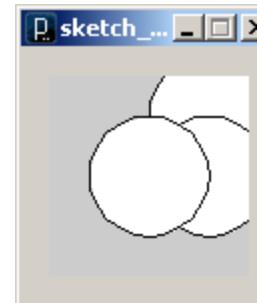
B

C



D

E

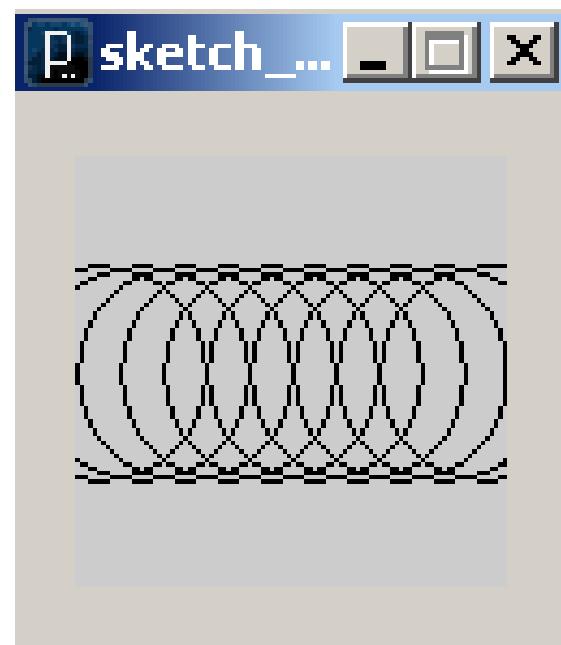


F

Loops Make Things Repeat

- *Looping* means *repeating* over and over
- *Repetition* is one of the fundamentals of programming
- If you wanted to display 10 **circles** to the screen, you could copy and paste code like this:

```
noFill()
ellipse(5,50,50,50)
ellipse(15,50,50,50)
ellipse(25,50,50,50)
ellipse(35,50,50,50)
ellipse(45,50,50,50)
ellipse(55,50,50,50)
ellipse(65,50,50,50)
ellipse(75,50,50,50)
ellipse(85,50,50,50)
ellipse(95,50,50,50)
```



Repetition: “Loops”

- On the other hand, if you wanted to display **100** or **1000** or even more circles to the screen, using copy and paste would be tedious.
- A much better way is to use a *loop*.

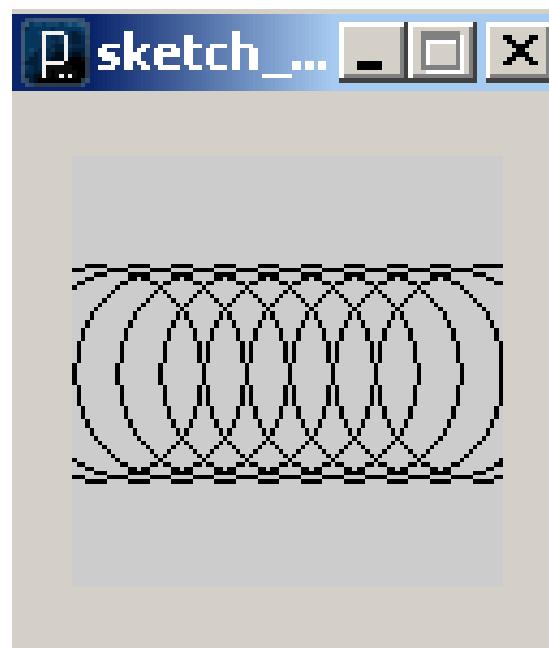
while Loops

- For the AP exam we need to know about both **for** each and **while** loops.
- **while** loops often use a variable to keep track of the number of iterations.
- The variable has a *start*, a *stop*, and a *step*. The *step* is a way of progressing from *start* to *stop*.

What is Changing?

- Each of the circles are identical except for one small change – each circle has a different x coordinate.

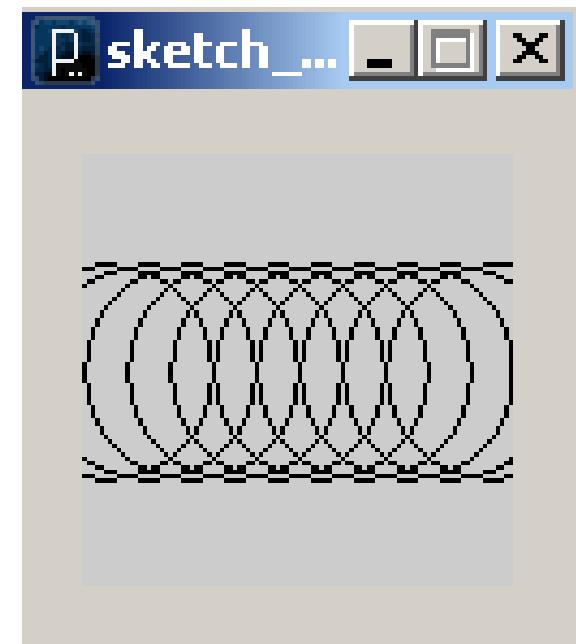
```
noFill()  
ellipse(5,50,50,50)  
ellipse(15,50,50,50)  
ellipse(25,50,50,50)  
ellipse(35,50,50,50)  
ellipse(45,50,50,50)  
ellipse(55,50,50,50)  
ellipse(65,50,50,50)  
ellipse(75,50,50,50)  
ellipse(85,50,50,50)  
ellipse(95,50,50,50)
```



To Store a Changing Value Use a Variable

We could make a variable for the changing x coordinate.

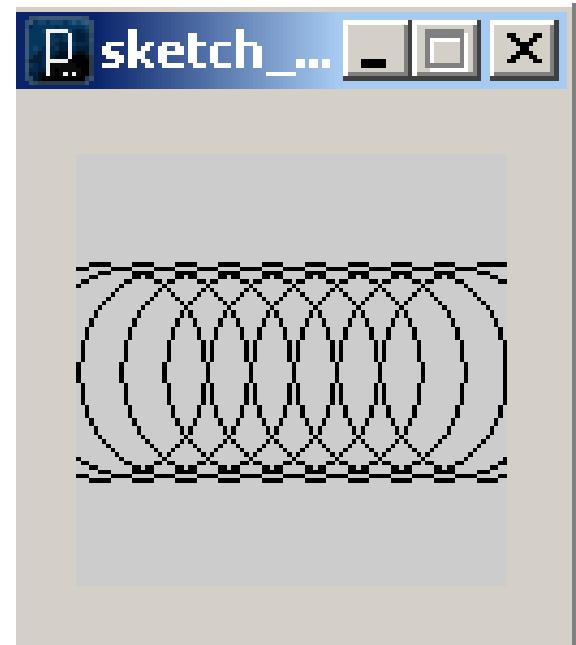
```
noFill()  
x = 5  
ellipse(x,50,50,50)  
x = x + 10  
ellipse(x,50,50,50)  
x = x + 10  
ellipse(x,50,50,50)  
# and so on. . .
```



Here it is with a **while** loop

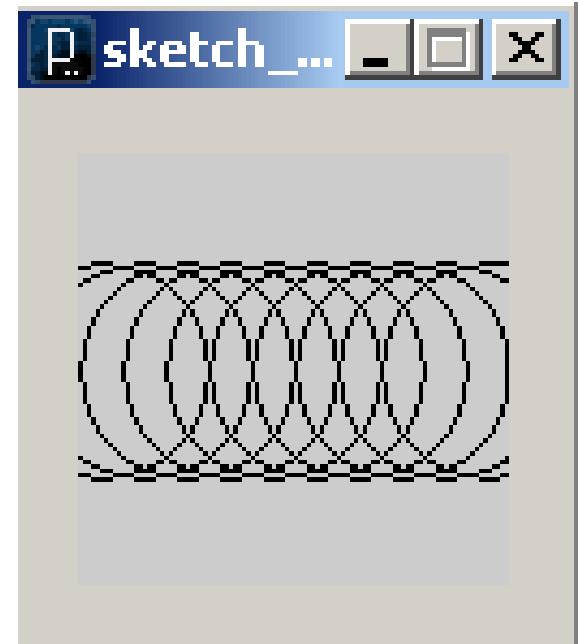
This is the entire program!

```
noFill()
x = 5
while x < 96:
    ellipse(x,50,50,50)
    x = x + 10
```



Indentation and the Colon

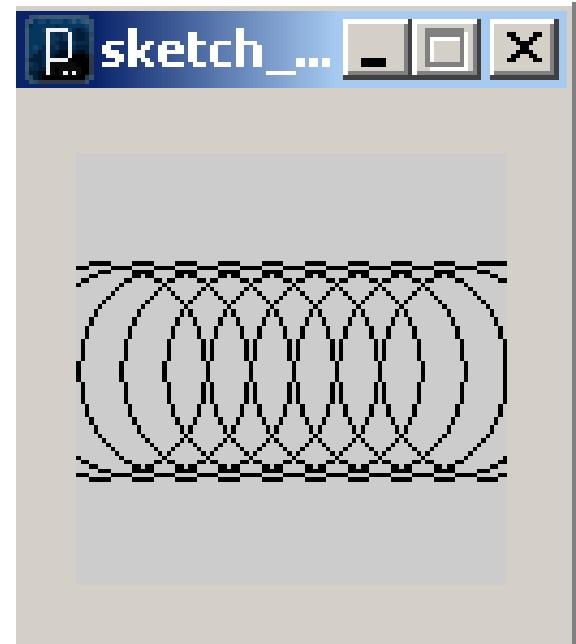
```
noFill()
x = 5
while(x < 96):
    → ellipse(x,50,50,50)
    → x = x + 10
```



Start, Stop and Step

The circles will have x coordinates of 5, 15, 25, 35, 45, 55, 65, 75, 85 and 95.

```
noFill()  
x = 5  
while x < 96:  
    ellipse(x,50,50,50)  
    x = x + 10
```

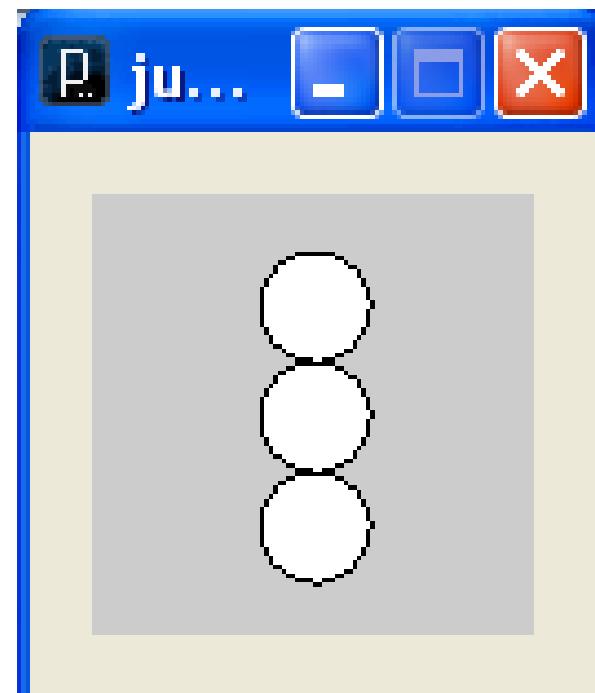


How many ellipses will this program make?

```
y = 25
while y < 76:
    ellipse(50,y,25,25)
    y = y + 25
```

How many ellipses will this program make?

```
y = 25
while y < 76:
    ellipse(50,y,25,25)
    y = y + 25
```



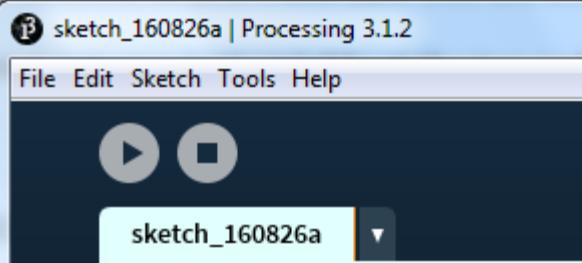
A Pattern of Ellipses of Different Widths.

The image shows the Processing 3.1.2 software interface. On the left, the code editor window displays the following Pseudocode:

```
smooth()
noFill()
stroke(240,20,229,175)
w = 0
while w < 300:
    ellipse(150,150,w,100)
    w = w + 10
```

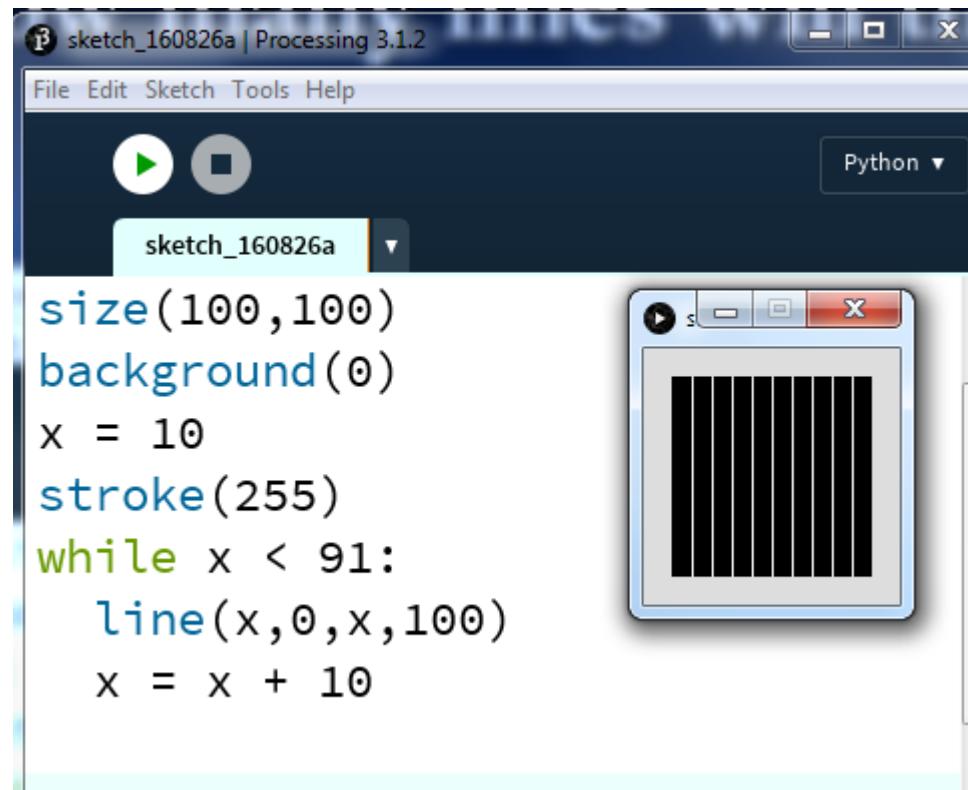
On the right, the preview window shows a black canvas with a series of concentric ellipses centered at (150, 150). The ellipses increase in width from 100 to 300 pixels, creating a pattern that looks like a stylized eye or a lens. The ellipses are drawn with a magenta stroke color.

How many ellipses will this program make?



```
sketch_160826a | Processing 3.1.2
File Edit Sketch Tools Help
sketch_160826a ▾
size(100,100)
background(0)
x = 10
stroke(255)
while x < 91:
    line(x,0,x,100)
    x = x + 10
```

How many lines will this program make?

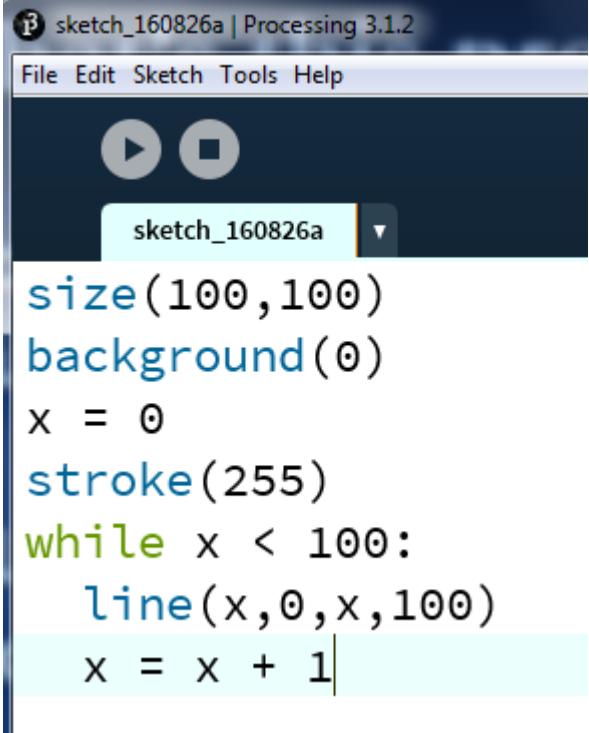


The image shows a screenshot of the Processing 3.1.2 software interface. The title bar reads "sketch_160826a | Processing 3.1.2". The menu bar includes File, Edit, Sketch, Tools, and Help. A toolbar with a play button and a stop button is visible. A dropdown menu shows "sketch_160826a" and a "Python" dropdown set to "Python". The code area contains the following Python code:

```
size(100,100)
background(0)
x = 10
stroke(255)
while x < 91:
    line(x,0,x,100)
    x = x + 10
```

To the right, a preview window displays a black square with 10 white vertical lines spaced evenly from left to right, corresponding to the 10 iterations of the loop.

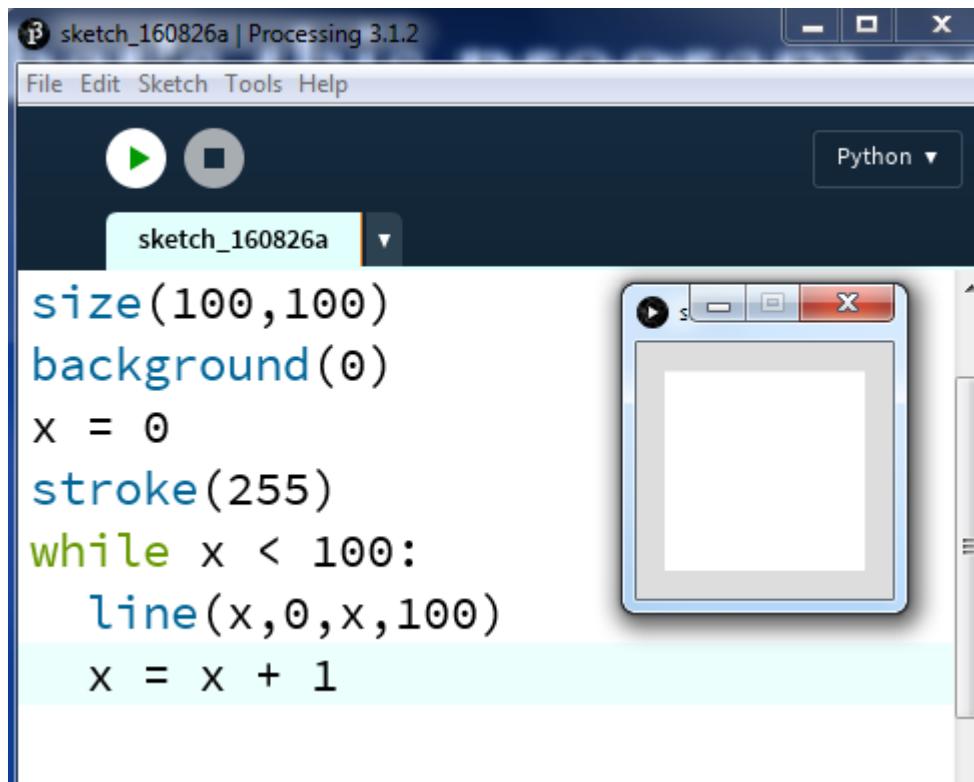
What's this Program Going to Do?



The image shows a screenshot of the Processing 3.1.2 software interface. The title bar reads "sketch_160826a | Processing 3.1.2". Below the title bar is a menu bar with "File", "Edit", "Sketch", "Tools", and "Help". Underneath the menu bar are two large circular buttons: a play button on the left and a stop button on the right. The main area of the window is titled "sketch_160826a" and contains the following Processing code:

```
size(100,100)
background(0)
x = 0
stroke(255)
while x < 100:
    line(x,0,x,100)
    x = x + 1
```

Where is the Black Background?

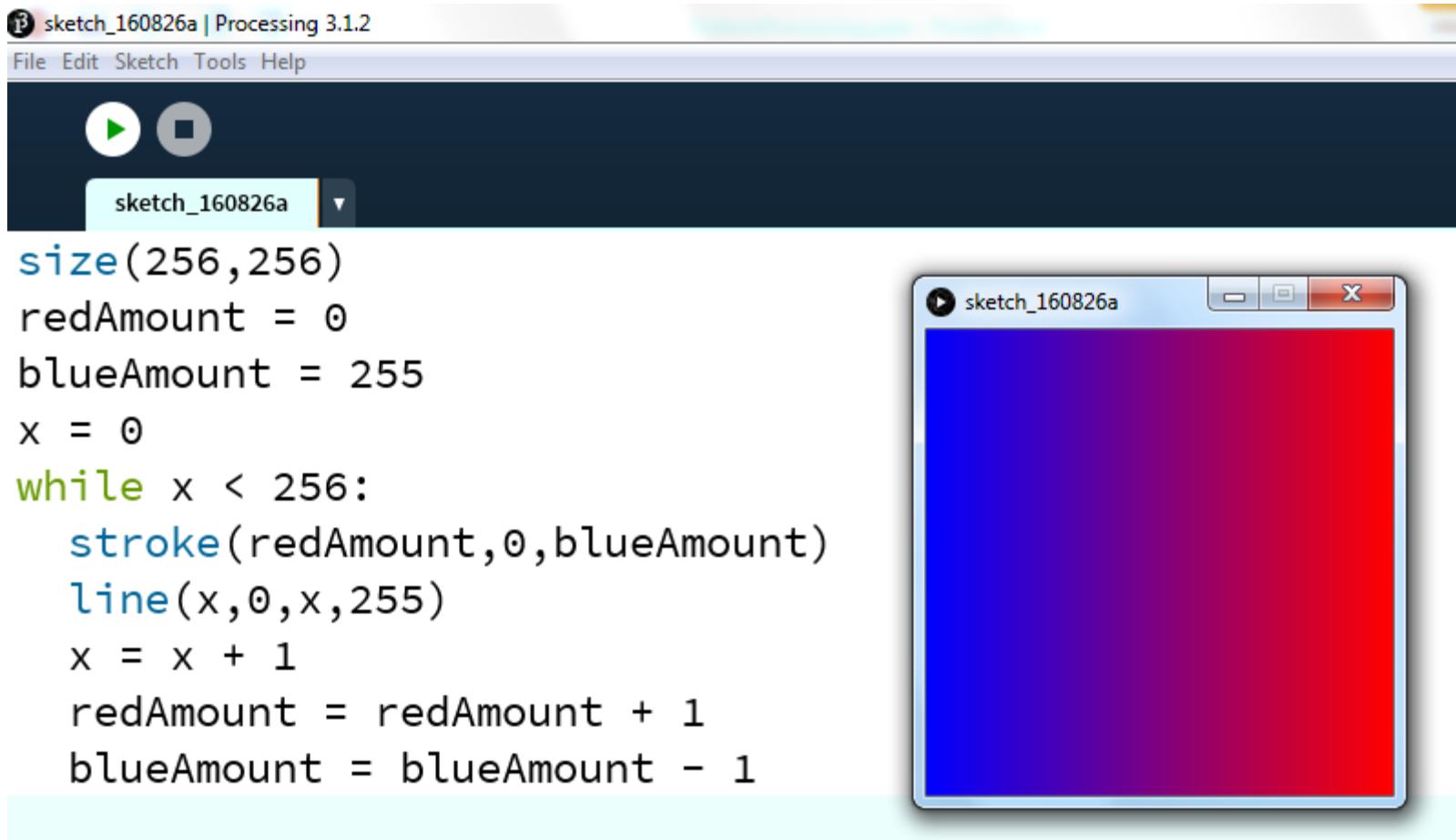


```
sketch_160826a | Processing 3.1.2
File Edit Sketch Tools Help
Python ▾
sketch_160826a
size(100,100)
background(0)
x = 0
stroke(255)
while x < 100:
    line(x,0,x,100)
    x = x + 1
```

We can extend this idea of filling the screen with different lines to make a *gradient*.

(demo)

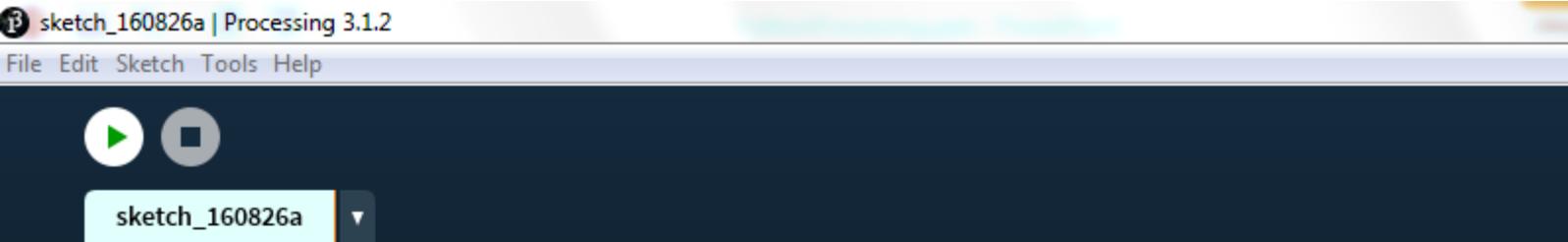
A Program that Uses a Loop to Make a Gradient in the Background.



The image shows the Processing IDE interface with a sketch titled "sketch_160826a". The code uses a while loop to draw a horizontal gradient line from blue to red across the entire width of the canvas. The sketch window displays a vertical color gradient from blue on the left to red on the right.

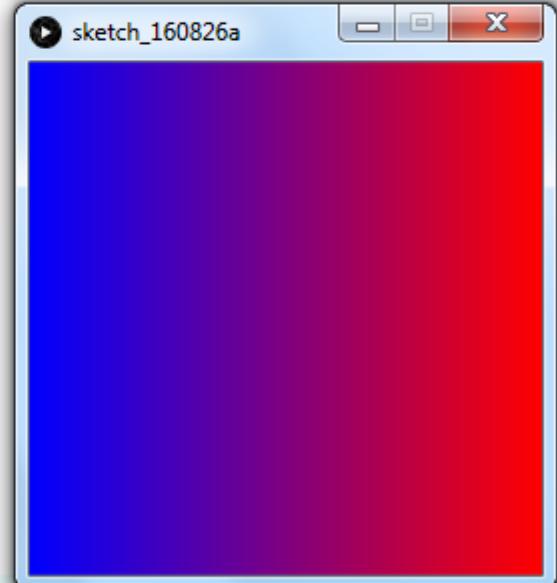
```
size(256,256)
redAmount = 0
blueAmount = 255
x = 0
while x < 256:
    stroke(redAmount,0,blueAmount)
    line(x,0,x,255)
    x = x + 1
    redAmount = redAmount + 1
    blueAmount = blueAmount - 1
```

We can have additional variables to change as well.



```
sketch_160826a | Processing 3.1.2
File Edit Sketch Tools Help
sketch_160826a
```

```
size(256,256)
redAmount = 0
blueAmount = 255
x = 0
while x < 256:
    stroke(redAmount,0,blueAmount)
    line(x,0,x,255)
    x = x + 1 ←
    redAmount = redAmount + 1 ←
    blueAmount = blueAmount - 1 ←
```



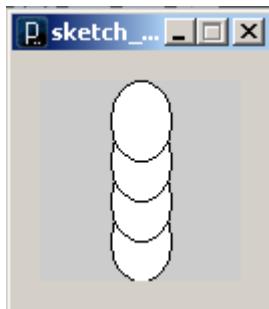
The code demonstrates how to create a vertical gradient from blue at the top to red at the bottom. It uses a while loop to iterate through each pixel from x=0 to x=256. For each pixel, it sets the stroke color to a shade where redAmount increases by 1 and blueAmount decreases by 1. The resulting image shows a smooth transition from blue to red across the entire height of the canvas.

Practice Quiz Question

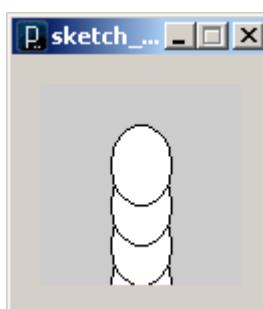
Find the Output.

```
numCircles = 1  
  
while numCircles < 5:  
    ellipse(50,numCircles * 20,30,40)  
  
    numCircles = numCircles + 1
```

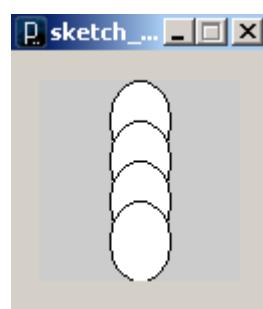
A



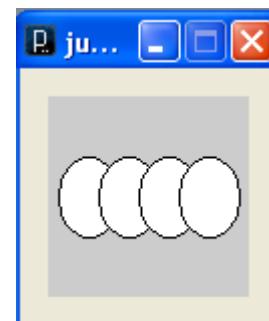
B



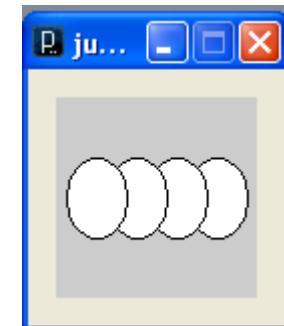
C



D



E

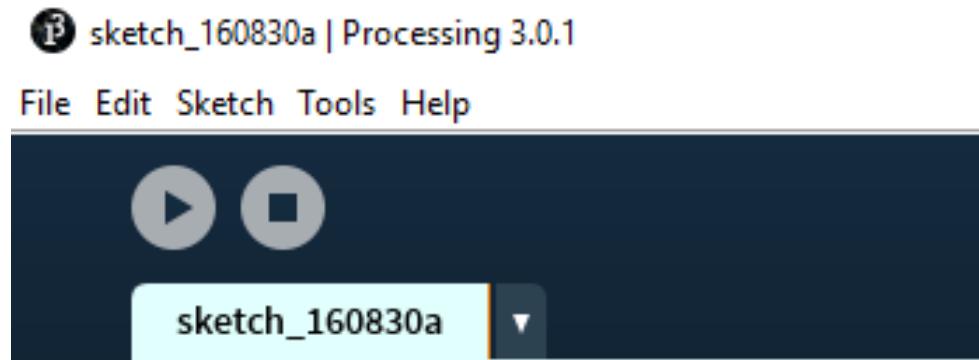


Three Short Assignments: Temperature Conversion, Computus and Buses & Taxis

- The three assignments are very similar
- Traverse a list of numbers with a for each loop
- Do a calculation
- Print and label the results

Example: Traverse a List of Numbers, Print and Label the Square of Each

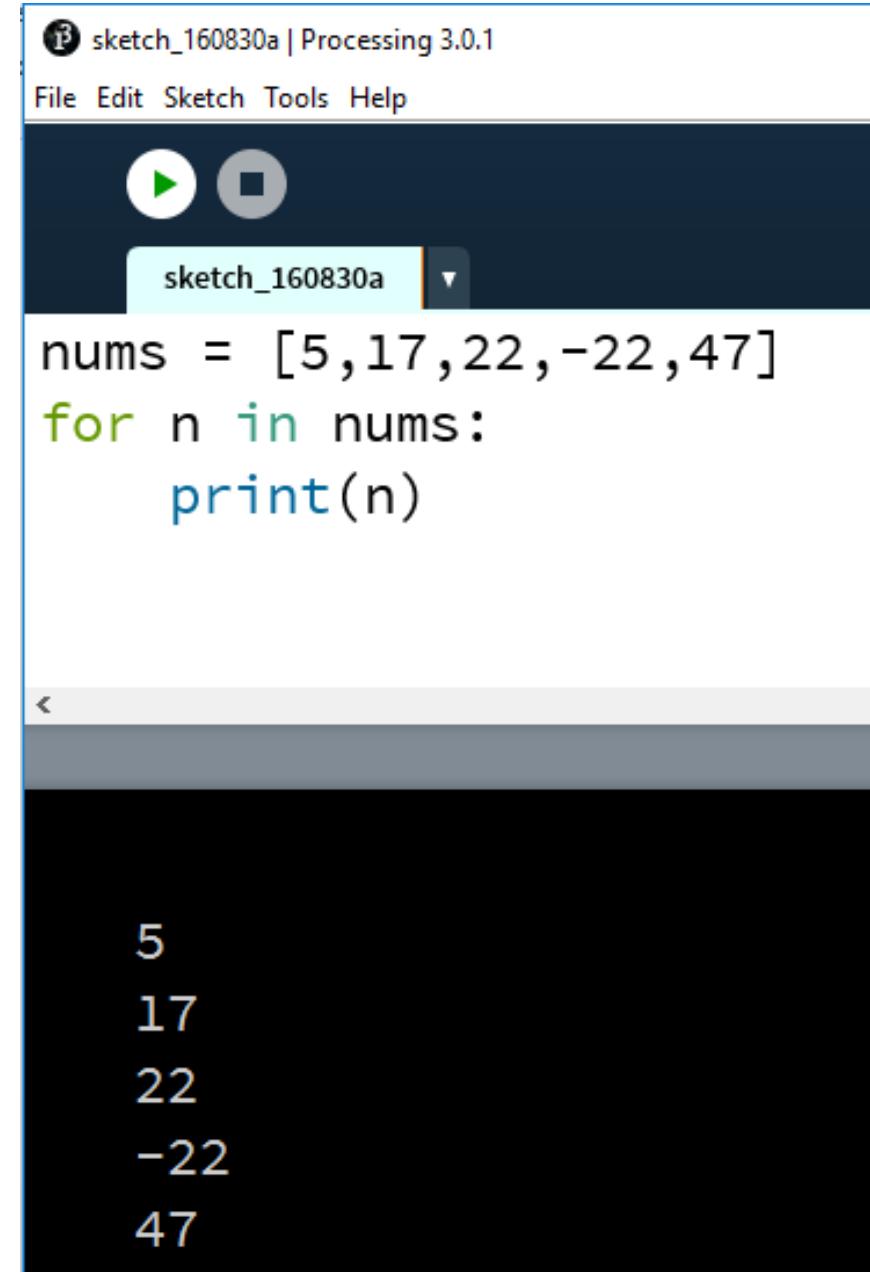
We'll start with a list of numbers.



The screenshot shows the Processing 3.0.1 software interface. At the top, there's a toolbar with icons for file operations, sketch tools, and help. Below the toolbar is a dark blue header bar with two large white circular buttons containing a play symbol and a square symbol respectively. The main workspace is dark, and at the bottom, there's a light blue status bar. On the status bar, the text "sketch_160830a" is displayed next to a dropdown arrow. Below the workspace, the code "nums = [5,17,22,-22,47]" is visible.

```
nums = [5,17,22,-22,47]
```

Then we'll
create a for each
loop to traverse
the list and print
the numbers.



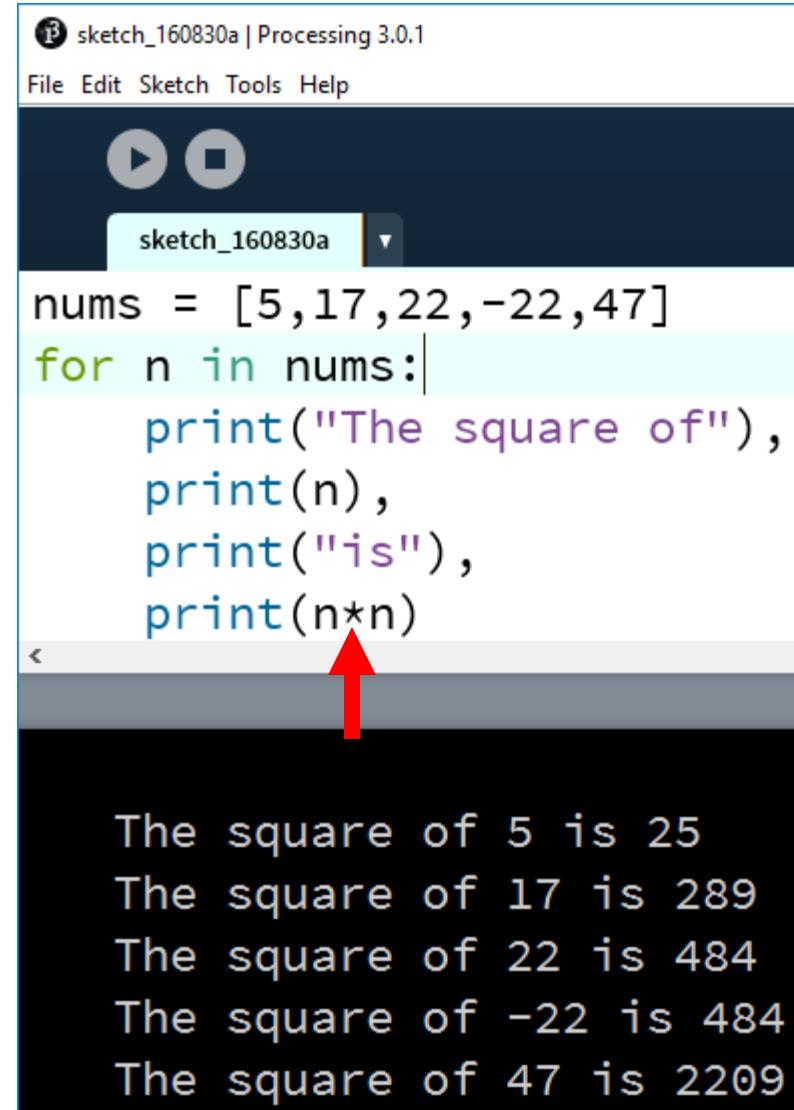
The screenshot shows the Processing 3.0.1 IDE interface. The title bar says "sketch_160830a | Processing 3.0.1". The menu bar includes File, Edit, Sketch, Tools, and Help. Below the menu is a toolbar with a play button and a stop button. The code editor contains the following Python-like pseudocode:

```
nums = [5,17,22,-22,47]
for n in nums:
    print(n)
```

The output window below the code editor displays the following text, indicating the execution of the loop:

```
5
17
22
-22
47
```

- Finally, we'll include an **expression** that performs a calculation.
- Print and label the results.

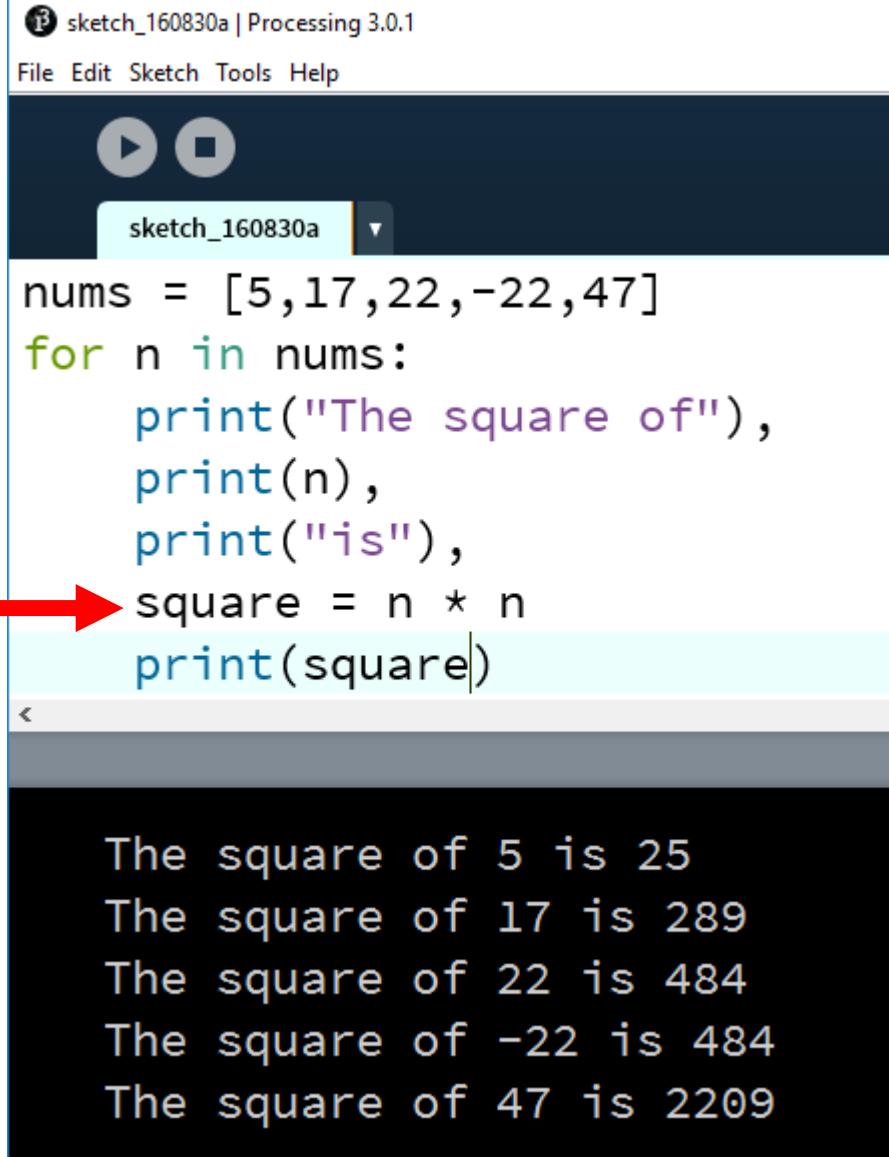


```
p sketch_160830a | Processing 3.0.1
File Edit Sketch Tools Help
sketch_160830a
nums = [5,17,22,-22,47]
for n in nums:
    print("The square of"),
    print(n),
    print("is"),
    print(n*n)
```

The square of 5 is 25
The square of 17 is 289
The square of 22 is 484
The square of -22 is 484
The square of 47 is 2209

A red arrow points to the multiplication expression `n*n` in the code editor, highlighting it as the expression being explained.

For computus and
buses and taxis
you should create
additional
variables in the
loop.



The screenshot shows the Processing 3.0.1 software interface. The title bar says "sketch_160830a | Processing 3.0.1". The menu bar includes File, Edit, Sketch, Tools, and Help. Below the menu is a toolbar with play and stop buttons. The code area contains the following pseudocode:

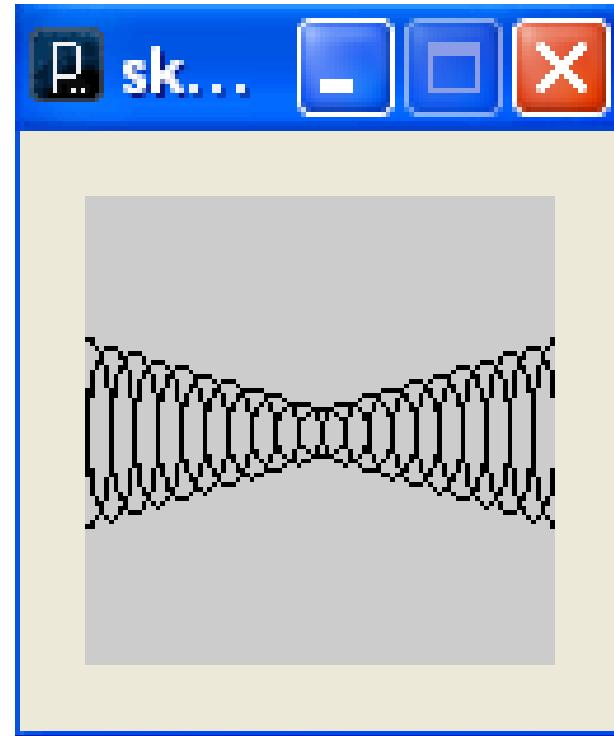
```
nums = [5,17,22,-22,47]
for n in nums:
    print("The square of"),
    print(n),
    print("is"),
    square = n * n
    print(square)
```

A red arrow points to the line "square = n * n". The output window below displays the results of the loop:

```
The square of 5 is 25
The square of 17 is 289
The square of 22 is 484
The square of -22 is 484
The square of 47 is 2209
```

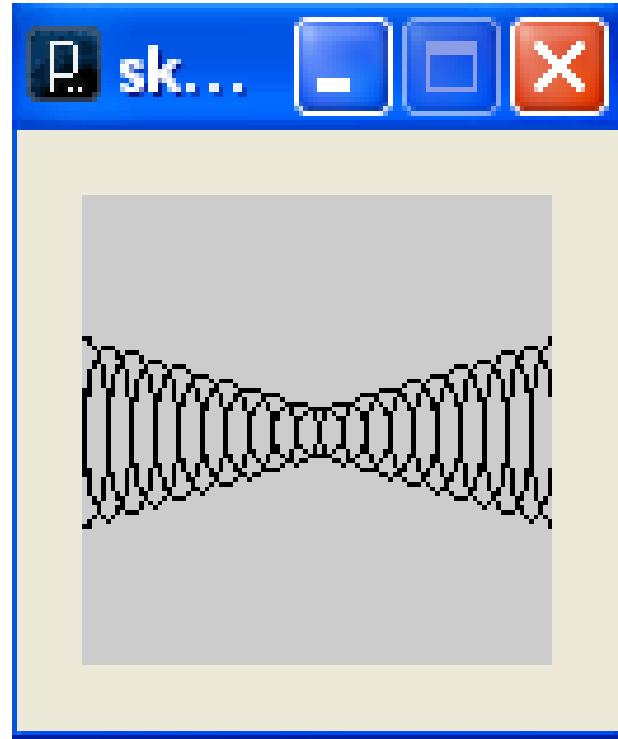
Symmetry in a Loop

```
h = 10
offset = 0
while offset < 100:
    ellipse(50 - offset,50,10,h)
    ellipse(50 + offset,50,10,h)
    h = h + 3
    offset = offset + 5
```



Notice that offset is "in charge" of the loop and h is "along for the ride"

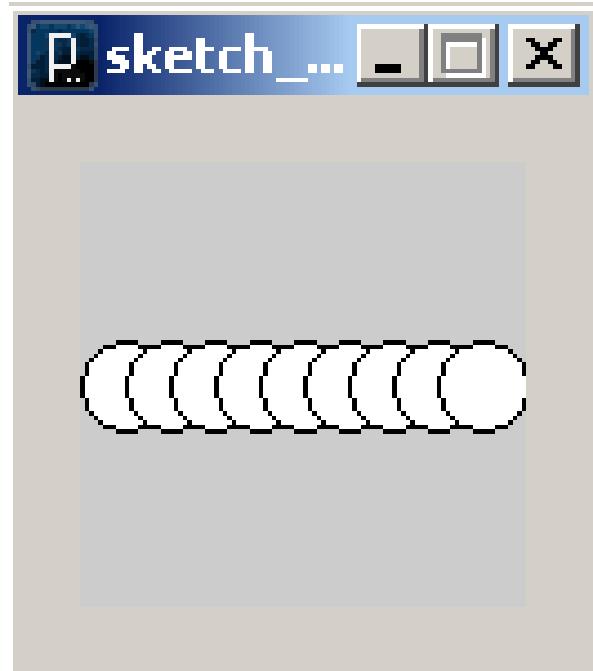
```
h = 10
offset = 0
while offset < 100:
    ellipse(50 - offset,50,10,h)
    ellipse(50 + offset,50,10,h)
    h = h + 3
    offset = offset + 5
```



Reversing a Loop

This loop positions the x coordinates at 10, 20, 30, 40, 50, 60, 70, 80 and 90.

```
x = 10
while x < 91:
    ellipse(x,50,20,20)
    x = x + 10
```



Reversing a Loop

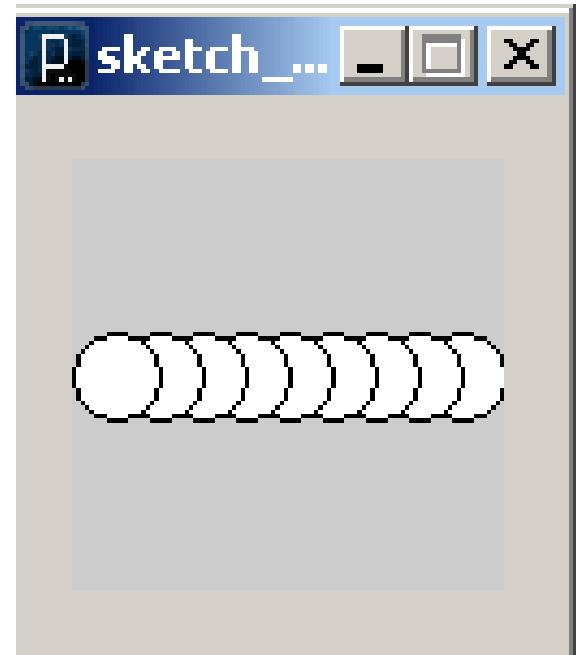
How would I make this pattern with the x coordinates at 90, 80, 70, 60, 50, 40, 30, 20 and 10?

x = ??

while x ??:

 ellipse(x, 50, 20, 20)

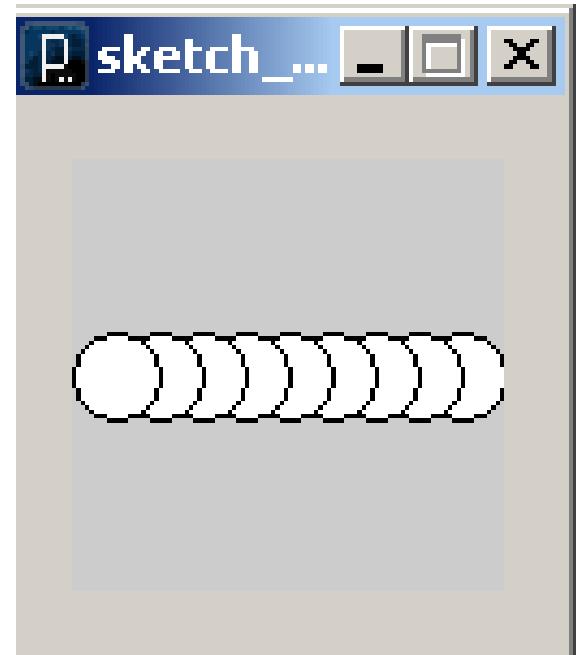
 x = x ??



Reversing a Loop

How would I make this pattern with the x coordinates at 90, 80, 70, 60, 50, 40, 30, 20 and 10?

```
x = 90;  
while x ??:  
    ellipse(x,50,20,20)  
    x = x ??
```



Reversing a Loop

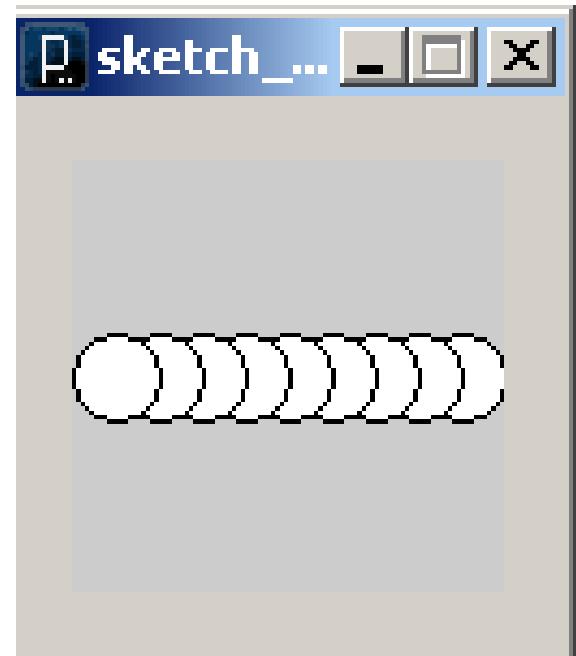
How would I make this pattern with the x coordinates at 90, 80, 70, 60, 50, 40, 30, 20 and 10?

x = 90

while x > 9:

 ellipse(x, 50, 20, 20)

 x = x ??



Reversing a Loop

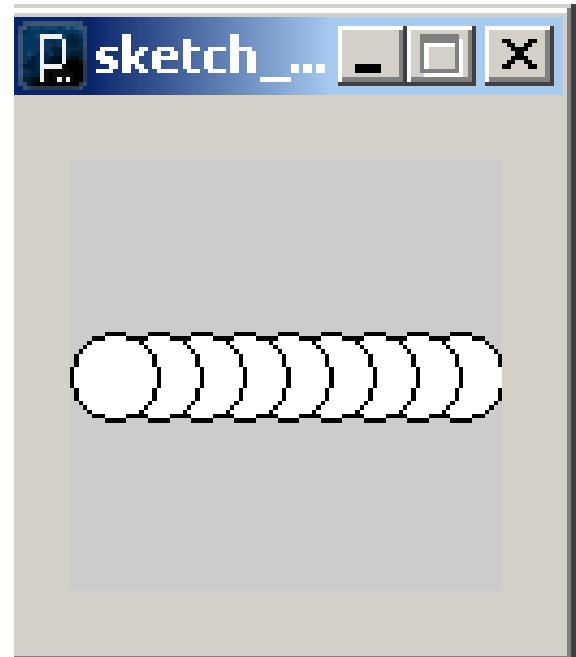
How would I make this pattern with the x coordinates at 90, 80, 70, 60, 50, 40, 30, 20 and 10?

```
x = 90
```

```
while x > 9:
```

```
    ellipse(x,50,20,20)
```

```
    x = x - 10
```



A Loop Within a Loop

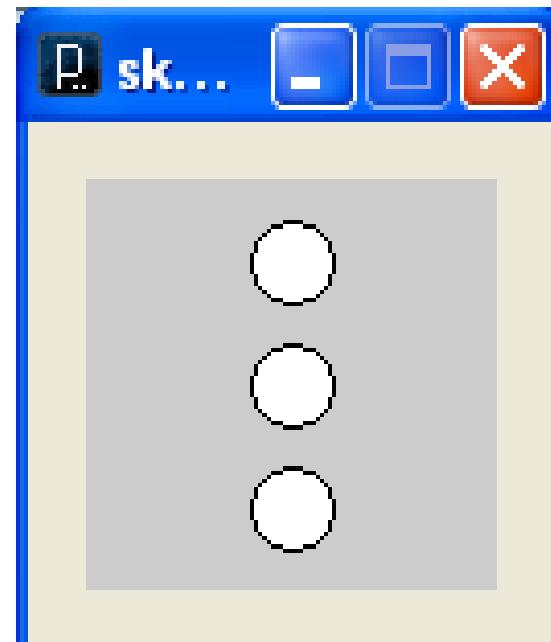
How many ellipses will this program make?

```
y = 20
while y < 81:
    ellipse(50,y,20,20)
    y = y + 30
```

A Loop Within a Loop

How many ellipses will this program make?

```
y = 20
while y < 81:
    ellipse(50,y,20,20)
    y = y + 30
```



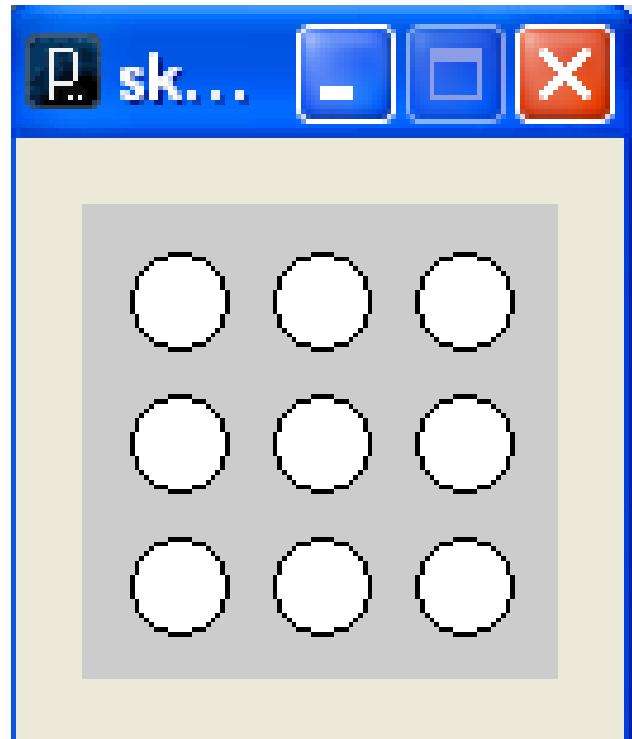
A Loop Within a Loop

```
x = 20
while x < 81:
    y = 20
    while y < 81:
        ellipse(x,y,20,20)
        y = y + 30
    x = x + 30
```

Now, what if we put that loop inside another loop ? How many ellipses would we get?

A Loop Within a Loop

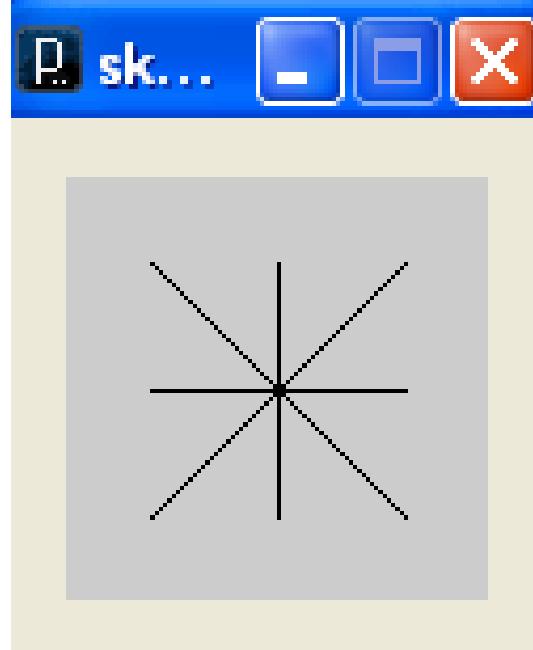
```
x = 20
while x < 81:
    y = 20
    while y < 81:
        ellipse(x,y,20,20)
        y = y + 30
    x = x + 30
```



A Loop Within a Loop

```
x = 20
while x < 81:
    y = 20
    while y < 81:
        line(x,y,50,50)
        y = y + 30
    x = x + 30
```

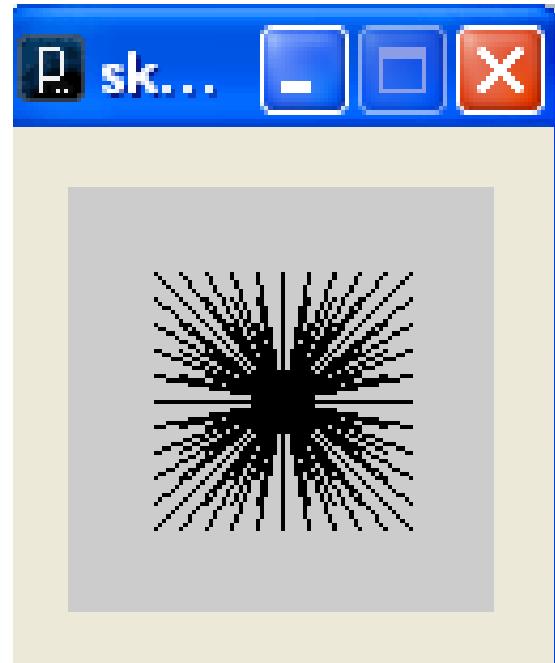
I've changed it
to make 9
lines.



A Loop Within a Loop

```
x = 20
while x < 81:
    y = 20
    while y < 81:
        line(x,y,50,50)
        y = y + 6
    x = x + 6
```

Now how many
line**s**?

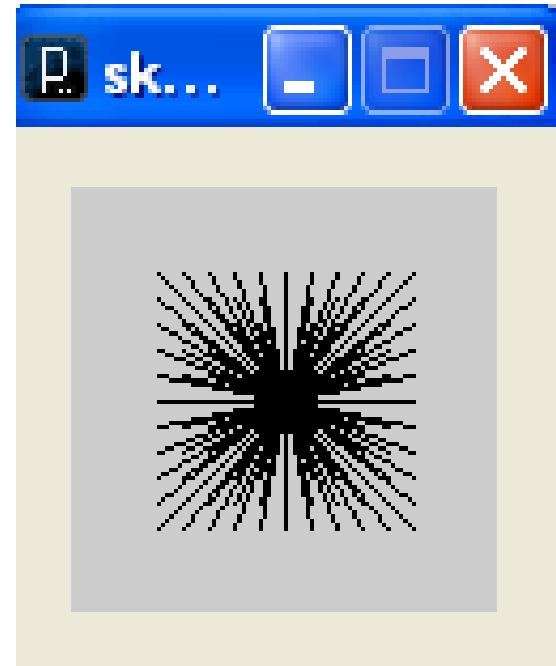


A Loop Within a Loop

```
x = 20
while x < 81:
    y = 20
    while y < 81:
        line(x,y,50,50);

        y = y + 6
        x = x + 6
```

- 121!
- A loop within a loop is called *Nested* loops

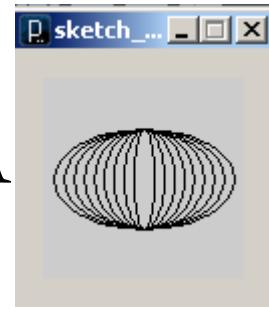


Practice Quiz Question

Find the Output

```
noFill()  
mystery = 10  
while mystery < 100:  
    ellipse(50,50,mystery,mystery)  
    mystery = mystery + 10
```

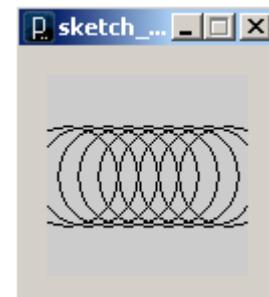
A



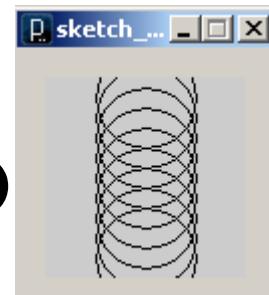
B



C



D



Selection: The **if** Statement

- For Buses and Taxis, it's possible to calculate the number of taxis with just +, / and %
- You may find it easier to use an **if**
- The **if** checks to see if a **condition** is true
- If it is, the **block of indented code** is executed once

```
if 3 < 4 :
```

```
    print("What do you know?")  
    print("3 IS less than 4!")
```

if vs. while

- The syntax of **if** and **while** is very similar
- They both check to see if a **condition** is true
- They both have a **colon**
- They both control a **block of indented code**

```
if 3 < 4:
```

```
    print("What do you know?")
```

```
x = 1
```

```
while x < 3:
```

```
    print(x)
```

```
    x = x + 1
```

if vs. while

- The BIG difference is that if the **condition** is true, the **if** will execute the **block** of indented code exactly **once**
- The **while** will execute the **block** of indented code **repeatedly** while **condition** is true
- ```
if 3 < 4:
 print("What do you know?")

x = 1
while x < 3:
 print(x)
 x = x + 1
```

# The if Statement

- Let's say **taxiPeople** holds the number of people left after the buses are loaded
- If **taxiPeople** is not a multiple of 5 we need an extra taxi

```
1 people will need 0 buses and 1 taxis
49 people will need 0 buses and 10 taxis
50 people will need 1 buses and 0 taxis
51 people will need 1 buses and 1 taxis
56 people will need 1 buses and 2 taxis
199 people will need 3 buses and 10 taxis
200 people will need 4 buses and 0 taxis
201 people will need 4 buses and 1 taxis
202 people will need 4 buses and 1 taxis
203 people will need 4 buses and 1 taxis
204 people will need 4 buses and 1 taxis
205 people will need 4 buses and 1 taxis
206 people will need 4 buses and 2 taxis
```

# The if Statement

We might write code like:

```
other code not shown
taxis = taxiPeople/5
if taxiPeople % 5 > 0:
 taxis = taxis + 1
```

# The if Statement

If the remainder of `taxiPeople` divided by 5 is greater than 0, then `taxiPeople` is not a multiple of 5 and we need an extra taxi.

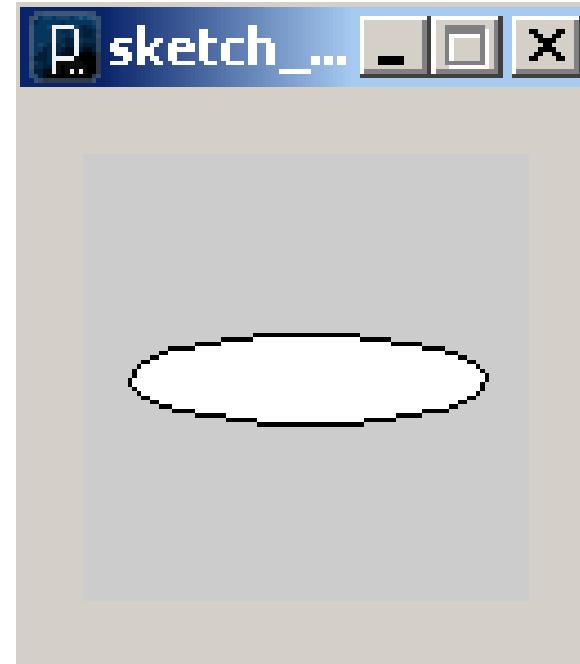
```
other code not shown
taxis = taxiPeople/5
if taxiPeople % 5 > 0:
 taxis = taxis + 1
```

# Rotations

- You can use `rotate()` and `translate()` to rotate shapes, but it's very confusing
- You will NOT be tested on `rotate()` and `translate()`

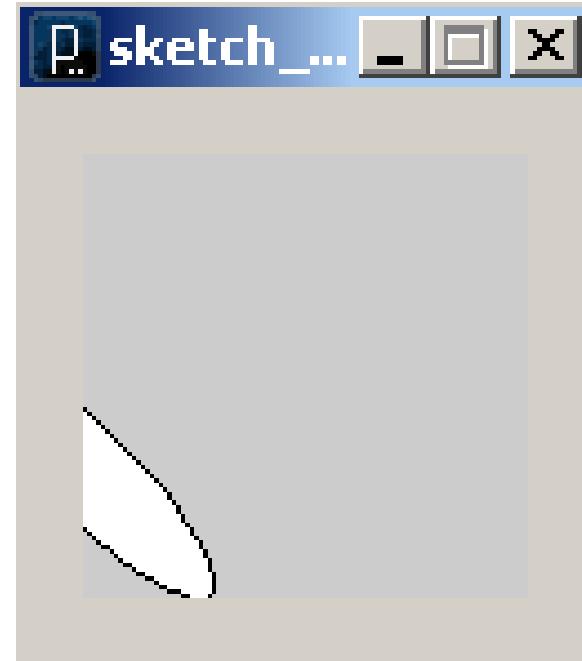
# Rotations

```
ellipse(50,50,80,20)
```



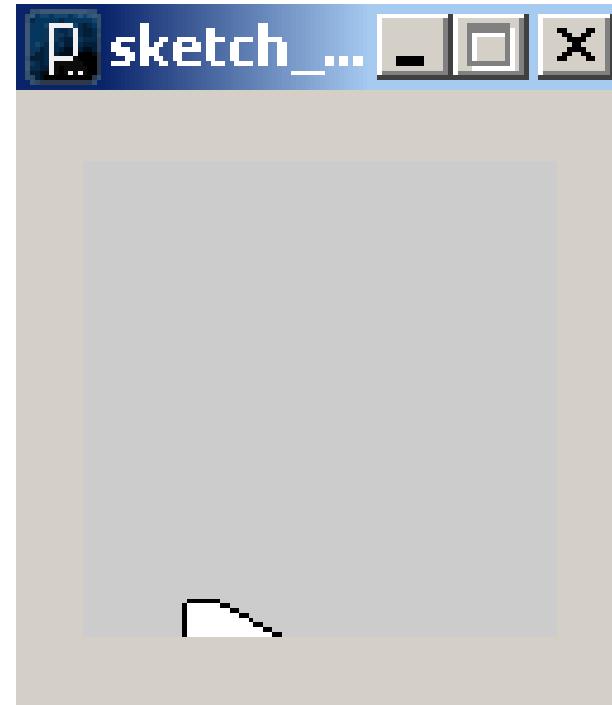
# Rotations

```
rotate(PI/4)
ellipse(50,50,80,20)
```



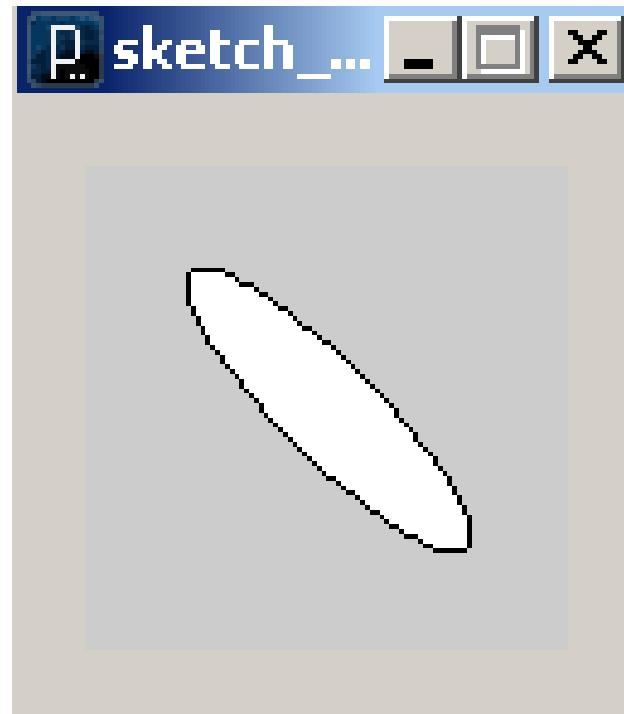
# Rotations

```
translate(50,50)
rotate(PI/4)
ellipse(50,50,80,20)
```



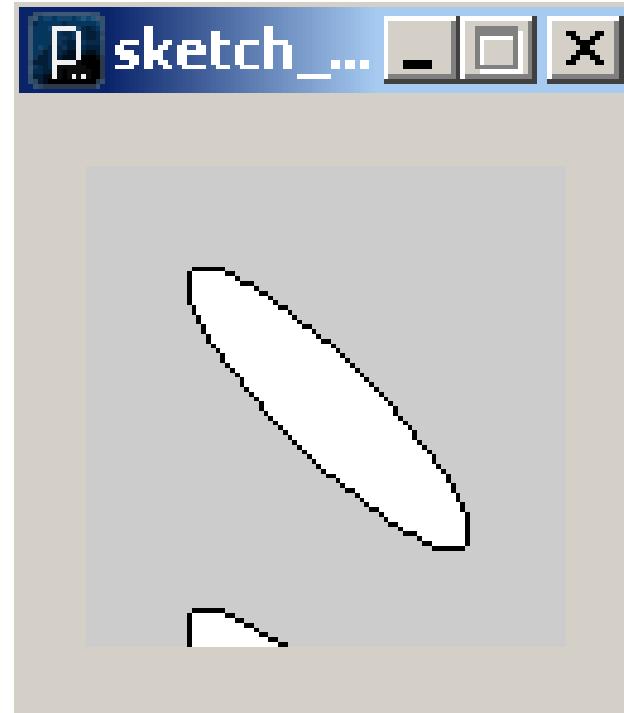
# Rotations

```
translate(50,50)
rotate(PI/4)
ellipse(0,0,80,20)
```



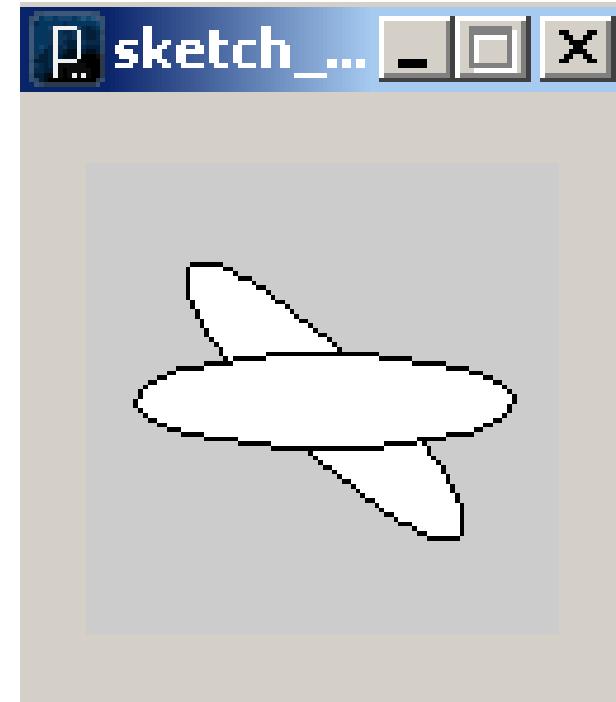
# Rotations

```
translate(50,50)
rotate(PI/4)
ellipse(0,0,80,20)
ellipse(50,50,80,20)
```

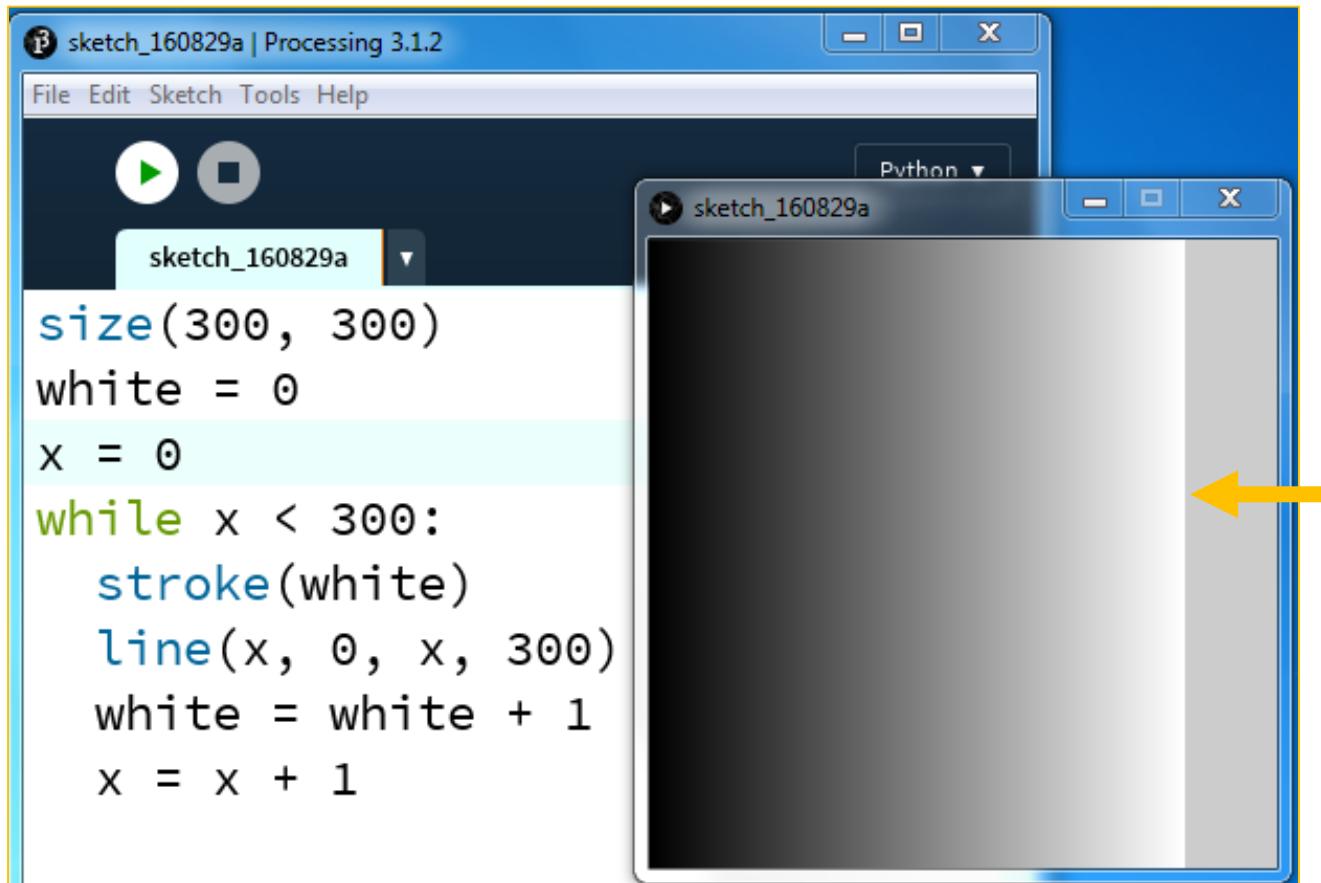


# Undo in Reverse Order

```
translate(50,50)
rotate(PI/4)
ellipse(0,0,80,20)
rotate(-PI/4)
translate(-50,-50)
ellipse(50,50,80,20)
```



# Oops! Watch out for this mistake. Colors must stay between 0 & 255.



# A “Scaled” Gradient

The screenshot shows the Processing 3.1.2 environment. The title bar says "sketch\_160830b | Processing 3.1.2". The code window contains the following pseudocode:

```
size(300,200)
white = 0
x = 0
while x < 300:
 stroke(white)
 line(x,0,x,200)
 white = white + 255/300.0
 x = x + 1
```

Two red arrows point to the line `stroke(white)` and the line `white = white + 255/300.0`. The preview window below shows a vertical gradient line from black at the top to white at the bottom.

# Another “Scaled” Gradient

The screenshot shows the Processing IDE interface. At the top, it says "sketch\_160830b | Processing 3.1.2". Below the menu bar, there are two red arrows pointing down to specific lines of code. The first arrow points to the line "white = 0", and the second arrow points to the line "white = white + 255/450.0". The code itself is as follows:

```
size(450,100)
white = 0
x = 0
while x < 450:
 stroke(white)
 line(x,0,x,100)
 white = white + 255/450.0
 x = x + 1
```

The bottom window shows the resulting grayscale gradient line.

**Oops, what  
happened  
here?  
Why didn't  
the gradient  
scale to  
white?**

sketch\_160830b | Processing 3.1.2

File Edit Sketch Tools Help

Python ▾

```
sketch_160830b
size(300,200)
white = 0
x = 0
while x < 300:
 stroke(white)
 line(x,0,x,200)
 white = white + 255/300
 x = x + 1
```

Console

sketch\_160830b

255/300 is  
zero so the  
stroke  
never  
lightens.

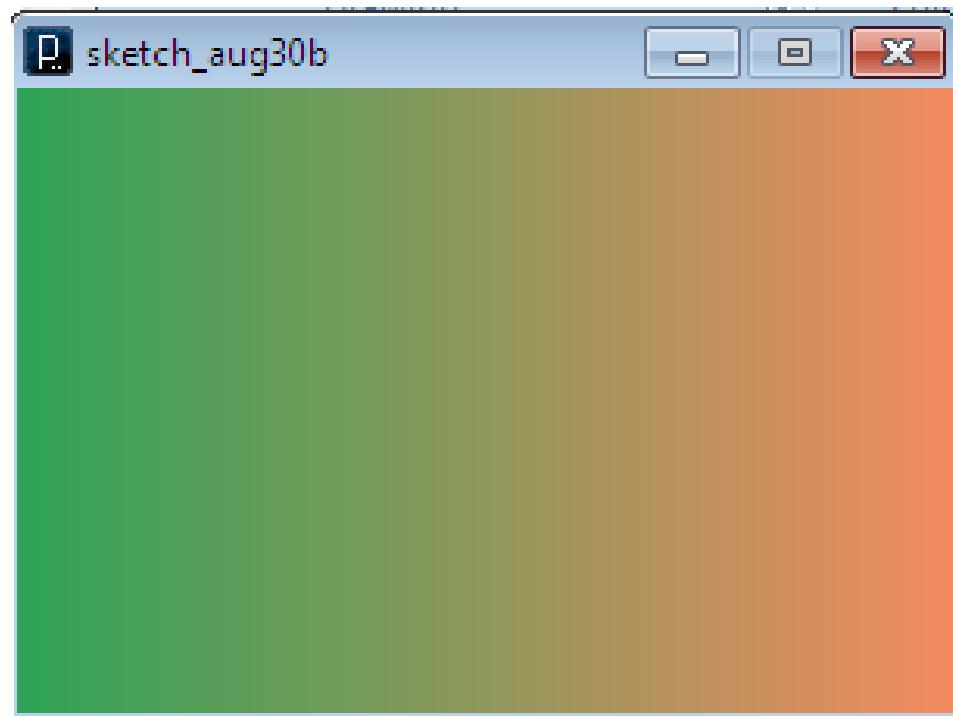
```
sketch_160830b | Processing 3.1.2
File Edit Sketch Tools Help
Python ▾
sketch_160830b
size(300,200)
white = 0
x = 0
while x < 300:
 stroke(white)
 line(x,0,x,200)
 white = white + 255/300
 x = x + 1
```

Console

A program that will make a gradient from a **first color** to a **second color** for a particular **width**

Example: Sea Green **46-164-87** to Sandy Brown **244-139-96**  
(yuk)

```
size(300,200)
r = 46
g = 164
b = 87
x = 0
while x < 300:
 stroke(r,g,b)
 line(x,0,x,200)
 x = x + 1
 r = r + (244-46)/300.0
 g = g + (139-164)/300.0
 b = b + (96-87)/300.0
```



# Python vs. AP Exam

## AP Exam

```
i <- 0
REPEAT UNTIL (i = 4)
{
 i <- i + 1
 DISPLAY (sum)
}
```

## Python

```
i = 0
while i < 4:
 i = i + 1
 print(i)
```

# Variable Assignment

AP Exam

```
i <- 0
REPEAT UNTIL (i =
4)
{
 i <- i + 1
 DISPLAY (sum)
}
```

Python

```
i = 0
while i < 4:
 i = i + 1
 print(i)
```

# REPEAT UNTIL condition is true while condition is true

AP Exam

```
i <- 0
REPEAT UNTIL (i =
4)
{
 i <- i + 1
 DISPLAY(i)
}
```

Python

```
i = 0
while i < 4:
 i = i + 1
 print(i)
```

# Both display 1 2 3 4

AP Exam

```
i <- 0
REPEAT UNTIL (i = i = 0
4)
{
 i <- i + 1
 DISPLAY(i)
}
```

Python

```
while i < 4:
 i = i + 1
 print(i)
```

# Practice Quiz Question:

## What will this algorithm display?

```
i <- 1
sum <- 0
REPEAT UNTIL (i = 3)
{
 sum <- sum + i
 i <- i + 1
}
DISPLAY(sum)
```

- A. 0
- B. 1
- 2
- C. 3
- D. 6
- E. nothing

Equivalent Python Code:

```
i = 1
sum = 0
while i < 3:
 sum = sum + i
 i = i + 1
print(sum)
```

# Write *Computus* Step by Step

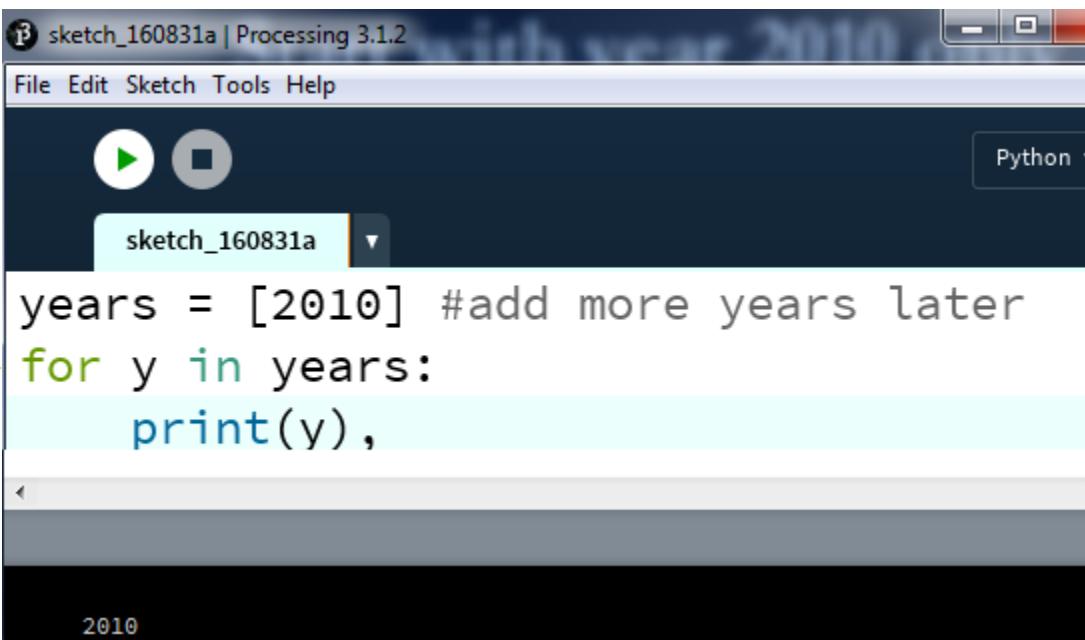
of the date of Easter is called *Computus* in Latin. One algorithm for *Computus* was invented by mathematician Carl Friedrich Gauss in the early 1800s and is shown below. All divisions are integer divisions.

1. Let  $y$  be the year.
2. Divide  $y$  by 19 and call the remainder  $a$ . Ignore the quotient.
3. Divide  $y$  by 100 to get a quotient  $b$  and a remainder  $c$ .
4. Divide  $b$  by 4 to get a quotient  $d$  and a remainder  $e$ .
5. Divide  $8 * b + 13$  by 25 to get a quotient  $g$ . Ignore the remainder.
6. Divide  $19 * a + b - d - g + 15$  by 30 to get a remainder  $h$ . Ignore the quotient.
7. Divide  $c$  by 4 to get a quotient  $j$  and a remainder  $k$ .
8. Divide  $a + 11 * h$  by 319 to get a quotient  $m$ . Ignore the remainder.
9. Divide  $2 * e + 2 * j - k - h + m + 32$  by 7 to get a remainder  $r$ . Ignore the quotient.
10. Divide  $h - m + r + 90$  by 25 to get a quotient  $n$ . Ignore the remainder.
11. Divide  $h - m + r + n + 19$  by 32 to get a remainder  $p$ . Ignore the quotient.

- The *Computus* Algorithm is complicated.
- Check each step before going to the next.
- Use the known values for 2010 to check your progress.

Check your calculations. For 2010  $a, b, c, d, e, g, h, j, k, m, r, n, p$  should be 15, 20, 10, 5, 0, 6, 9, 2, 2, 0, 4, 4, 4

# Start with Year 2010 Only

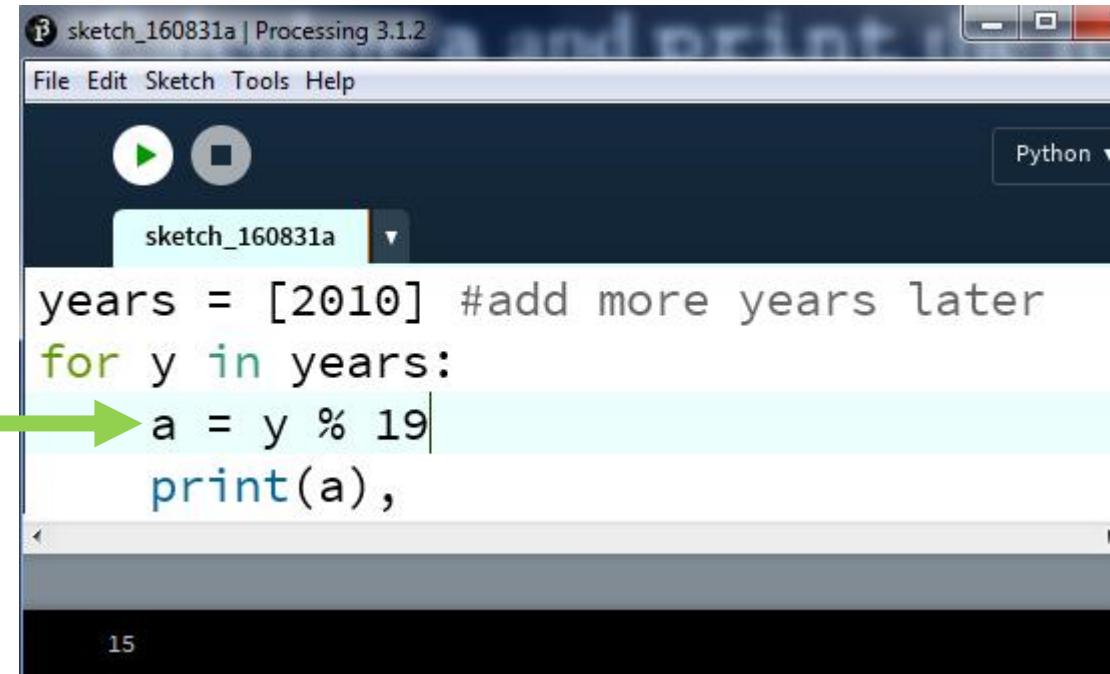


```
sketch_160831a | Processing 3.1.2
File Edit Sketch Tools Help
Python ▾
sketch_160831a
years = [2010] #add more years later
for y in years:
 print(y),
2010
```

A yellow arrow points to the first line of code, `years = [2010]`.

- Let's get the program working for 2010 first
- We can add more years to the list later
- Our first instruction says “**Let y be the year**”
- We should store 2010 in **y** and **print()** it to test it

# Calculate a and print the Result



```
sketch_160831a | Processing 3.1.2
File Edit Sketch Tools Help
Python ▾
sketch_160831a
years = [2010] #add more years later
for y in years:
 a = y % 19
 print(a),
15
```

- Since we already checked **y**, we no longer need to **print** it.
- Our second instruction say “Divide **y** by 19 and call the **remainder a**. Ignore the quotient”
- That means we are only interested in the remainder
- For 2010 **a** should be 15 .

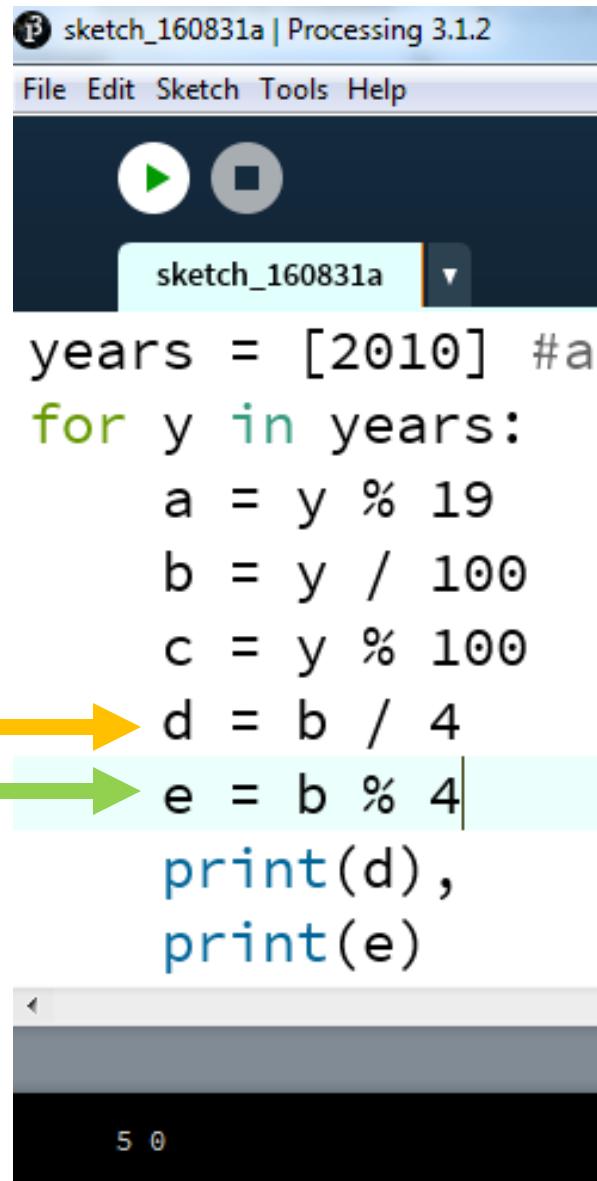
# Calculate b and c and print the Result

```
sketch_160831a | Processing 3.1.2
File Edit Sketch Tools Help
Python ▾
sketch_160831a
years = [2010] #add more years later
for y in years:
 a = y % 19
 → b = y / 100
 → c = y % 100
 print(b),
 print(c)
20 10
```

- Our next instruction says “Divide **y** by 100 to get a quotient **b** and a **remainder c**.” For 2010 that’s **20 & 10**.
- Since we already checked **a**, we no longer need to **print** it.

# Calculate d and e and print the Result

- Our next instruction says “Divide b by 4 to get a **quotient d** and a **remainder e**.”
- Since we already checked **b** and **c** we no longer need to **print** them
- For 2010, **d** and **e** should be 5 and 0
- Repeat the process for the remaining variables



The screenshot shows the Processing 3.1.2 software interface. The title bar reads "sketch\_160831a | Processing 3.1.2". The menu bar includes File, Edit, Sketch, Tools, and Help. Below the menu is a toolbar with a play button and a stop button. The sketch window contains the following code:

```
years = [2010] #array
for y in years:
 a = y % 19
 b = y / 100
 c = y % 100
 d = b / 4
 e = b % 4
 print(d),
 print(e)
```

A yellow arrow points to the line `d = b / 4`, and a green arrow points to the line `e = b % 4`. The output window at the bottom shows the result: `5 0`.

# Practice Quiz Question:

## What Will this Algorithm Display?

```
i <- 1
REPEAT UNTIL (i = 5)
{
 DISPLAY(i)
 i <- i + 2
}
```

- A. 5
- B. 3
- C. 3
- 5
- D. 1
- 3
- E. 1
- 3
- 5

Equivalent Python Code:

```
i = 1
while i < 5:
 print(i)
 i = i + 2
```

# Infinite Loops

- An *infinite loop* is a mistake.
- It's a loop that never reaches it's stopping point.
- This program repeatedly prints 1 because there is no progression or step.
- The **condition** **x < 10** is always true.

P sketch\_160902a | Processing 3.0.1

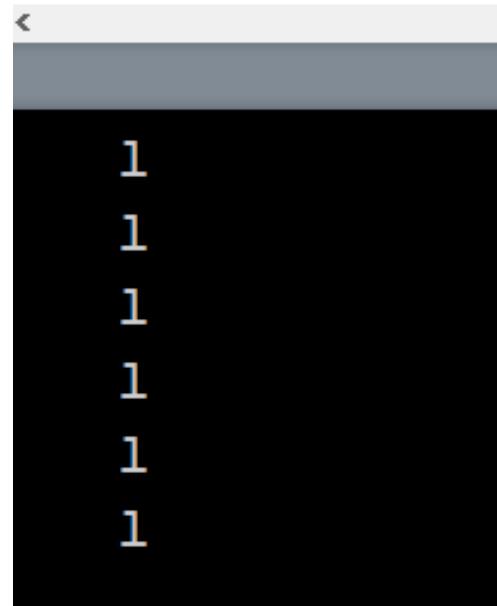
File Edit Sketch Tools Help



```
x = 1
```

```
while x < 10:
```

```
 print(x)
```



# Modern programs are very big— they are organized into *Functions*

| Year | Operating System     | Lines of Code |
|------|----------------------|---------------|
| 1993 | Windows NT 3.1       | 6,000,000     |
| 1996 | Windows NT 4.0       | 16,000,000    |
| 2000 | Windows 2000         | 29,000,000    |
| 2001 | Windows XP           | 40,000,000    |
| 2005 | Windows Vista Beta 2 | 50,000,000    |
| 2005 | Mac OS X 10.4        | 86,000,000    |
| 2005 | Red Hat Linux 7.1    | 30,000,000    |
| 2005 | Debian GNU/Linux     | 213,000,000   |

- Programs can be difficult to understand.
- *Windows Vista has over 50,000,000 lines of computer code!*
- One way to make programs easier to understand is to break them down into smaller "chunks" or modules.
- One name for these modules is *functions*.

# You Wouldn't Write a Paper that was Just One Long Paragraph

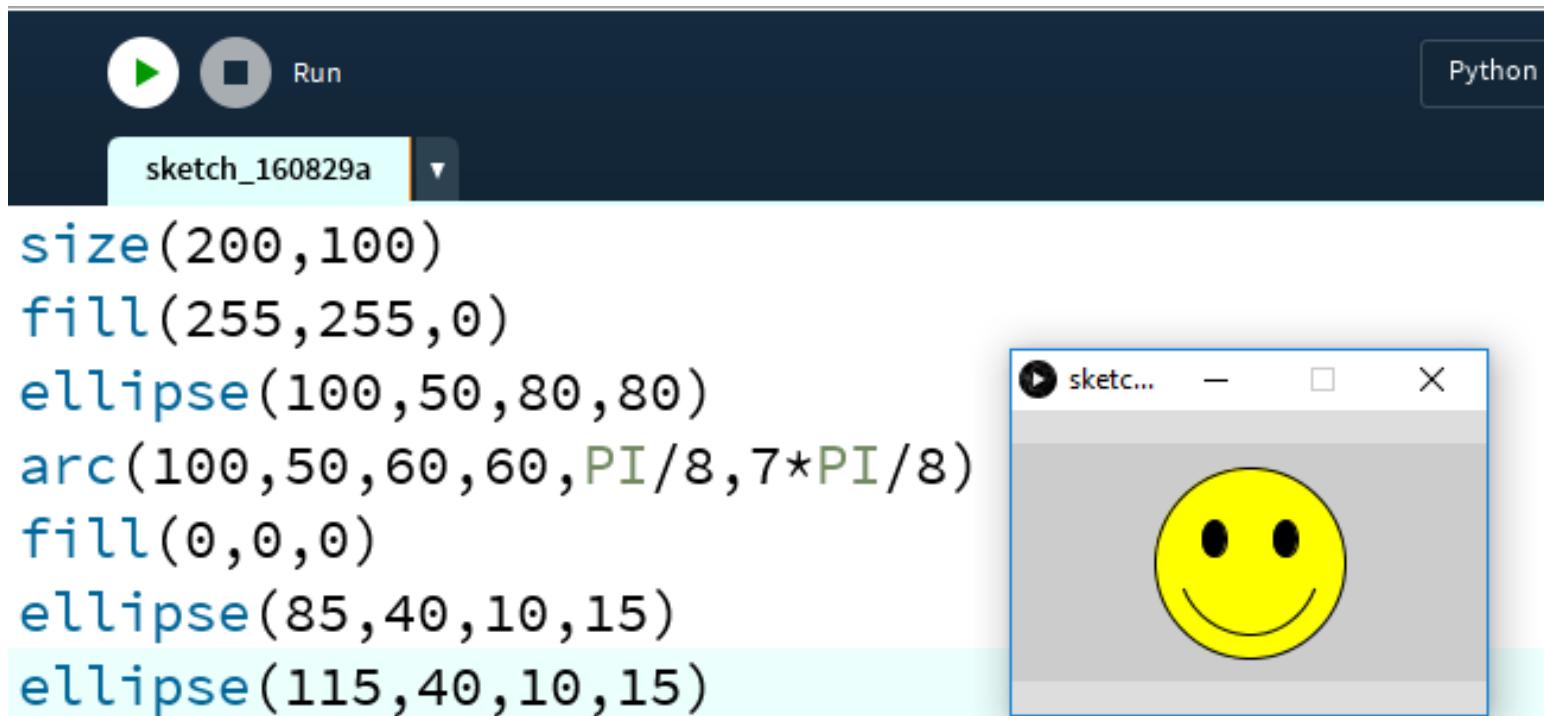
- You'd have many paragraphs, each of which would focus on one topic.
- It's the same in programs—we divide our program into chunks called **functions**.
- Each function focuses on one job or task.

# **setup()** and **draw()**

- Processing programs that use functions must contain two special functions called **setup()** and **draw()** .
- Put things that happen only once at the beginning in **setup()** .
- Put code that draws in **draw()** .
- You may create as many additional functions as you want.

# Happy Face Example

Let's say I had a program that draws a happy face.



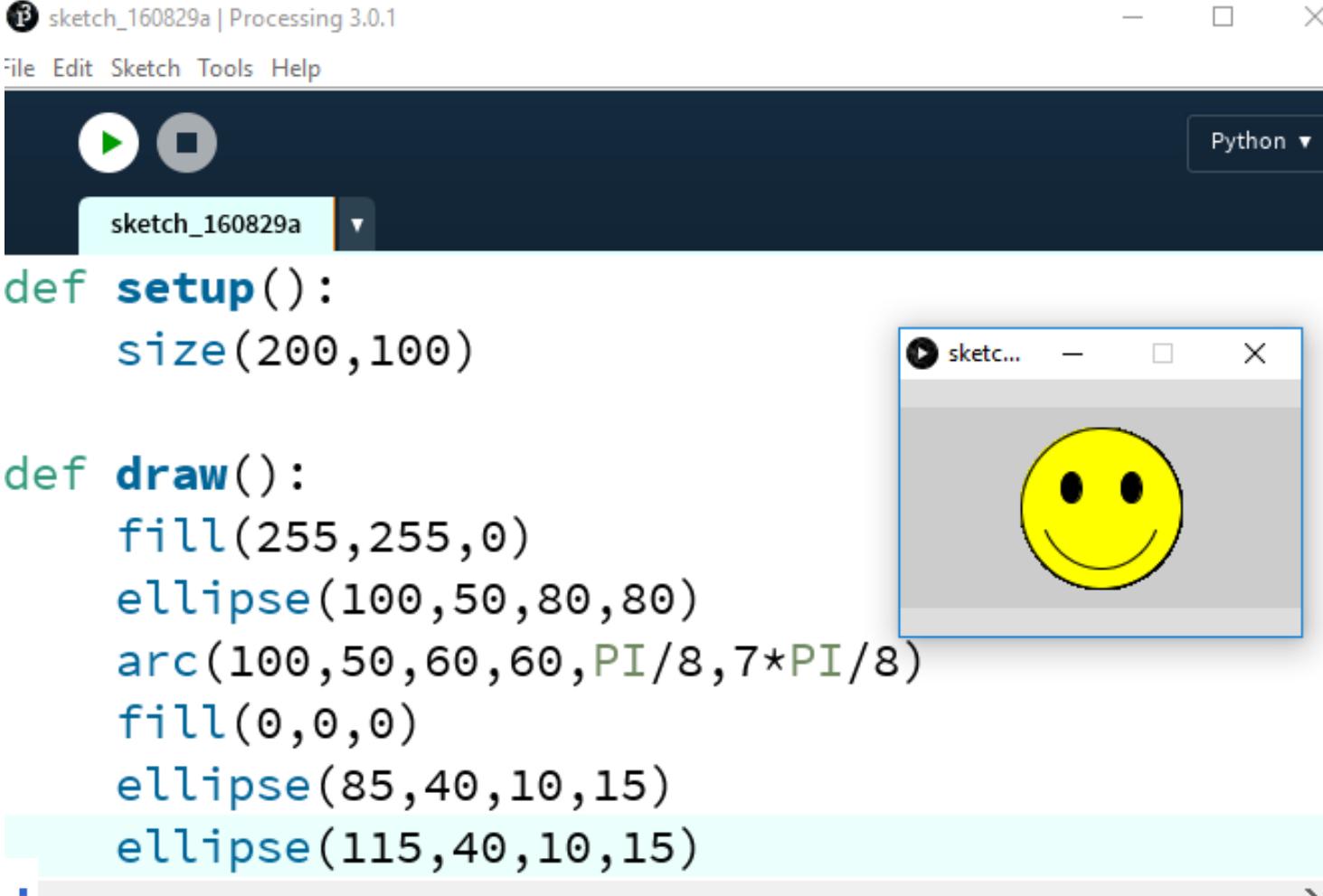
The image shows a screenshot of a code editor interface. At the top, there are buttons for 'Run' and 'Python'. Below the buttons, the file name 'sketch\_160829a' is displayed. The main area contains the following code:

```
size(200,100)
fill(255,255,0)
ellipse(100,50,80,80)
arc(100,50,60,60,PI/8,7*PI/8)
fill(0,0,0)
ellipse(85,40,10,15)
ellipse(115,40,10,15)
```

To the right of the code editor, a window titled 'sketc...' shows a yellow smiley face with black eyes and a curved smile, centered on a gray background.

# Happy Face Example

Here's what it would look like separated into `setup()` and `draw()` functions.



The screenshot shows the Processing 3.0.1 interface with a sketch titled "sketch\_160829a". The code defines a setup function that sets the size to 200x100 pixels. The draw function fills the background with white, draws a yellow ellipse at (100, 50) with a diameter of 80 pixels, and draws a curved arc from (100, 50) to (60, 60) with a radius of 60 pixels, spanning an angle from PI/8 to 7\*PI/8. It then fills the eyes with black and draws two black ellipses at (85, 40) and (115, 40) with a diameter of 15 pixels each. To the right, a preview window titled "sketc..." shows a yellow smiley face with black eyes and a curved smile.

```
sketch_160829a | Processing 3.0.1
File Edit Sketch Tools Help
Python ▾

def setup():
 size(200,100)

def draw():
 fill(255,255,0)
 ellipse(100,50,80,80)
 arc(100,50,60,60,PI/8,7*PI/8)
 fill(0,0,0)
 ellipse(85,40,10,15)
 ellipse(115,40,10,15)
```

# The Syntax of Functions

```
def setup():
 size(200,100)
```

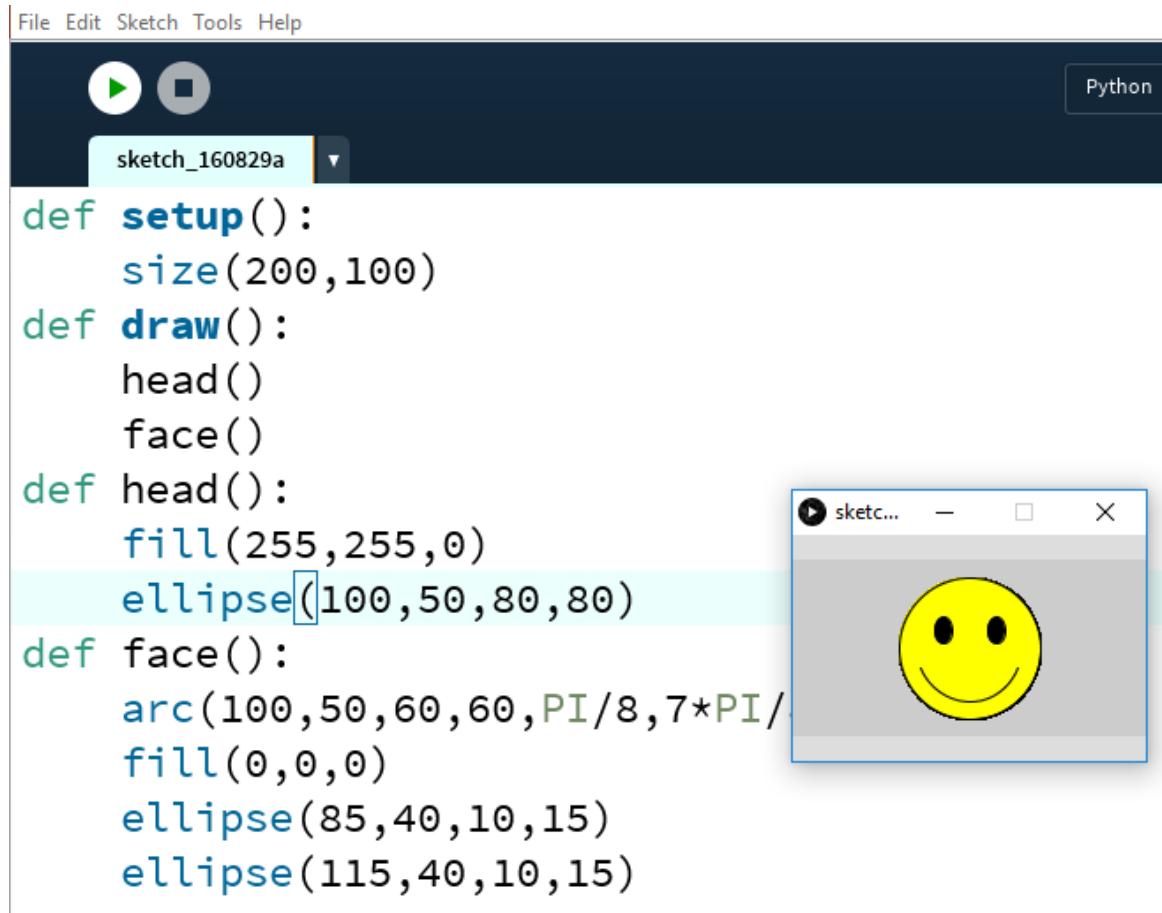
- **def** stands for “definition”
- Then a **name** followed by **parenthesis**
- Then a **colon**
- One or more lines of **indented code** that “belongs” to the function

# setup() and draw()

- **setup()** runs first and gets the drawing canvas ready, then **draw()** runs repeatedly
- In **setup()** you might:
  - Set the **size()** of the drawing canvas
  - Set the **background()** color
  - Set the **stroke()** and **fill()** colors (if they are the same for all shapes)
- In **draw()** you might:
  - Set the **background()** color
  - Set the **stroke()** and **fill()** colors (if they are different for different shapes)
  - Draw shapes like **ellipse()**, **line()**, **rect()**, etc.

# Happy Face Example

Here I've added my own **face()** and **head()** functions.

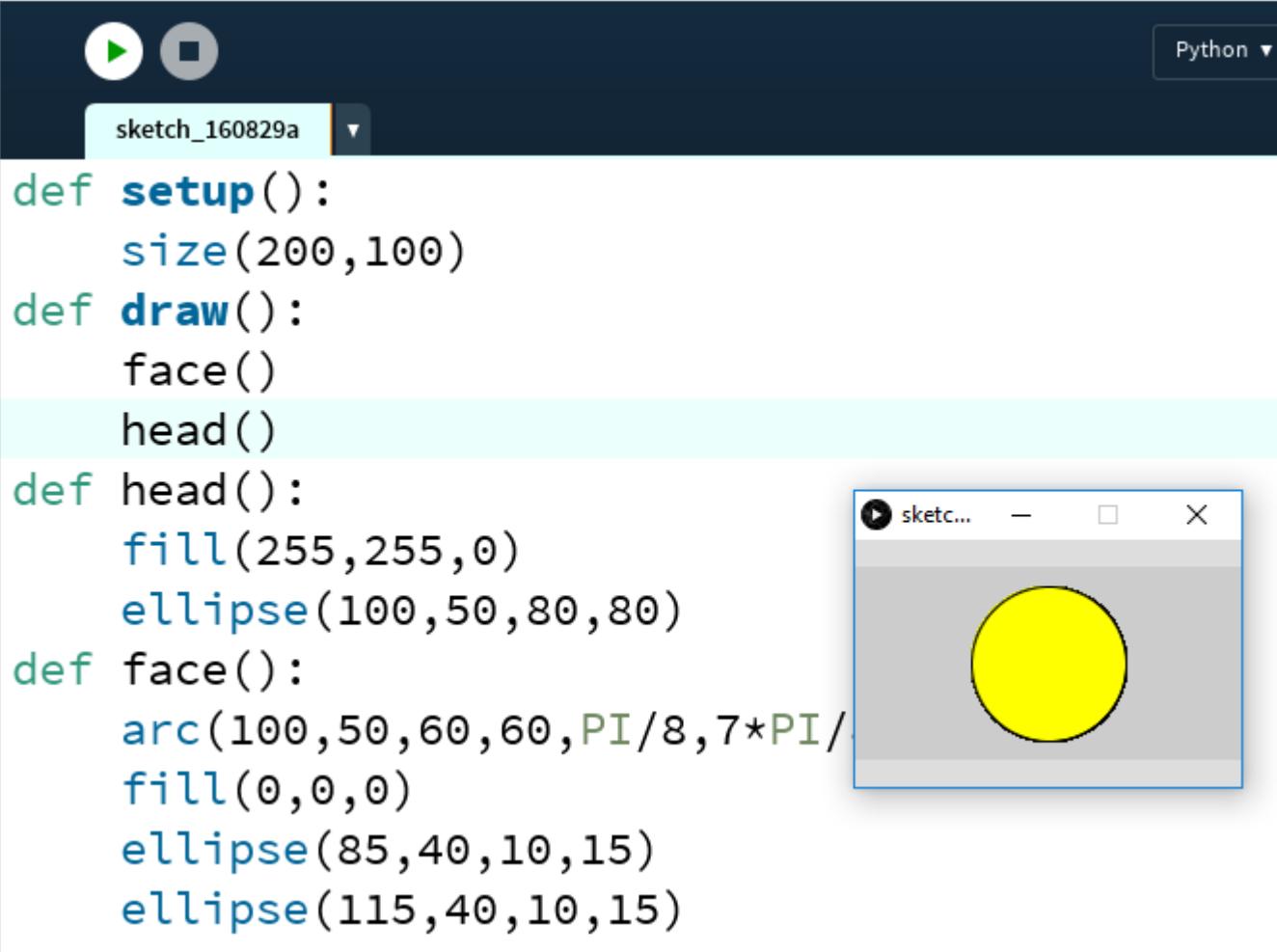


The image shows the Processing IDE interface. At the top, there's a menu bar with File, Edit, Sketch, Tools, and Help. Below the menu is a toolbar with a play button, a square button, and a Python dropdown menu set to "Python". The main area is titled "sketch\_160829a". The code is written in Python:

```
def setup():
 size(200,100)
def draw():
 head()
 face()
def head():
 fill(255,255,0)
 ellipse(100,50,80,80)
def face():
 arc(100,50,60,60,PI/8,7*PI/
 fill(0,0,0)
 ellipse(85,40,10,15)
 ellipse(115,40,10,15)
```

To the right of the code editor, a preview window titled "sketch\_160829a" displays a yellow smiley face with black eyes and a curved smile.

# Oops! What happened?

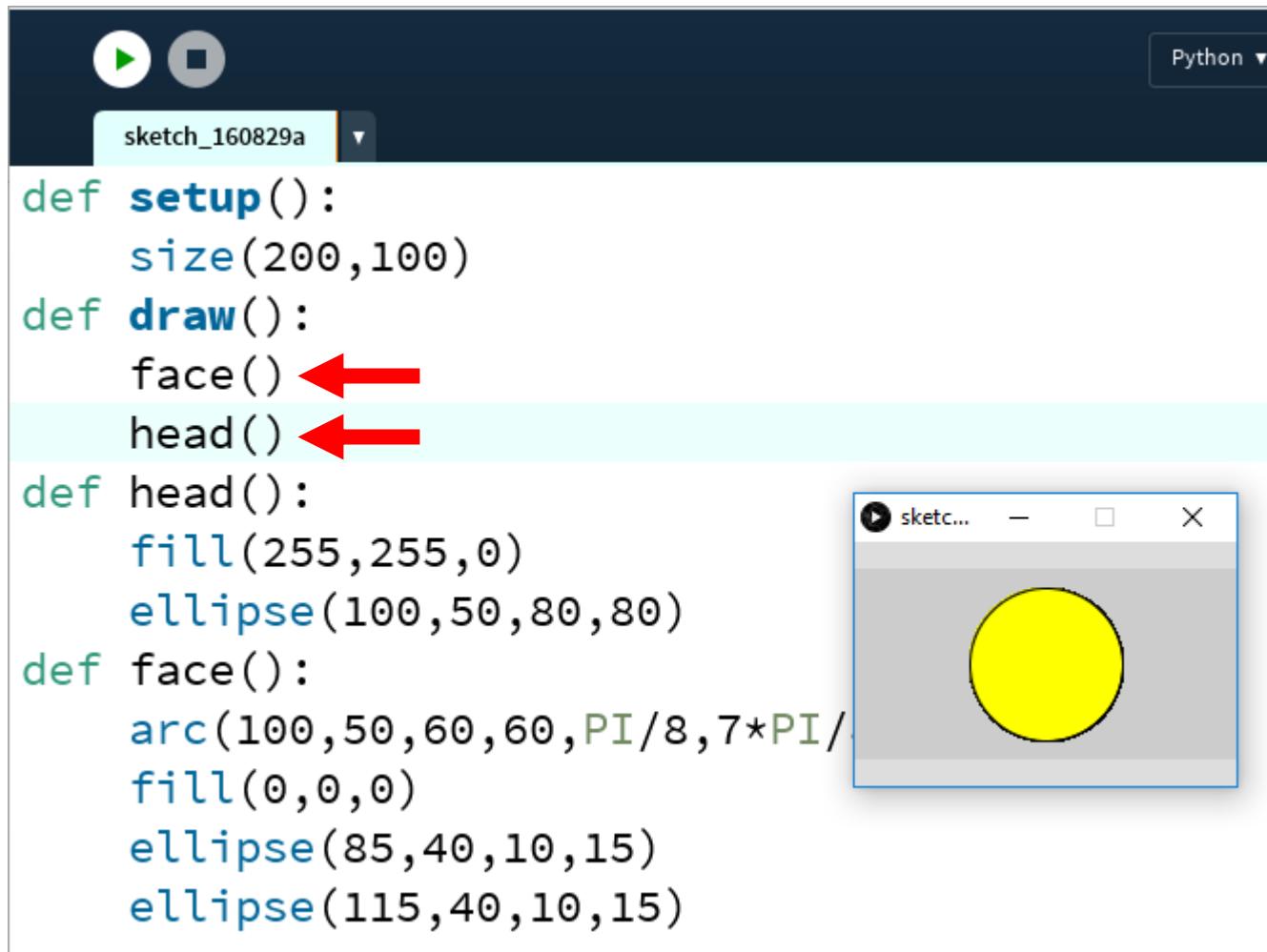


The screenshot shows the Processing IDE interface. At the top, there are play and stop buttons, a dropdown menu labeled "sketch\_160829a", and a "Python" dropdown. The main code area contains the following:

```
def setup():
 size(200,100)
def draw():
 face()
 head()
def head():
 fill(255,255,0)
 ellipse(100,50,80,80)
def face():
 arc(100,50,60,60,PI/8,7*PI/
 fill(0,0,0)
 ellipse(85,40,10,15)
 ellipse(115,40,10,15)
```

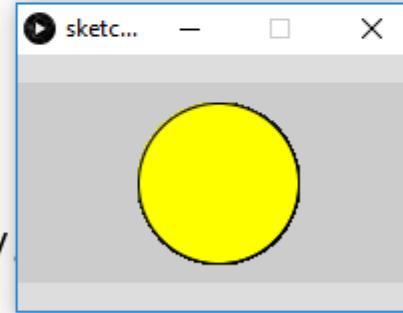
To the right of the code, a preview window titled "sketch\_160829a" displays a single yellow circle centered at (100, 50) with a diameter of 80 pixels.

# The Order of the Function Calls

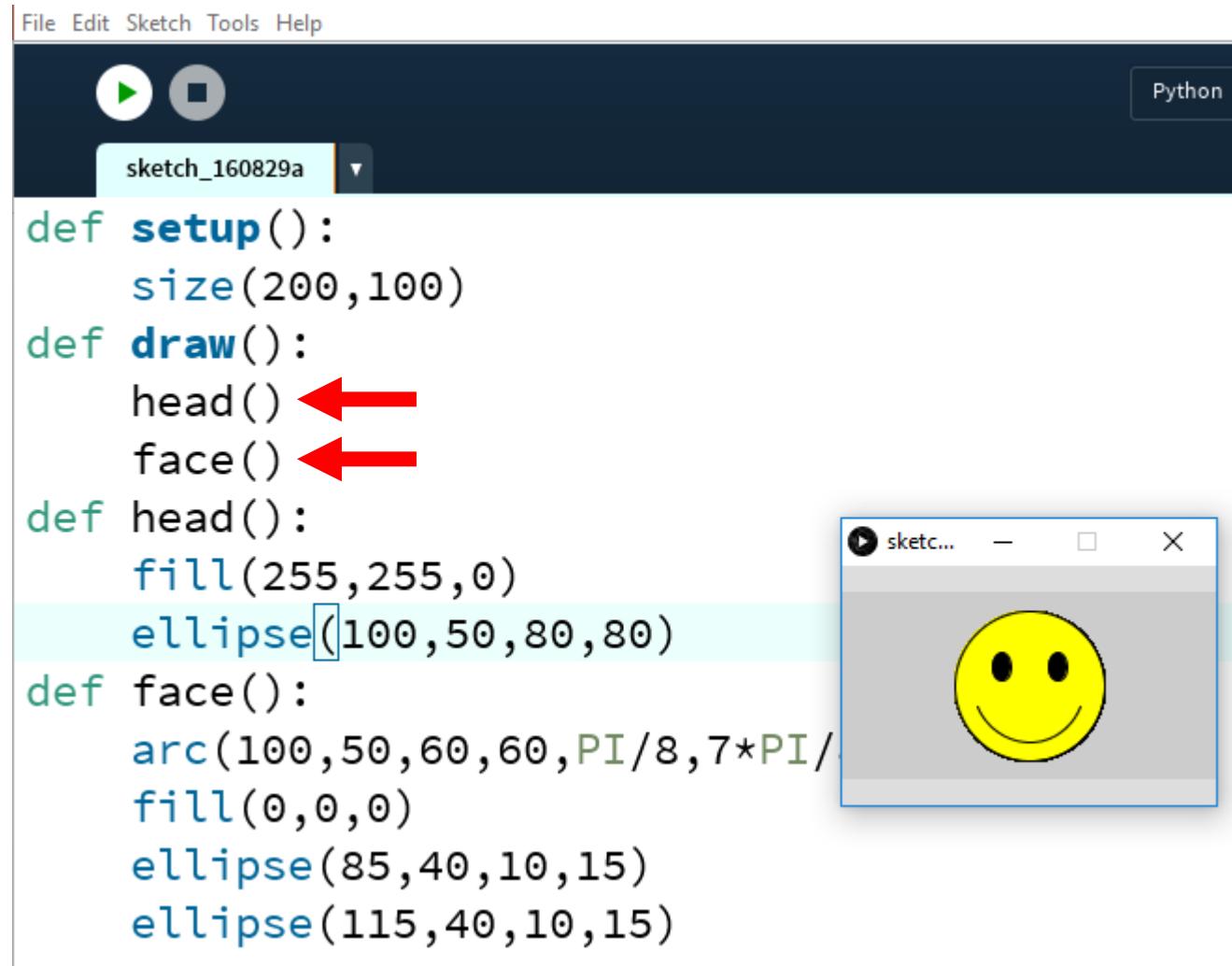


```
sketch_160829a
Python ▾

def setup():
 size(200,100)
def draw():
 face() ←
 head() ←
def head():
 fill(255,255,0)
 ellipse(100,50,80,80)
def face():
 arc(100,50,60,60,PI/8,7*PI/
 fill(0,0,0)
 ellipse(85,40,10,15)
 ellipse(115,40,10,15)
```



# The Order of the Function Calls



```
File Edit Sketch Tools Help
Python ▾
sketch_160829a
def setup():
 size(200,100)
def draw():
 head() ←
 face() ←
def head():
 fill(255,255,0)
 ellipse(100,50,80,80)
def face():
 arc(100,50,60,60,PI/8,7*PI/
 fill(0,0,0)
 ellipse(85,40,10,15)
 ellipse(115,40,10,15)
```

# Functions are an Example of Abstraction

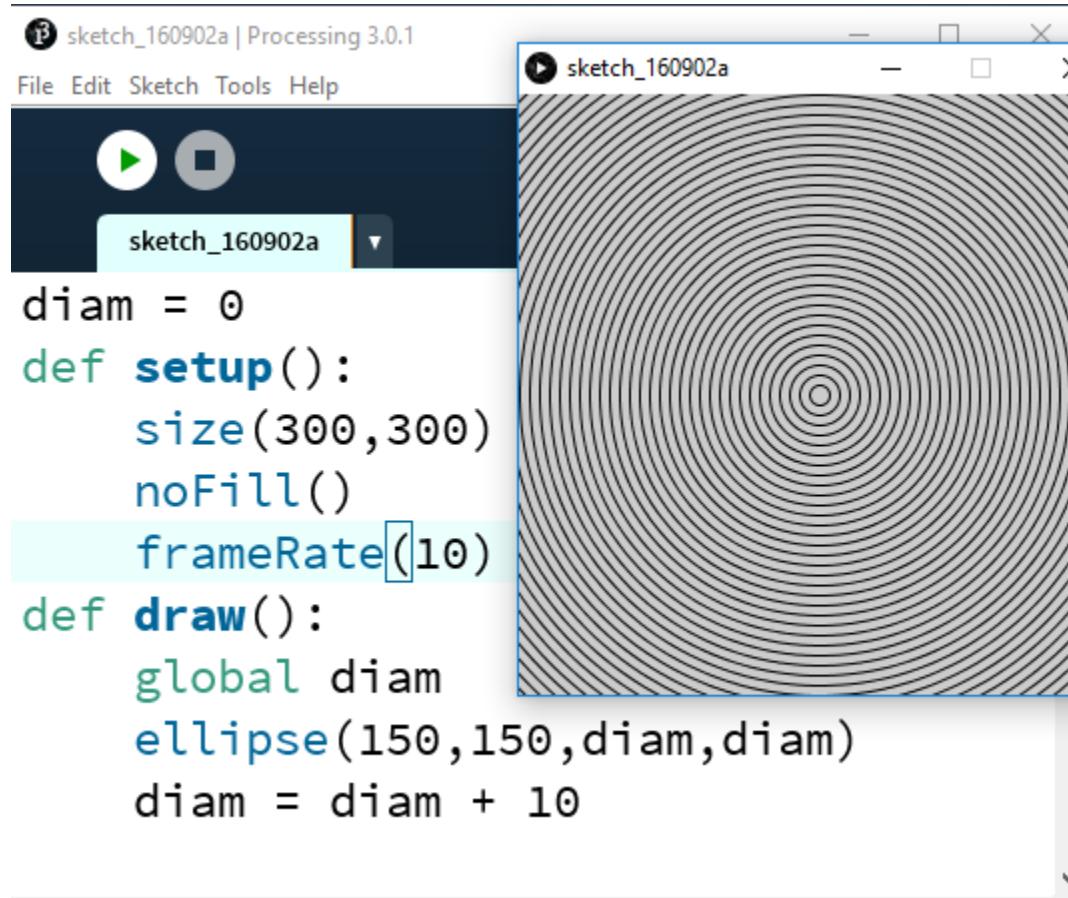
- Each function is a collection of lines of code.
- We give the function a name so that we can think of it as one thing.
- On the previous slide, `face()` and `head()` contained multiple lines of code, but we could think of them as performing one single job.
- Abstraction is reducing detail to manage complexity.

# Simple Animation

- We can take advantage of the way `setup()` and `draw()` work to create simple animations
- (demo)

# Simple Animation

We can take advantage of the way **setup()** and **draw()** work to create simple animations.



```
p sketch_160902a | Processing 3.0.1
File Edit Sketch Tools Help
sketch_160902a
diam = 0
def setup():
 size(300,300)
 noFill()
 frameRate(10)
def draw():
 global diam
 ellipse(150,150,diam,diam)
 diam = diam + 10
```

# Practice Quiz Question:

## What Will this Algorithm Display?

```
i <- 0
sum <- 0
REPEAT UNTIL (i = 4)
{
 i <- 1
 sum <- sum + i
 i <- i + 1
}
DISPLAY(sum)
```

- A. 0
- B. 6
- C. 10
- D. Nothing because there is an infinite loop

Equivalent Python Code:

```
i = 0
sum = 0
while i < 4:
 i = 1
 sum = sum + i
 i = i + 1
print(sum)
```

# **while** loops and **draw()**

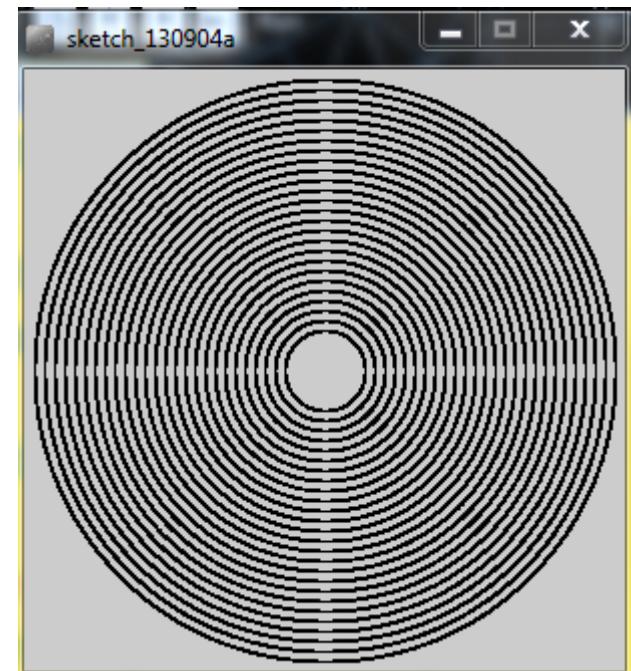
- One of the most confusing things in processing is the “invisible” loop that repeatedly cause the **draw()** function to run
- **while** loops will run *instantaneously*
- The **draw()** function loops over time

# while loops and draw()

This program draws all the circles immediately.

```
def setup():
 size(300,300)
 noFill()

def draw():
 diam = 40
 while diam < 300:
 ellipse(150,150,diam,diam)
 diam = diam + 10
```



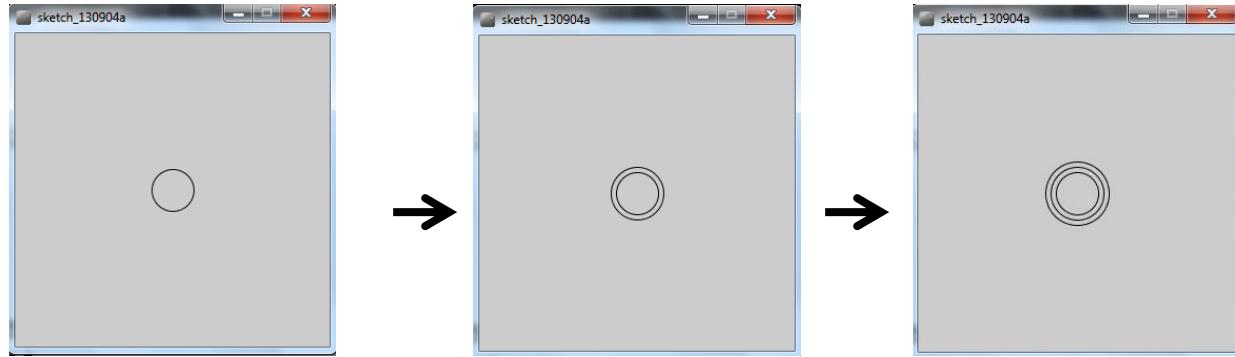
# while loops and draw()

This program draws the circles one at a time.

```
diam = 40
```

```
def setup():
 size(300,300)
 noFill()
```

```
def draw():
 global diam
 ellipse(150, 150, diam, diam)
 diam = diam + 10
```



# **while** loops and **draw()**

- The general rule of thumb is:
  - If you want the loop to run *immediately*, use a **while** loop
  - If you want an animation that changes over time, use the “invisible” **draw()** loop

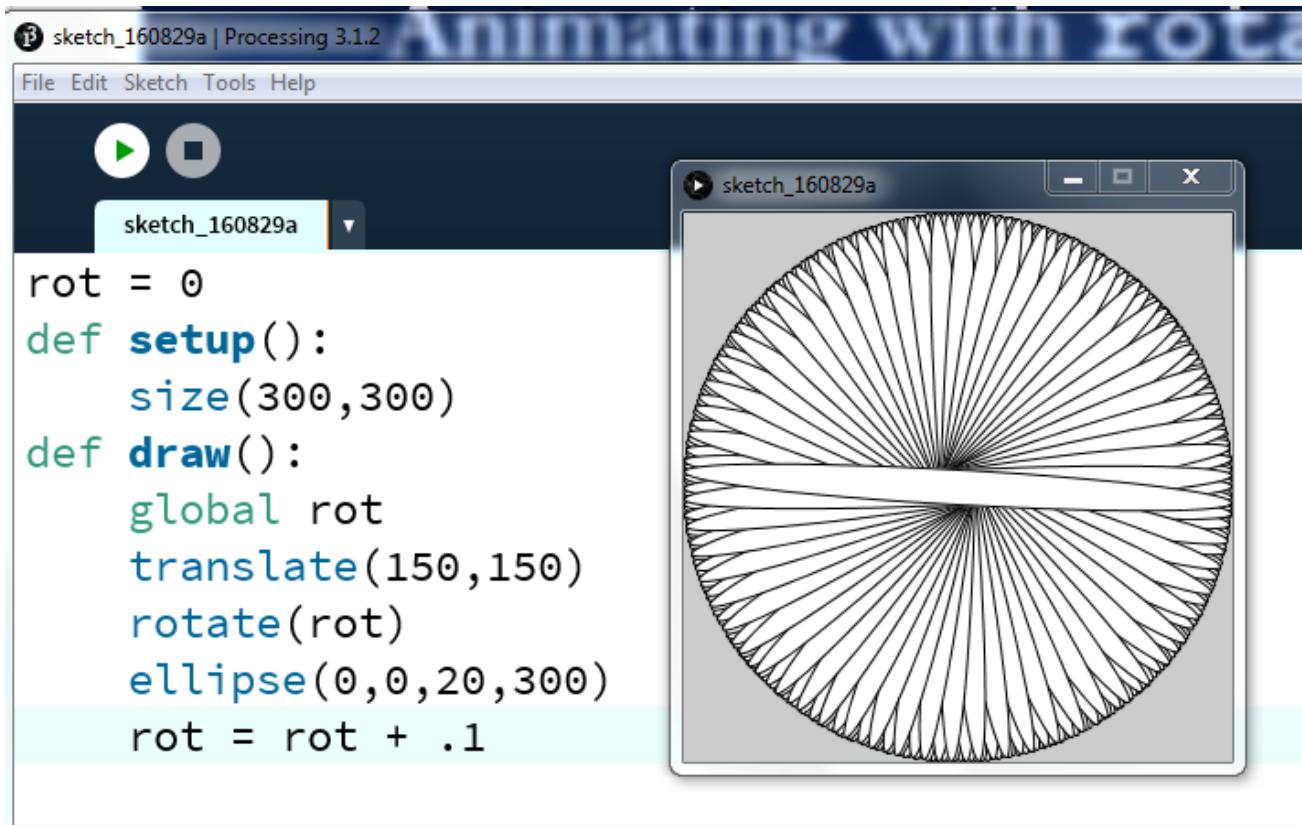
# Global Variables

- By default, variables in Python are local to a function.
- If we want functions to share the same variable we need to tell the function that the variable is **global**.

```
diam = 40
def setup():
 size(300,300)
 noFill()
def draw():
 global diam
 ellipse(150, 150, diam, diam)
 diam = diam + 10
```

# Animating with `rotate()`

You can make a animation that rotates objects by changing the amount of rotation (called `rot` in this program) over time.



The image shows the Processing IDE interface. The title bar says "sketch\_160829a | Processing 3.1.2". Below the title bar is a menu bar with File, Edit, Sketch, Tools, and Help. To the left of the code editor is a toolbar with a play button and a stop button. The code editor window has a dropdown menu showing "sketch\_160829a". The code itself is as follows:

```
rot = 0
def setup():
 size(300,300)
def draw():
 global rot
 translate(150,150)
 rotate(rot)
 ellipse(0,0,20,300)
 rot = rot + .1
```

To the right of the code editor is the preview window titled "sketch\_160829a". It displays a circular pattern of many thin black lines radiating from a central point, creating a fan-like effect. A horizontal ellipse is centered at the origin of the circle. The preview window has standard window controls (minimize, maximize, close).

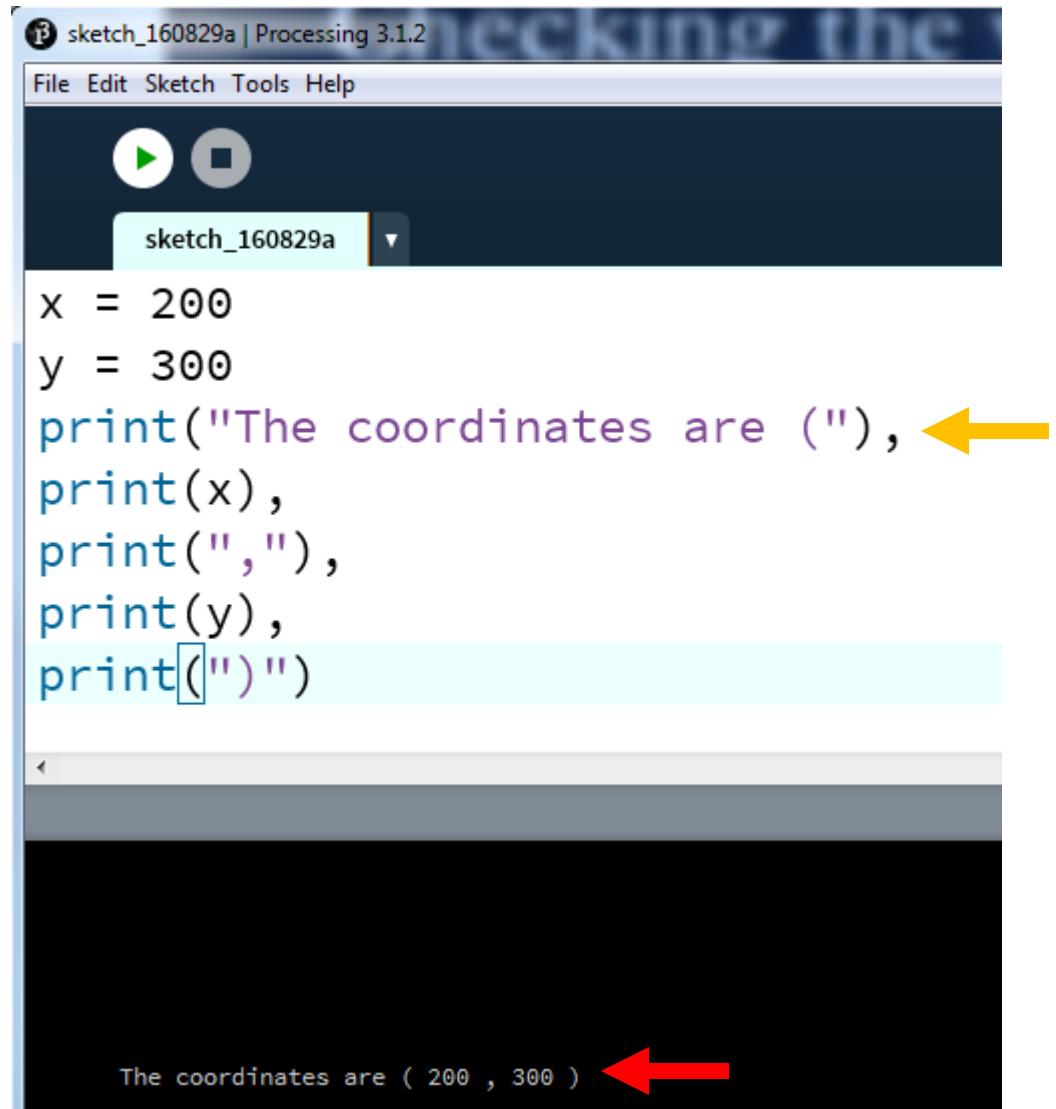
# Checking the Value in a Variable

- One of the confusing things in programming is keeping track of the values as they change.
- You can print the values to the black box at the bottom of Processing with

```
-print(), # on same line
-print() # print first
 # then next line
```

# Checking the Value in a Variable

Adding text to  
label your output  
can make it more  
readable.



The screenshot shows the Processing 3.1.2 software interface. The title bar says "sketch\_160829a | Processing 3.1.2". The menu bar includes File, Edit, Sketch, Tools, and Help. Below the menu is a toolbar with a play button and a stop button. The sketch window displays the following code:

```
x = 200
y = 300
print("The coordinates are ("),
print(x),
print(","),
print(y),
print(")")
```

A yellow arrow points to the opening parenthesis of the print statement. The output window at the bottom shows the result of the code execution:

The coordinates are ( 200 , 300 )

A red arrow points to the output text.

# Printing Text

- Text can be words, sentences, paragraphs, numbers and more.
- It's any collection of characters, punctuation, numbers and spaces.
- You can print text with **double quotes** .

```
print("Testing, 1, 2, 3")
```

- "Testing, 1, 2, 3" is an example of a *literal*.

# Local Variables

- If you **declare a variable** in a function, you can only use it in that function.
- The variable is called a local variable.



A screenshot of the Processing 3.1.2 software interface. The title bar says "sketch\_160830b | Processing 3.1.2". The menu bar includes File, Edit, Sketch, Tools, and Help. Below the menu is a toolbar with play and stop buttons. A dropdown menu shows "sketch\_160830b". The code editor contains the following script:

```
def setup():
 num = 5
 print(num)
def draw():
 print(num)
```

A green arrow points to the line "print(num)" in the draw() function. The status bar at the bottom displays the error message "NameError: global name 'num' is not defined".

# The Basic Scope Rule\*

- **Scope** is like a neighborhood, it's where the variable name is known and understood.
- The basic scope rule\* is **the scope of variable begins with its declaration and ends with the end of the indented block of code where it was declared.**

\*there are exceptions to this rule, but we don't really care and we certainly aren't going to worry about it now

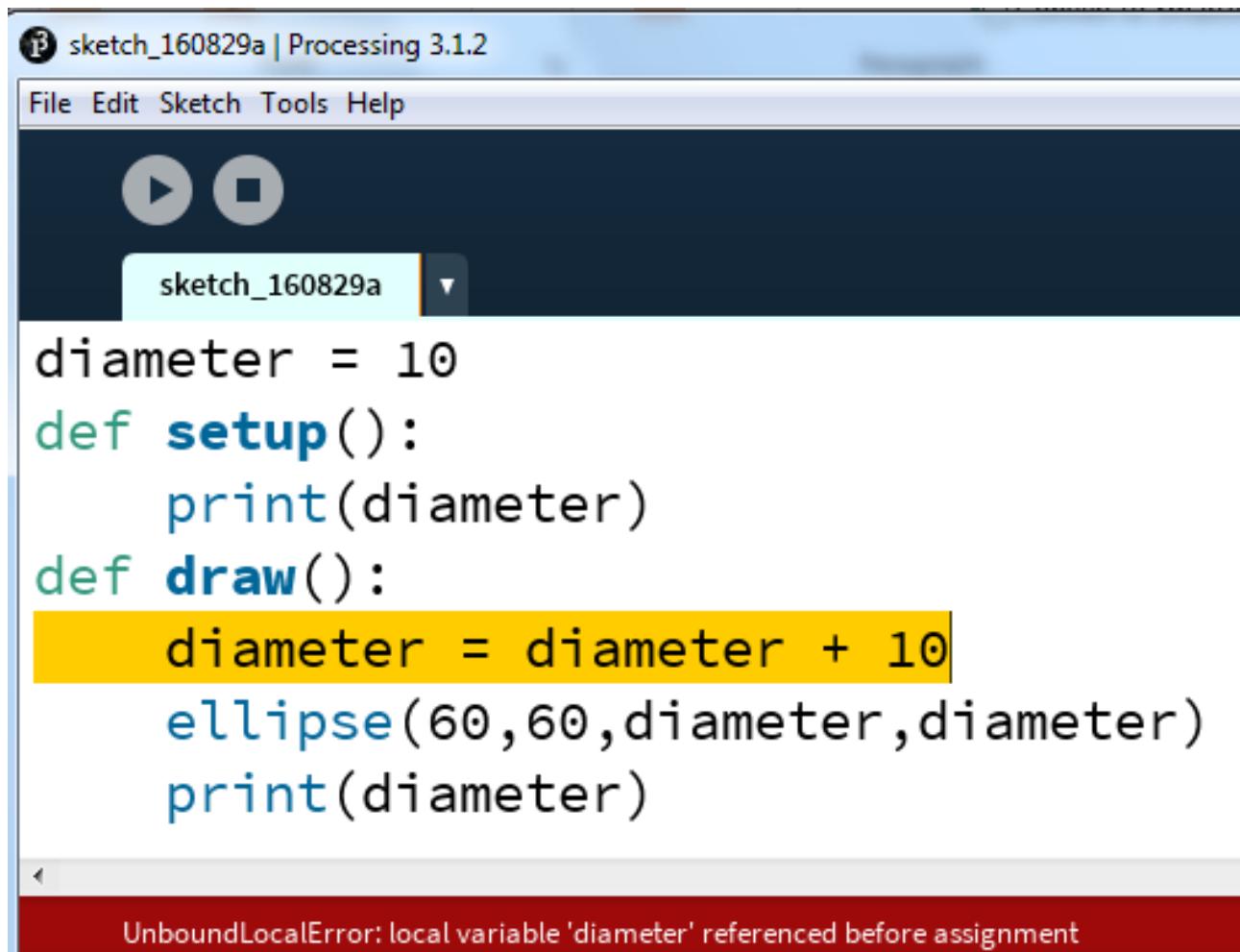
# The scope of diameter is in Orange

```
def draw():
 ellipse(30,30,50,80)
 noFill()
 strokeWeight(5)
 diameter = 100
 stroke(255,0,0)
 ellipse(60,60,diameter,diameter)
 println(diameter)
end of indented code block
```

If you declare the variable at the top of the program outside of any function, you can use it in any function where you label it global

```
diameter = 10
def setup():
 global diameter
 print(diameter)
def draw():
 global diameter
 diameter = diameter + 10
 ellipse(60, 60, diameter, diameter)
 print(diameter)
```

If you don't label it **global** you may get an error.



The screenshot shows the Processing 3.1.2 software interface. The title bar reads "sketch\_160829a | Processing 3.1.2". The menu bar includes "File", "Edit", "Sketch", "Tools", and "Help". Below the menu is a toolbar with play and stop buttons. The code editor window contains the following code:

```
diameter = 10
def setup():
 print(diameter)
def draw():
 diameter = diameter + 10
 ellipse(60,60,diameter,diameter)
 print(diameter)
```

The line "diameter = diameter + 10" is highlighted in yellow, indicating it is the source of the error. A red status bar at the bottom displays the error message: "UnboundLocalError: local variable 'diameter' referenced before assignment".

# Practice Quiz Question

Which picture matches the output of this program?

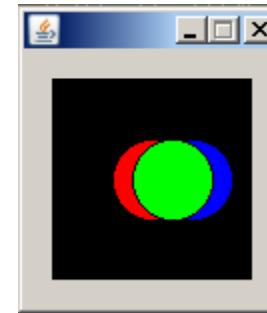
```
def setup():
 background(0,0,0)

def draw():
 anotherMystery()
 mysteryFunction()
 fill(255,0,0)
 ellipse(50,50,40,40)

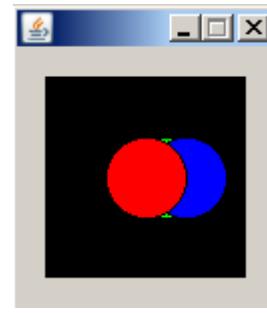
def mysteryFunction():
 fill(0,255,0)
 ellipse(60,50,40,40)

def anotherMystery():
 fill(0,0,255)
 ellipse(70,50,40,40)
```

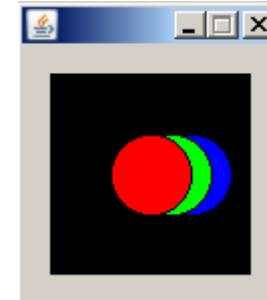
A



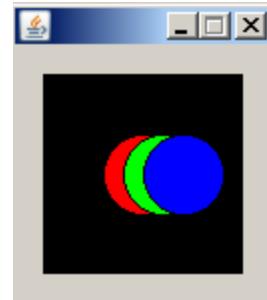
B



C



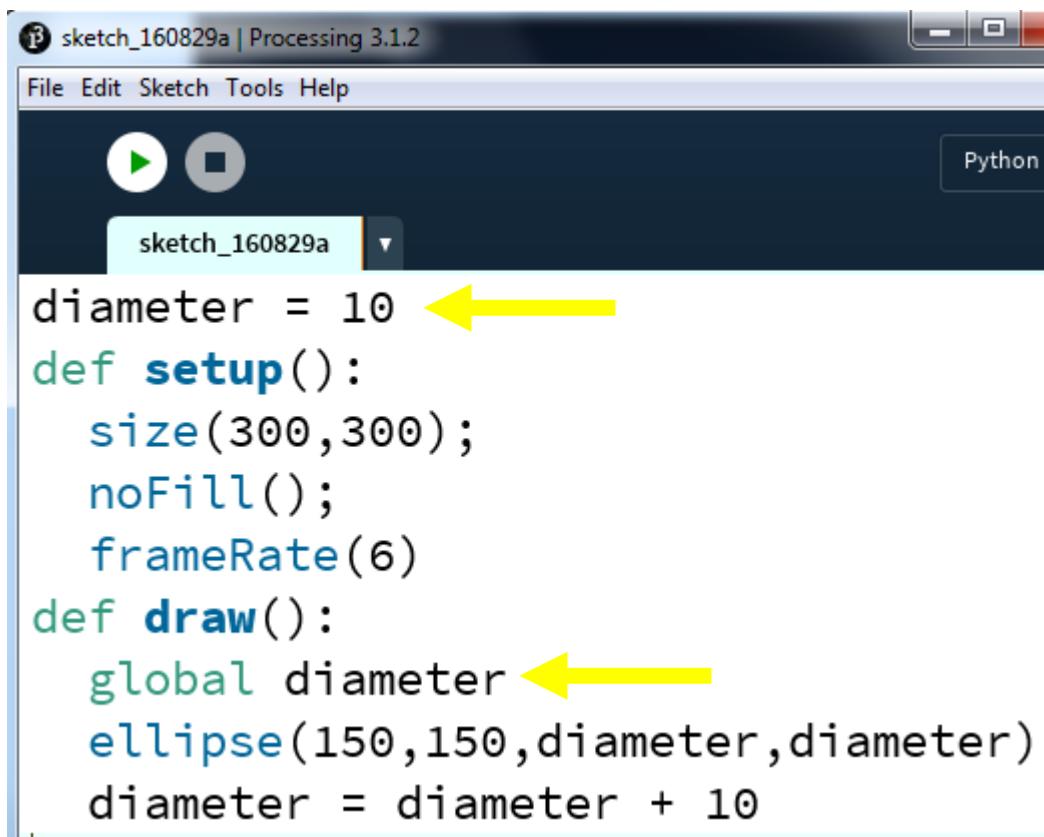
D



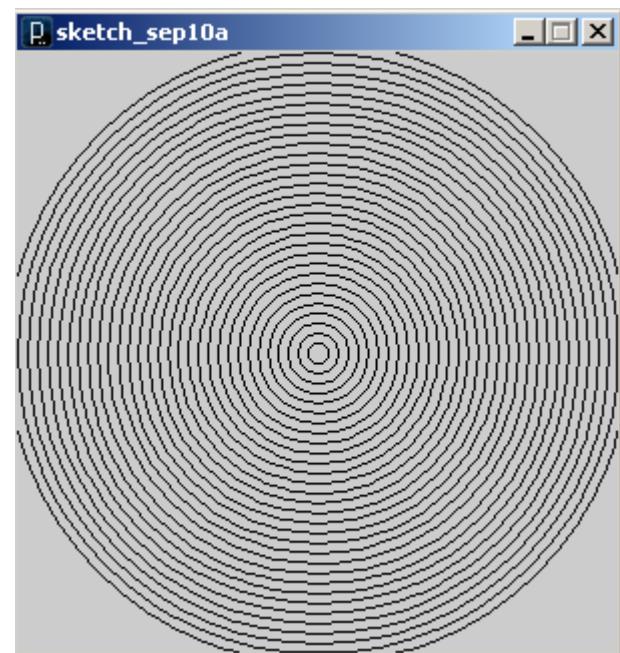
# Local vs. Global

- A *global* variable can be used anywhere in the program.
- You create a *global* variable by declaring it at the top of the program.
- To use the global variable in the function label it **global**.
- A *local* variable is declared in a function and can only be used in that function.

# With a global variable the Circle Gets Bigger.



```
diameter = 10
def setup():
 size(300,300);
 noFill();
 frameRate(6)
def draw():
 global diameter
 ellipse(150,150,diameter,diameter)
 diameter = diameter + 10
```



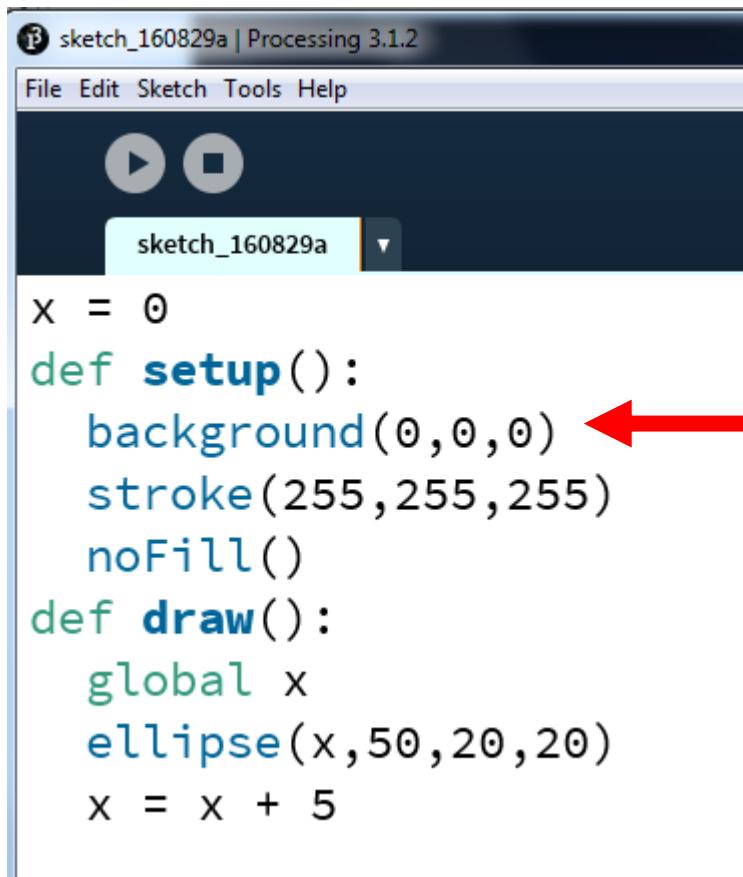
# With a local variable the Circle DOESN'T Get Bigger.

The image displays two Processing sketches. The left sketch, titled "sketch\_160829a", contains the following Python code:

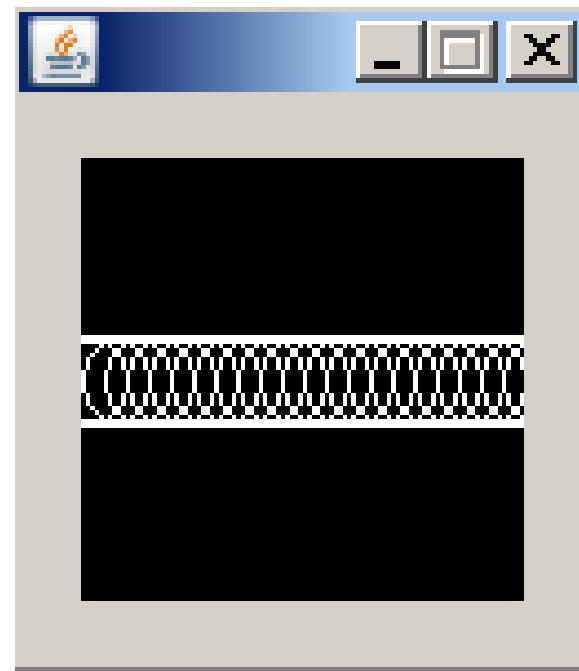
```
def setup():
 size(300,300);
 noFill();
 frameRate(6)
def draw():
 diameter = 10 ←
 ellipse(150,150,diameter,diameter)
 diameter = diameter + 10
```

A red arrow points to the line `diameter = 10`, highlighting it. The right sketch, titled "sketch\_sep10a", shows a single small circle at the center of the canvas, indicating that the variable `diameter` is not being updated or is not being used correctly.

# The Background Drawn Once in setup () (leaves a trail of circles)



```
sketch_160829a | Processing 3.1.2
File Edit Sketch Tools Help
sketch_160829a
x = 0
def setup():
 background(0,0,0) ←
 stroke(255,255,255)
 noFill()
def draw():
 global x
 ellipse(x,50,20,20)
 x = x + 5
```



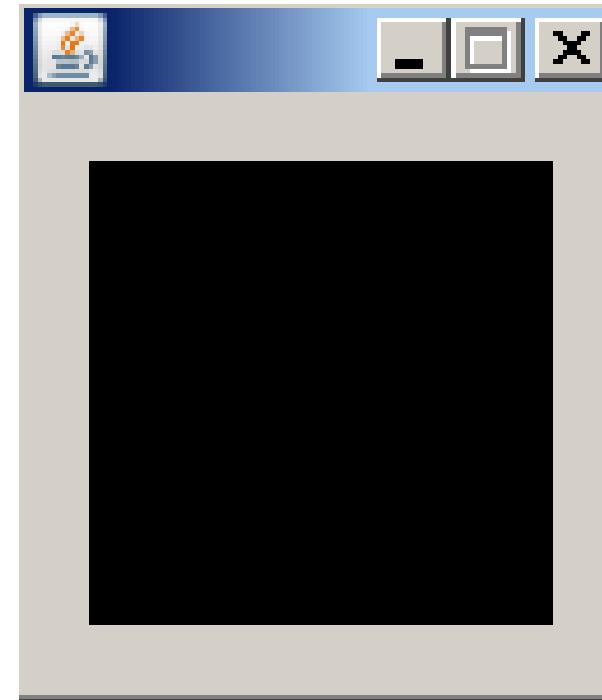
# The Background is Drawn Every Time the Screen is Drawn (no trail)

sketch\_160829a | Processing 3.1.2

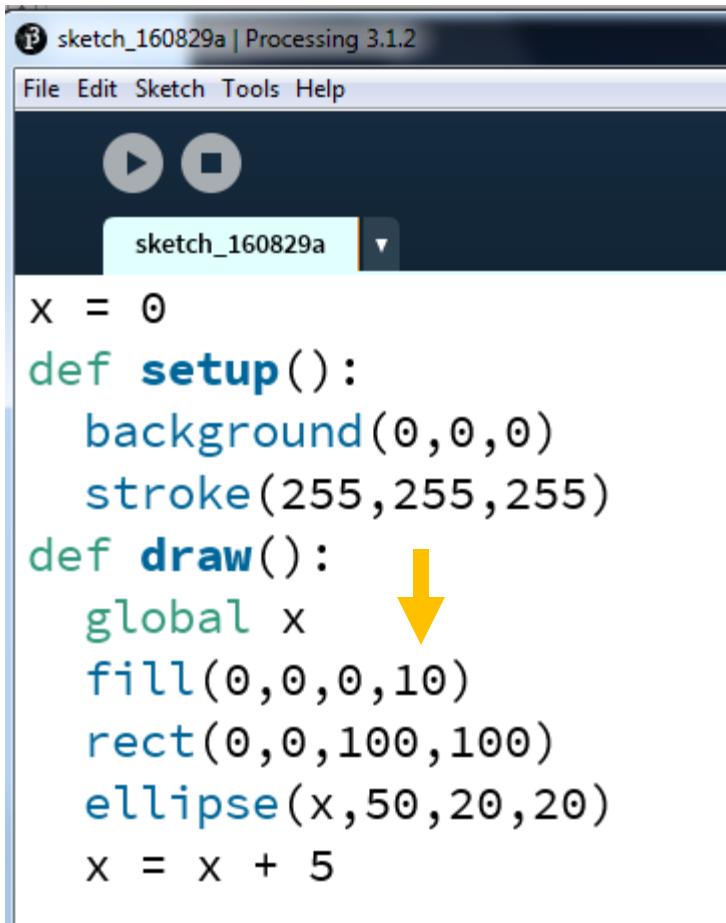
File Edit Sketch Tools Help

sketch\_160829a

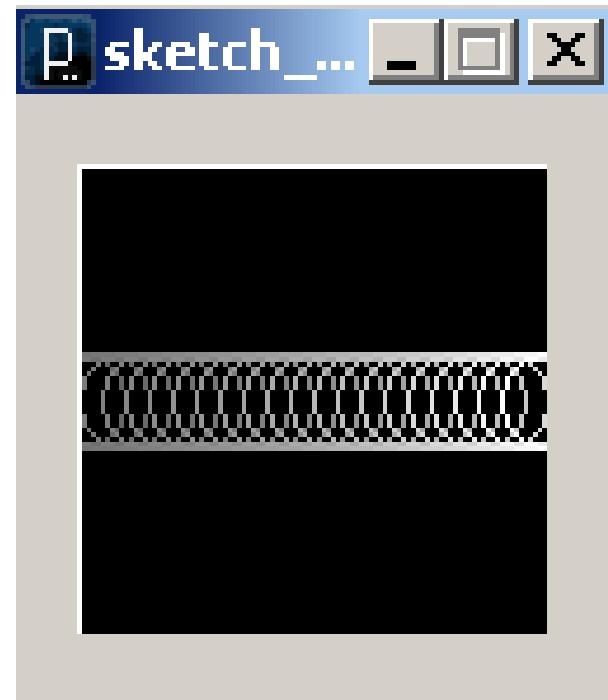
```
x = 0
def setup():
 stroke(255,255,255)
 noFill()
def draw():
 global x
 background(0,0,0) ←
 ellipse(x,50,20,20)
 x = x + 5
```



# A Black Rectangle Drawn Every Time with Opacity (faint trail)



```
sketch_160829a | Processing 3.1.2
File Edit Sketch Tools Help
sketch_160829a
x = 0
def setup():
 background(0,0,0)
 stroke(255,255,255)
def draw():
 global x
 ↓
 fill(0,0,0,10)
 rect(0,0,100,100)
 ellipse(x,50,20,20)
 x = x + 5
```



# The if Statement

```
if num > 150:
 print("num is bigger than 150")
 print("so it is pretty big")
```

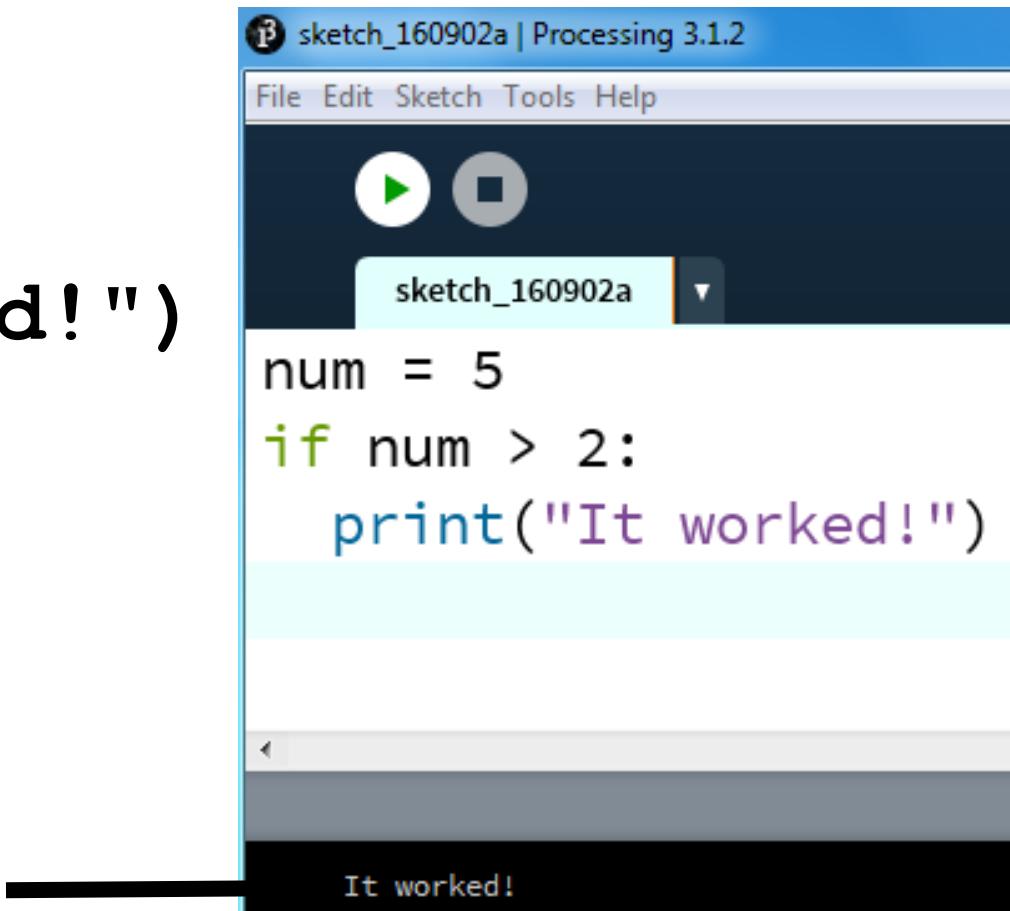
- a condition
- Followed by a colon
- Then one or more lines in a block of indented code
- The condition is any expression that evaluates to true or false.
- The if-statement evaluates the condition and then runs the block of indented code only if the condition is true. If the condition is false, the block is skipped.

# if statement

```
num = 5

if num > 2:

 print("It worked!")
```



The screenshot shows the Processing IDE interface. The title bar reads "sketch\_160902a | Processing 3.1.2". Below the title bar are menu options: File, Edit, Sketch, Tools, Help. In the center, there are two large buttons: a green play button and a grey square button. To the right of these buttons is the sketch name "sketch\_160902a" and a dropdown arrow. The main workspace contains the following code:

```
num = 5

if num > 2:
 print("It worked!")
```

The line "print("It worked!")" is highlighted with a light blue background. At the bottom of the screen, the output window displays the text "It worked!".

# Empty!

```
num = 1
```

```
if num > 2:
 print("It
worked!")
```



A screenshot of the Processing 3.1.2 software interface. The title bar says "sketch\_160902a | Processing 3.1.2". The menu bar includes "File", "Edit", "Sketch", "Tools", and "Help". Below the menu is a toolbar with a play button and a stop button. The sketch name "sketch\_160902a" is displayed in the center. The code area contains the following code:

```
num = 1
if num > 2:
 print("It worked!")
```

# Using an if to "start over if it gets too big"

```
diameter = 10
def setup():
 size(300,300)
 background(0,0,0)
 noFill()
 stroke(255,255,255)
 frameRate(10)
def draw():
 global diameter
 ellipse(150,150,diameter,diameter)
 diameter = diameter + 20
 if diameter > 300:
 background(0,0,0)
 diameter = 10
```



# Find the Output

```
def setup():{
 size(300,300)

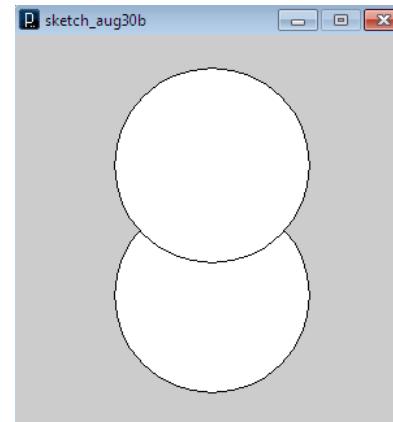
def draw():
 mystery3()
 mystery1()

def mystery1():
 ellipse(150,100,150,150)

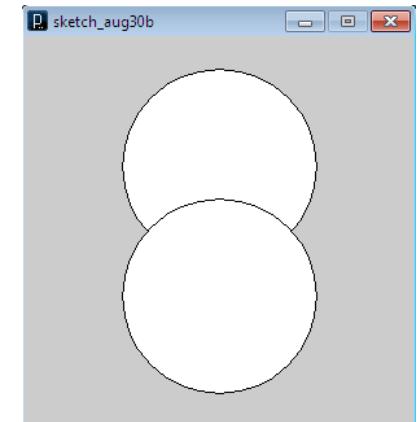
def mystery2():
 ellipse(150,150,150,150)

def mystery3():
 ellipse(150,200,150,150)
```

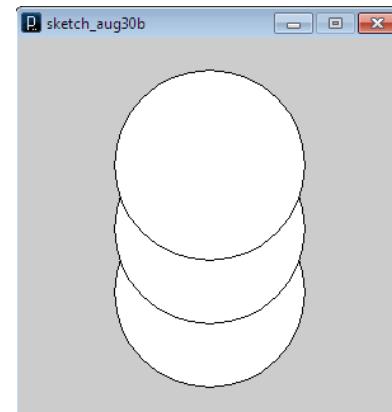
A



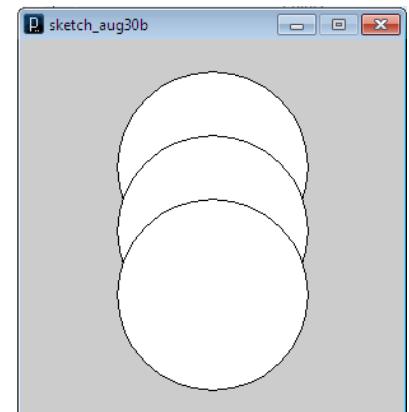
B



C



D



# A Circle that Moves Left to Right

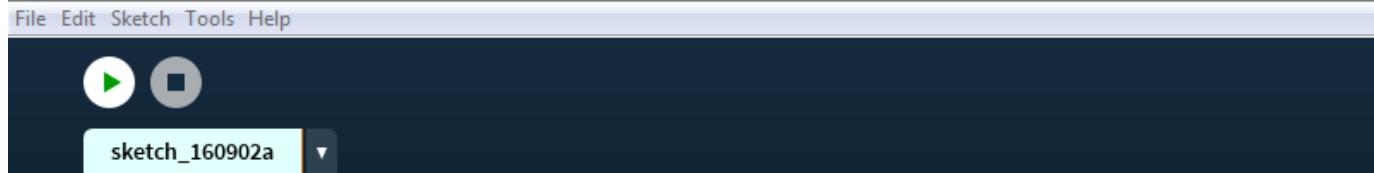
```
x = 0
def setup():
 size(300,300)
def draw():
 global x
 background(0)
 ellipse(x,150,30,30)
 x = x + 1
```

# Adding a `change` Variable so that the Circle Moves Back & Forth

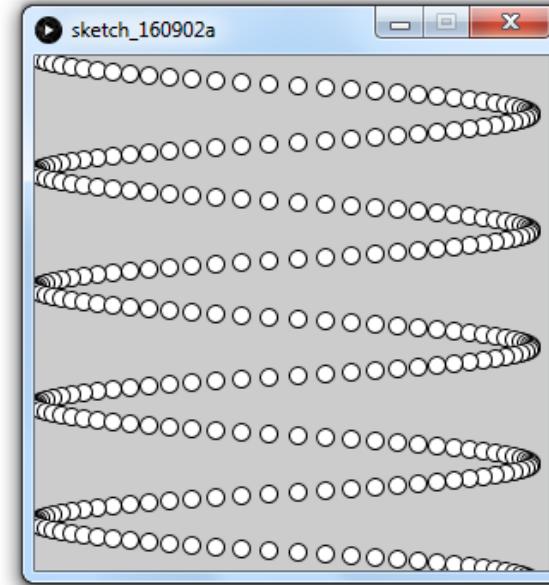
```
x = 0;
change = 1;
def setup():
 size(300,300)
def draw():
 global x
 global change
 background(0)
 ellipse(x,150,30,30)
 x = x + change
 if x > 300:
 change = -1
 if x < 0:
 change = 1
```

# Changing the Amount of Change

The rate of change in the x and y direction is the slope. If the rate of change stays the same, the result is a straight line. Changing the amount of change produces a curve.



```
x = 0
y = 0
xChange = 1
def setup():
 size(300,300)
def draw():
 global x, y, xChange
 ellipse(x,y,10,10)
 x = x + xChange
 y = y + 1
 if x < 150:
 xChange = xChange + 1
 if x >= 150:
 xChange = xChange - 1
```



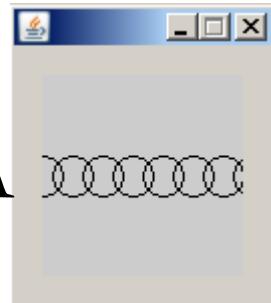
# Practice Quiz Question

Find the output that best matches the following code.

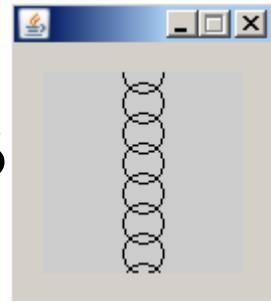
```
mystery = 0
def setup():
 noFill()
def draw():
 global mystery

 ellipse(50,mystery,20,20)
 mystery = mystery + 15
```

A



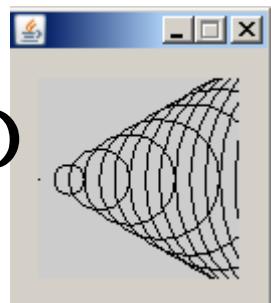
B



C



D



# Input

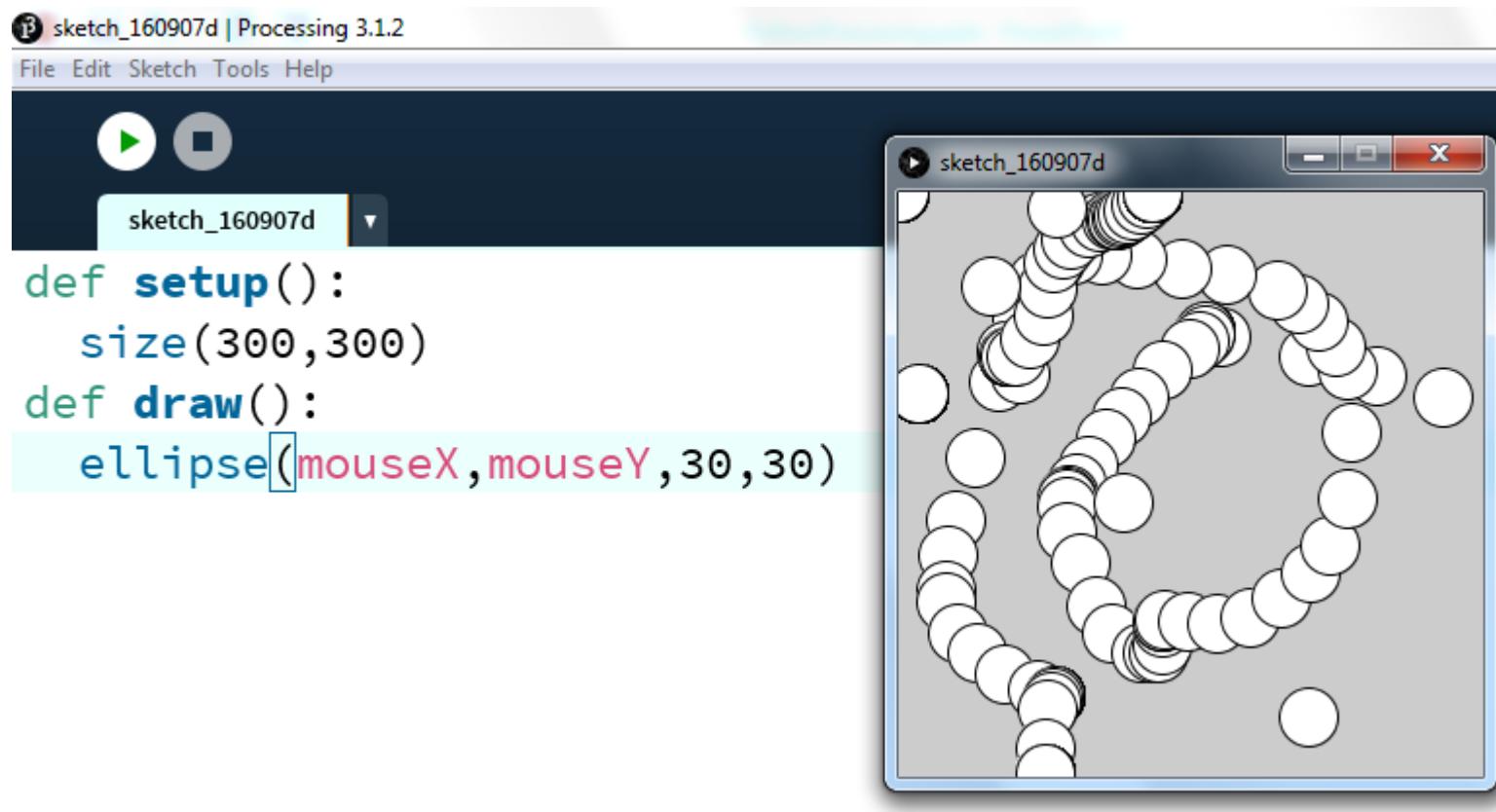
- Most programs we use get input from people
- The most common ways people provide input are:
  - Typing on the keyboard
  - Moving and/or clicking the mouse

# Input

- Processing makes input from the mouse and keyboard about as easy as it gets.
- There are several predefined "system variables."
  - `mouseX`
  - `mouseY`
  - `key`
- “Predefined system variables” are variables that Processing has already *declared* and *initialized*.

# Moving a Circle with the Mouse

The ellipse will track the mouse.



The image shows the Processing IDE interface. The top bar displays "sketch\_160907d | Processing 3.1.2". The menu bar includes "File", "Edit", "Sketch", "Tools", and "Help". On the left, there are buttons for play/pause and sketch selection. The code editor contains the following script:

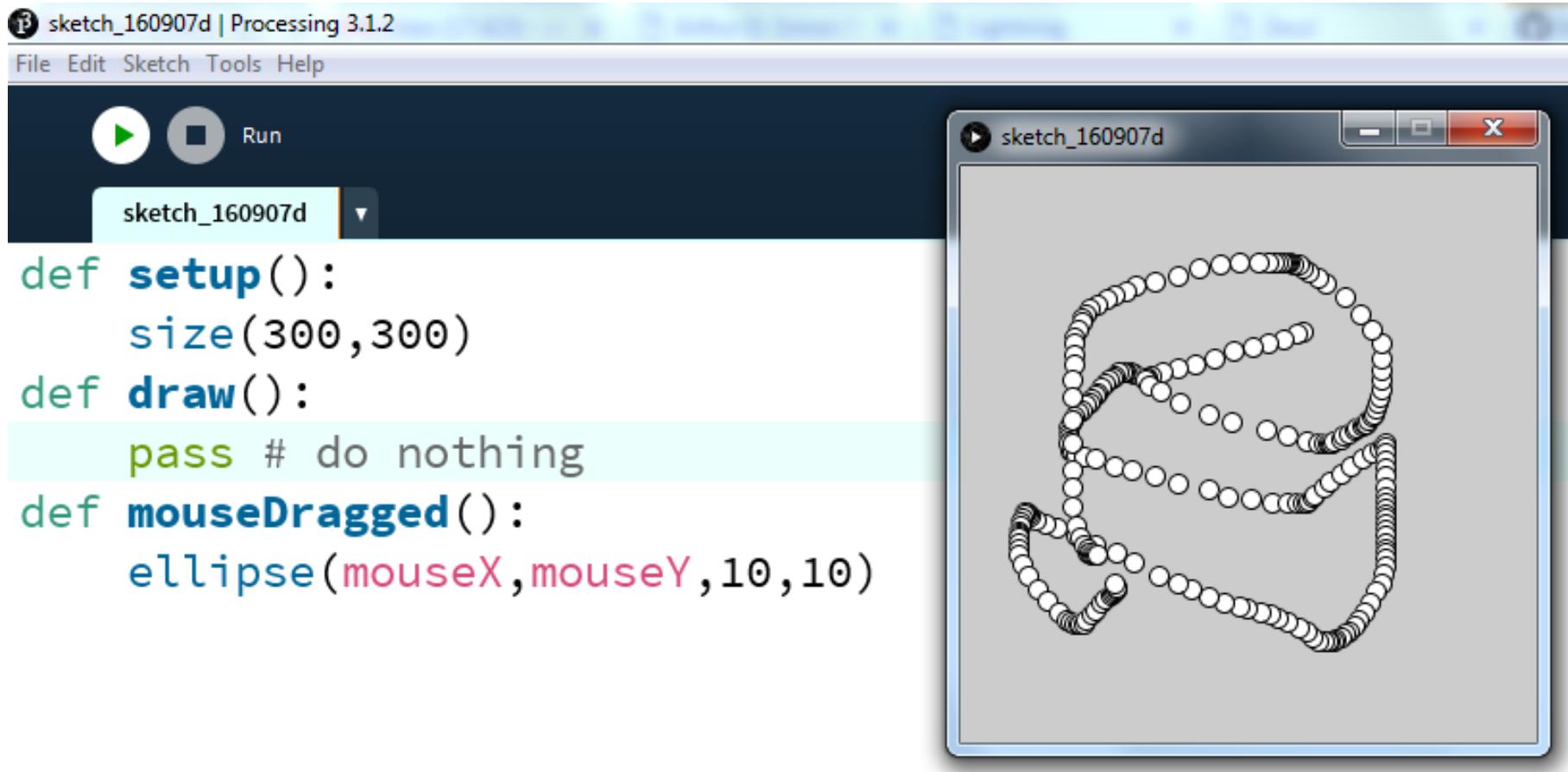
```
def setup():
 size(300,300)
def draw():
 ellipse(mouseX, mouseY, 30, 30)
```

The preview window titled "sketch\_160907d" shows a series of white circles connected by thin black lines, forming a path that follows the movement of the mouse cursor.

# Event Handlers: Functions that Respond to Events

- **keyTyped()**
- **keyPressed()**
- **keyReleased()**
- **mouseMoved()**
- **mousePressed()**
- **mouseReleased()**
- **mouseClicked()**
- **mouseDragged()**

# Painting Only if Mouse is dragged



sketch\_160907d | Processing 3.1.2

File Edit Sketch Tools Help

Run

sketch\_160907d

```
def setup():
 size(300,300)
def draw():
 pass # do nothing
def mouseDragged():
 ellipse(mouseX, mouseY, 10, 10)
```

The screenshot shows the Processing IDE interface with a sketch titled "sketch\_160907d". The code defines three functions: "setup", "draw", and "mouseDragged". The "setup" function sets the canvas size to 300x300 pixels. The "draw" function contains a single line of code: "pass # do nothing". The "mouseDragged" function uses the "ellipse" command to draw a small circle at the current mouse position, with a diameter of 10 pixels. This results in a continuous, winding line of circles as the mouse is moved across the screen.

# More on if: Relational Operators

<   <=   ==   >=   >   !=

The operators that are used for comparisons are called *relational operators*

# More on **if**: Relational Operators

<   <=   ==   >=   >   !=

**if** statements can be created by comparing two things with a relational operator

**if** 5 < 3:

**if** num == 5:

**if** num >= 5:

**if** num != 5:

if the **comparison** in the is true, the code in the indented block after the **if** executes. Otherwise it is skipped.

# More on if: Relational Operators

```
if num1 < num2 < num3:
```

*Comparisons must be done two at a time*

## = VS. ==

- A *single equals* (called the assignment operator) MAKES two things equal

`num = 3;`

- Don't put this in an `if`— it will always be true!
- The *double equals* asks a question: Are these two things equal?

`num==3`

- Use the *double equals* anywhere you would test a condition: `if`, etc.

```
if num==3:
 print("num is three")
```

# Moving the Ellipse with the Keyboard

The screenshot shows the Processing IDE interface. The title bar reads "sketch\_160907d | Processing 3.1.2". The menu bar includes "File", "Edit", "Sketch", "Tools", and "Help". Below the menu is a toolbar with a play button, a stop button, and a dropdown menu set to "sketch\_160907d". The code editor contains the following Pseudocode:

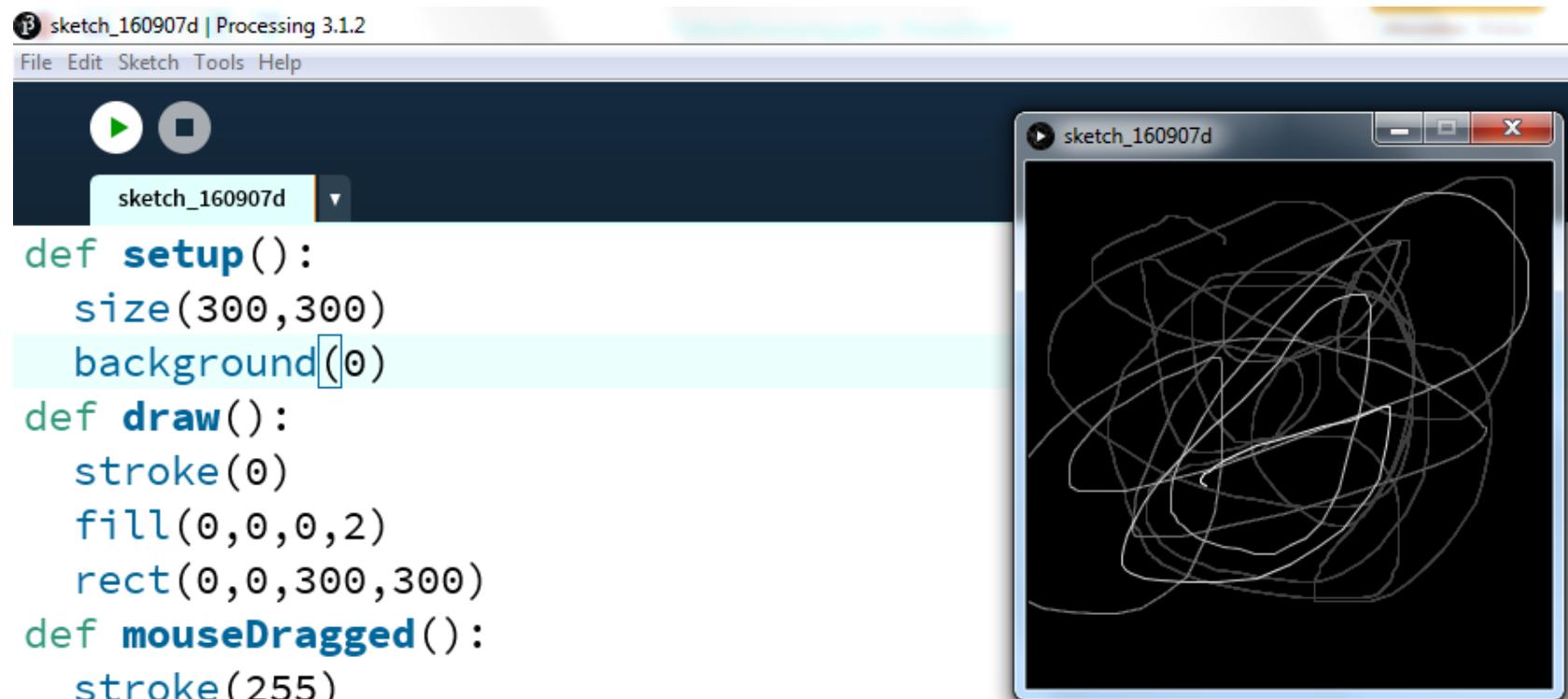
```
x = y = 150
def setup():
 size(300,300)
def draw():
 ellipse(x,y,30,30)
def keyPressed():
 global y
 if key == 'w':
 y = y - 5
```

The processing window displays a single black ellipse centered at (150, 150) on a light gray background. The ellipse has a thin black outline and a white fill.

# More system variables: `pmouseX` and `pmouseY`

- `mouseX` and `mouseY` hold the *current* position of the mouse.
- `pmouseX` and `pmouseY` hold the *previous* position of the mouse.

# A Drawing Program



The image shows a screenshot of the Processing 3.1.2 software interface. The title bar reads "sketch\_160907d | Processing 3.1.2". The menu bar includes "File", "Edit", "Sketch", "Tools", and "Help". Below the menu is a toolbar with a play button, a square, and a red X. A dropdown menu shows "sketch\_160907d". The code editor contains the following Pseudocode:

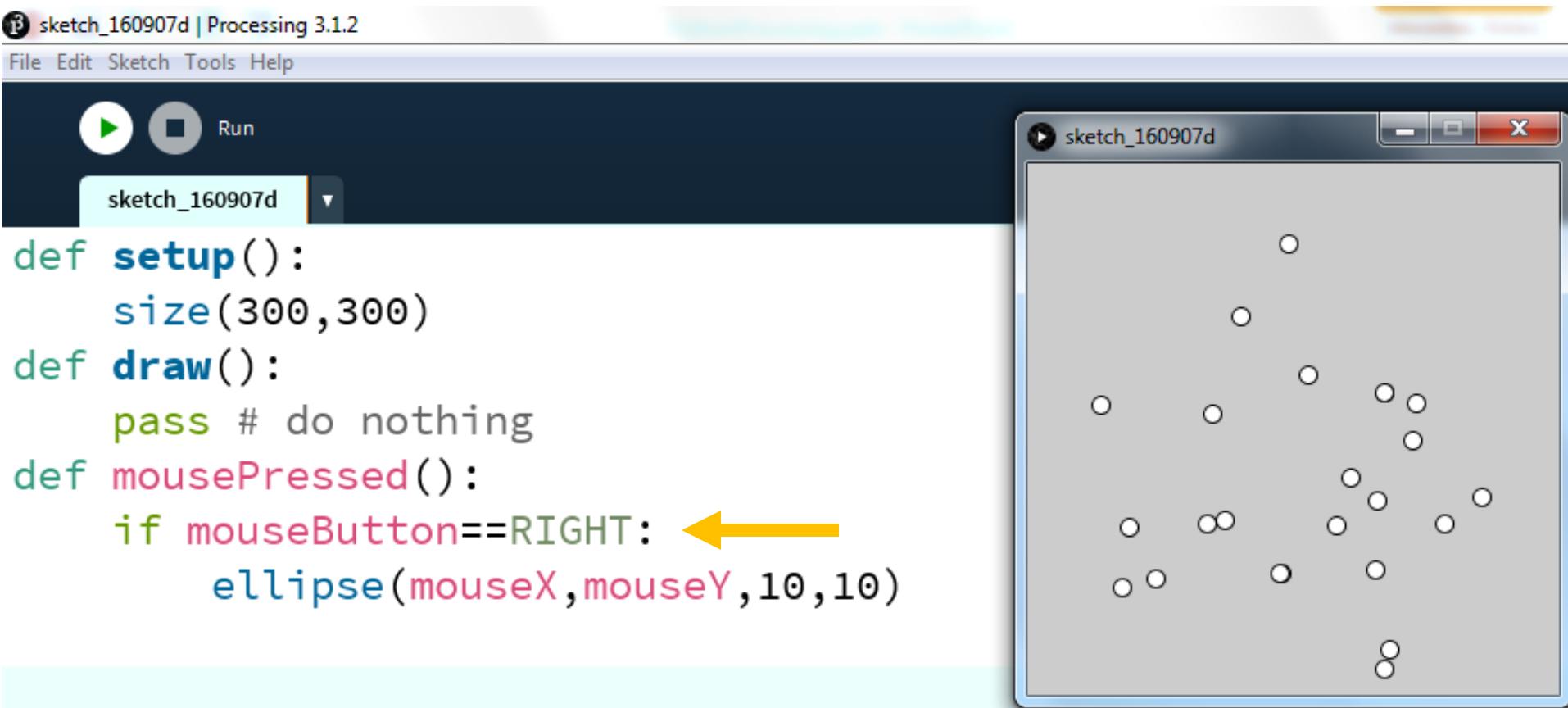
```
def setup():
 size(300,300)
 background(0)
def draw():
 stroke(0)
 fill(0,0,0,2)
 rect(0,0,300,300)
def mouseDragged():
 stroke(255)
 line(mouseX,mouseY,pmouseX,pmouseY)
```

The preview window on the right displays a black canvas with a series of overlapping, curved white lines forming a complex, abstract shape.

# The complete list of input functions and variables is in the API

| Mouse           | Keyboard      |
|-----------------|---------------|
| mouseButton     | key           |
| mouseClicked()  | keyCode       |
| mouseDragged()  | keyPressed()  |
| mouseMoved()    | keyPressed    |
| mousePressed()  | keyReleased() |
| mousePressed    | keyTyped()    |
| mouseReleased() |               |
| mouseWheel()    |               |
| mouseX          |               |
| mouseY          |               |
| pmouseX         |               |
| pmouseY         |               |

# Drawing an ellipse only if the right mouse button is pressed



sketch\_160907d | Processing 3.1.2

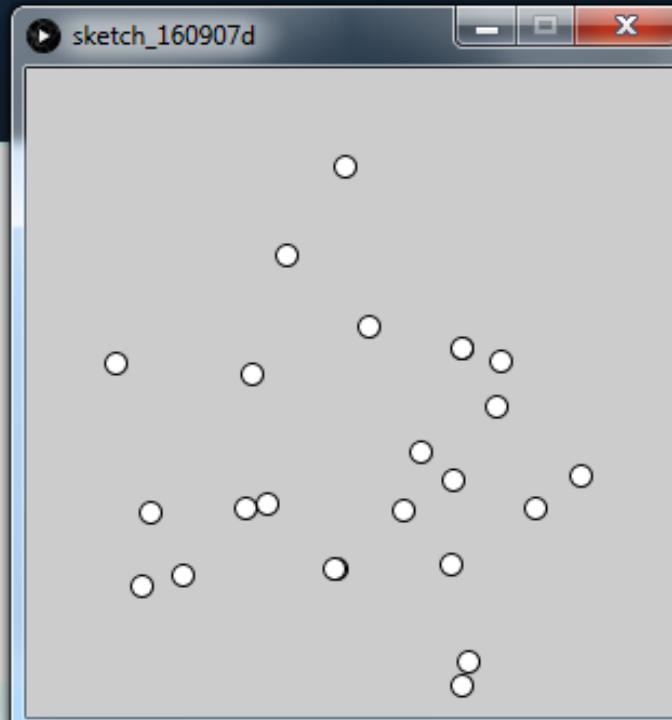
File Edit Sketch Tools Help

Run

sketch\_160907d

```
def setup():
 size(300,300)
def draw():
 pass # do nothing
def mousePressed():
 if mouseButton==RIGHT: ←
 ellipse(mouseX,mouseY,10,10)
```

sketch\_160907d



# Logical Operators

| and   | True  | False | or    | True | False | not |       |      |
|-------|-------|-------|-------|------|-------|-----|-------|------|
| True  | True  | False | True  | True | True  |     | False | True |
| False | False | False | False | True | False |     |       |      |

- and or not
- Used to combine multiple conditions ("tests")
- Truth tables shown above
- Examples, for nCount = 110

0 < nCount and nCount <= 100

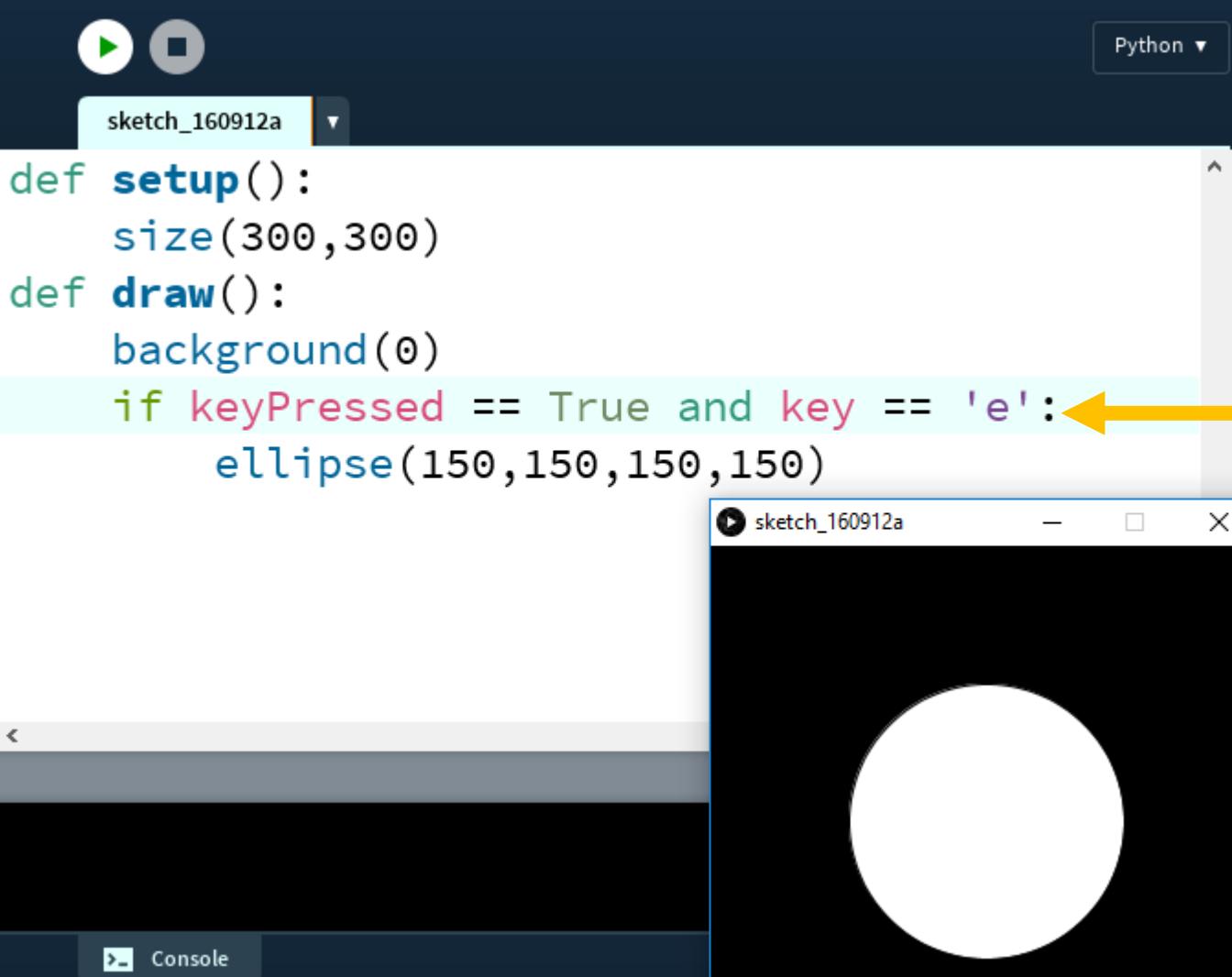
0 < nCount or nCount <= 100

not nCount > 100

not (200 >= nCount and nCount >= 13)

200 < nCount or nCount < 100

# Drawing an ellipse only if the *e* on the keyboard is pressed



```
sketch_160912a
Python ▾

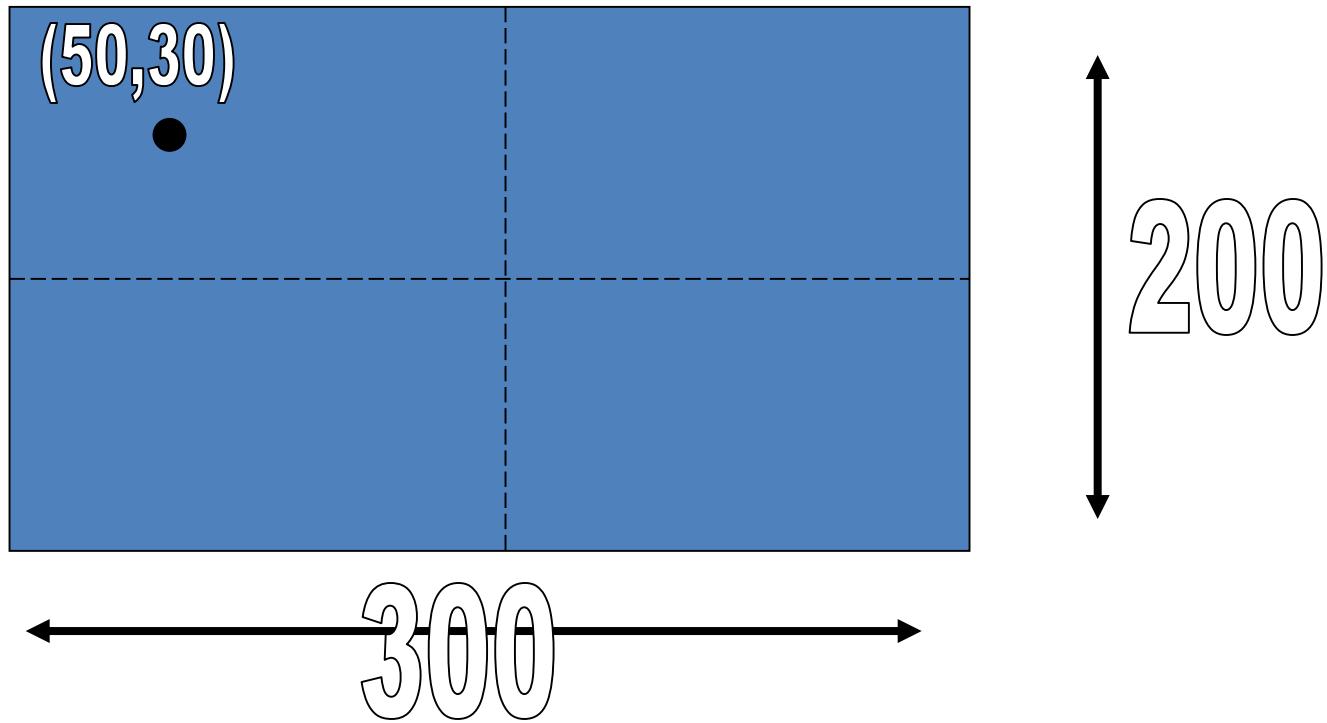
def setup():
 size(300,300)
def draw():
 background(0)
 if keyPressed == True and key == 'e':
 ellipse(150,150,150,150)
```

Python ▾

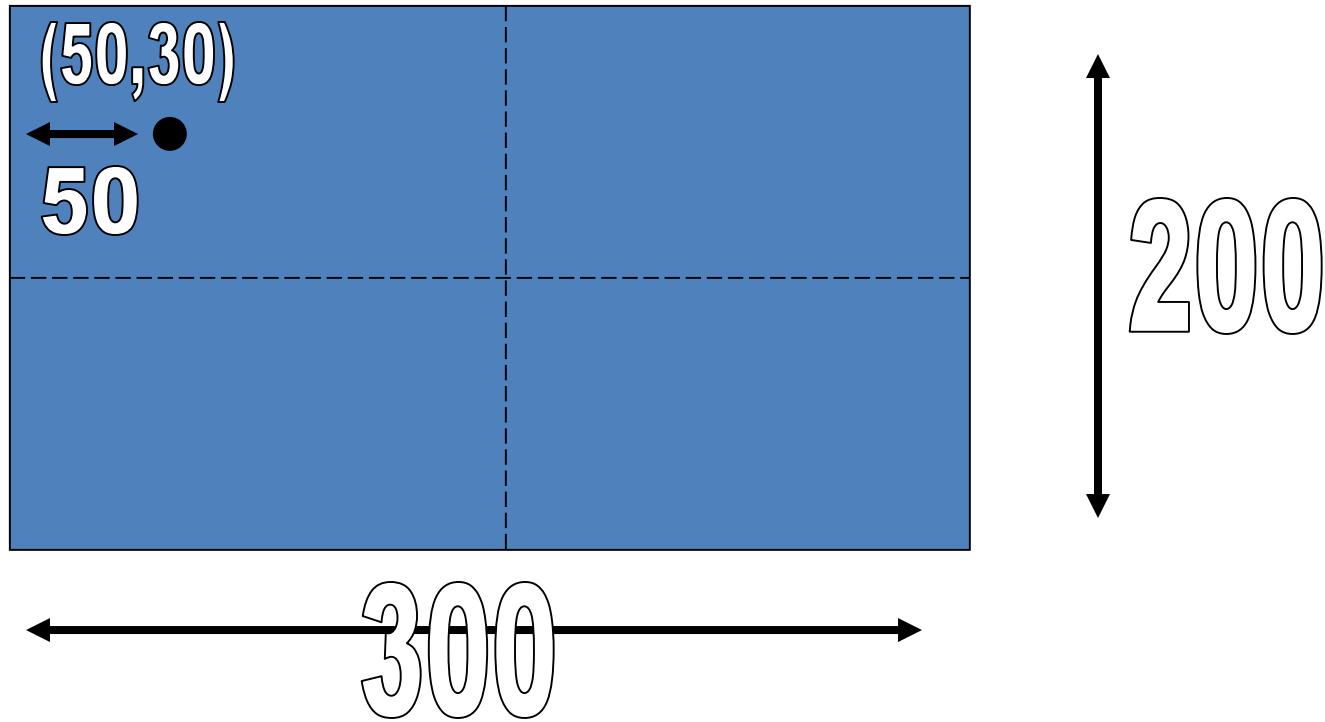
# Practice Quiz Question: What is the output?

```
num = 4
dNum = 7.2
if num > 5 and dNum < 8:
 print("first")
if num > 5 or dNum < 8:
 print("second")
if not num > 5:
 print("third")
if not dNum < 8:
 print("fourth")
```

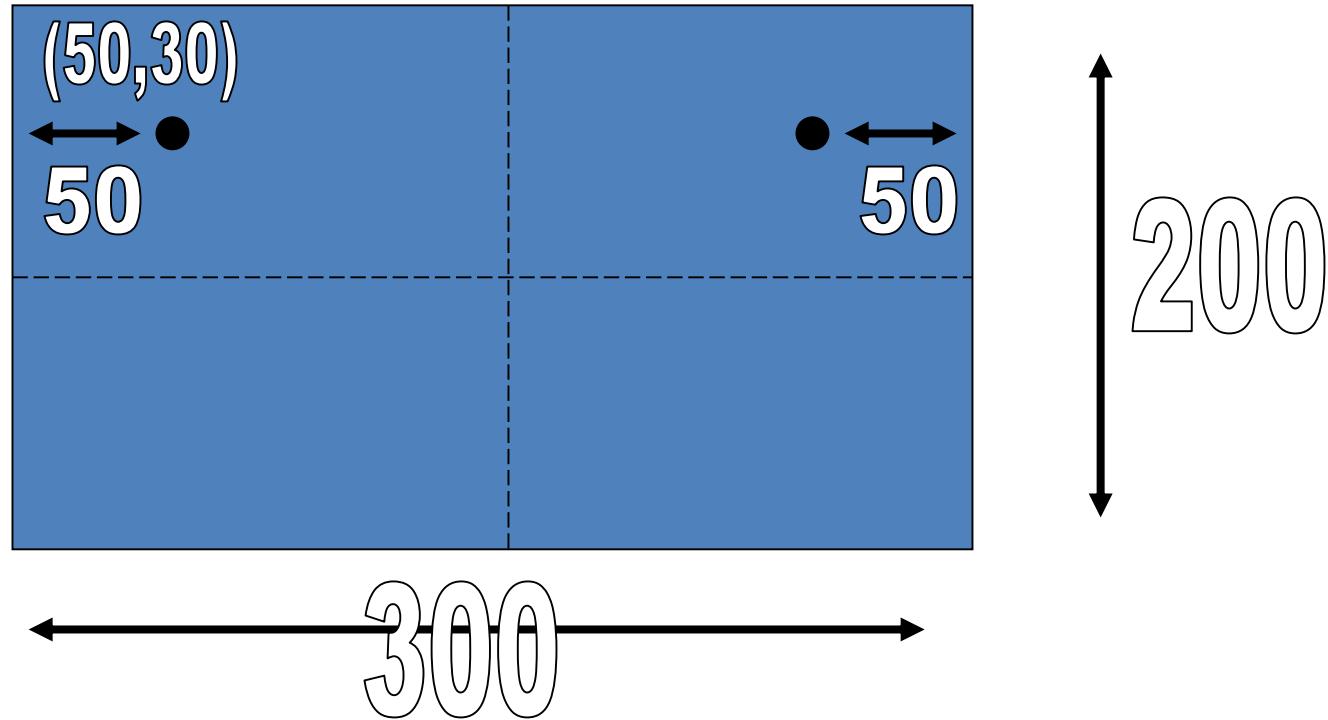
# Symmetrical Reflections (Mirrors)



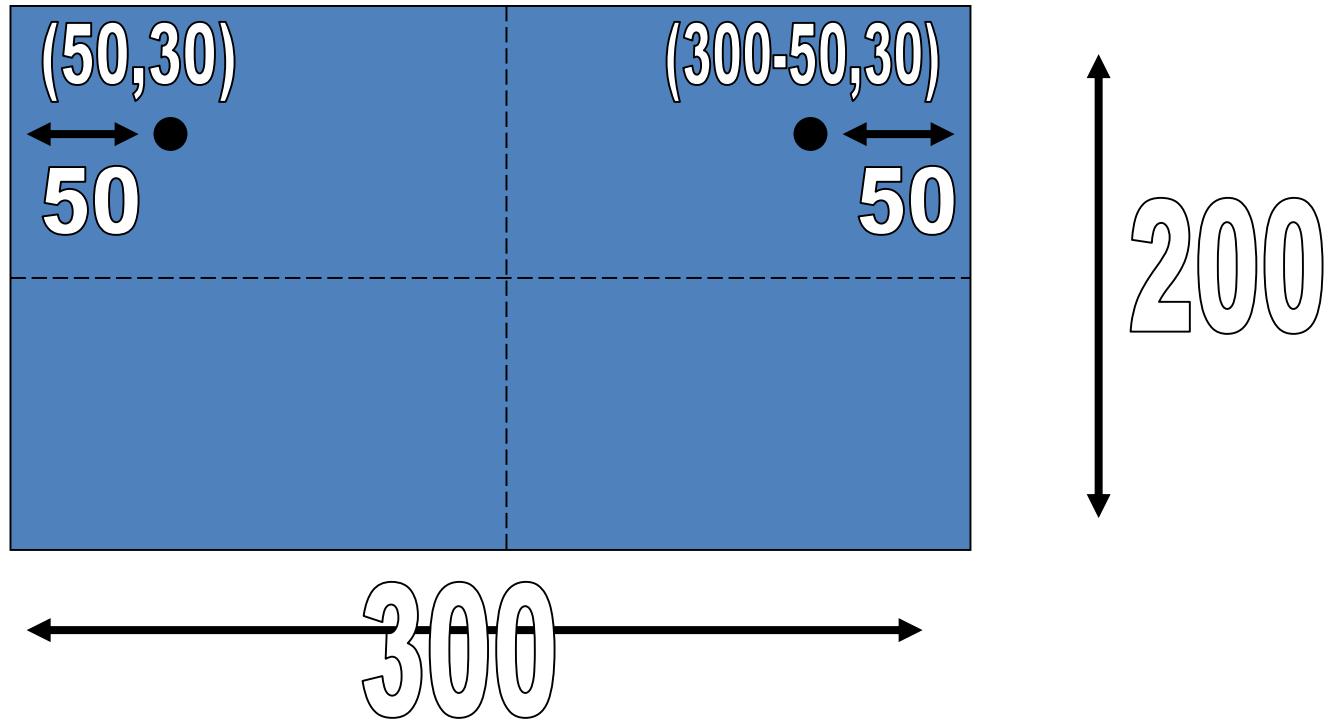
# Symmetrical Reflections (Mirrors)



# Symmetrical Reflections (Mirrors)



# Symmetrical Reflections (Mirrors)



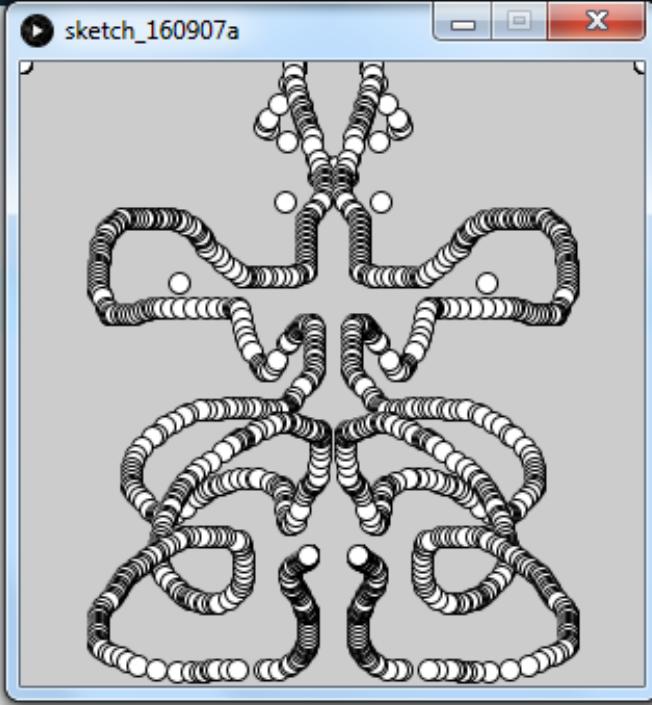
# Symmetrical Reflections (Mirrors)

sketch\_160907a | Processing 3.1.2

File Edit Sketch Tools Help

sketch\_160907a

```
def setup():
 size(300,300)
def draw():
 ellipse(mouseX,mouseY,10,10)
 ellipse(300-mouseX,mouseY,10,10)
```



# Expressions vs. Literals

These two function calls produce different output.

```
print("2 + 3")
```

```
print(2 + 3)
```

# Expressions vs. Literals

These two function calls produce different output.

```
print("2 + 3")
//displays 2 + 3

print(2 + 3)
//displays 5
```

# **print()** vs. **print()**,

- **print()** means "go to the next line AFTER you print."
- **print()**, doesn't

```
print("x"),
print(2)
//displays: x2
print("x")
print(2)
//displays: x
// 2
```

# Practice with `print()`

What's the output?

```
x = 2
```

```
print("x") ,
```

```
print(x) ,
```

# Practice with `print()`

Now, what's the output?

```
x = 2
print("x")
print(x)
```

# Practice with `print()`

Here's another one:

```
x = 2
y = 3
print("x")
print(x)
print("y")
print(y)
```

# Practice with `print()`

Notice some are `print()` and some are  
`print()`,

`x = 2`

`y = 3`

`print ("x") ,`

`print(x)`

`print ("y") ,`

`print(y)`

# Practice with `print()`

Here the order of `print()` and `print()`, are reversed

```
x = 2
```

```
y = 3
```

```
print ("x")
```

```
print(x),
```

```
print ("y")
```

```
print(y),
```

# Practice Quiz Question

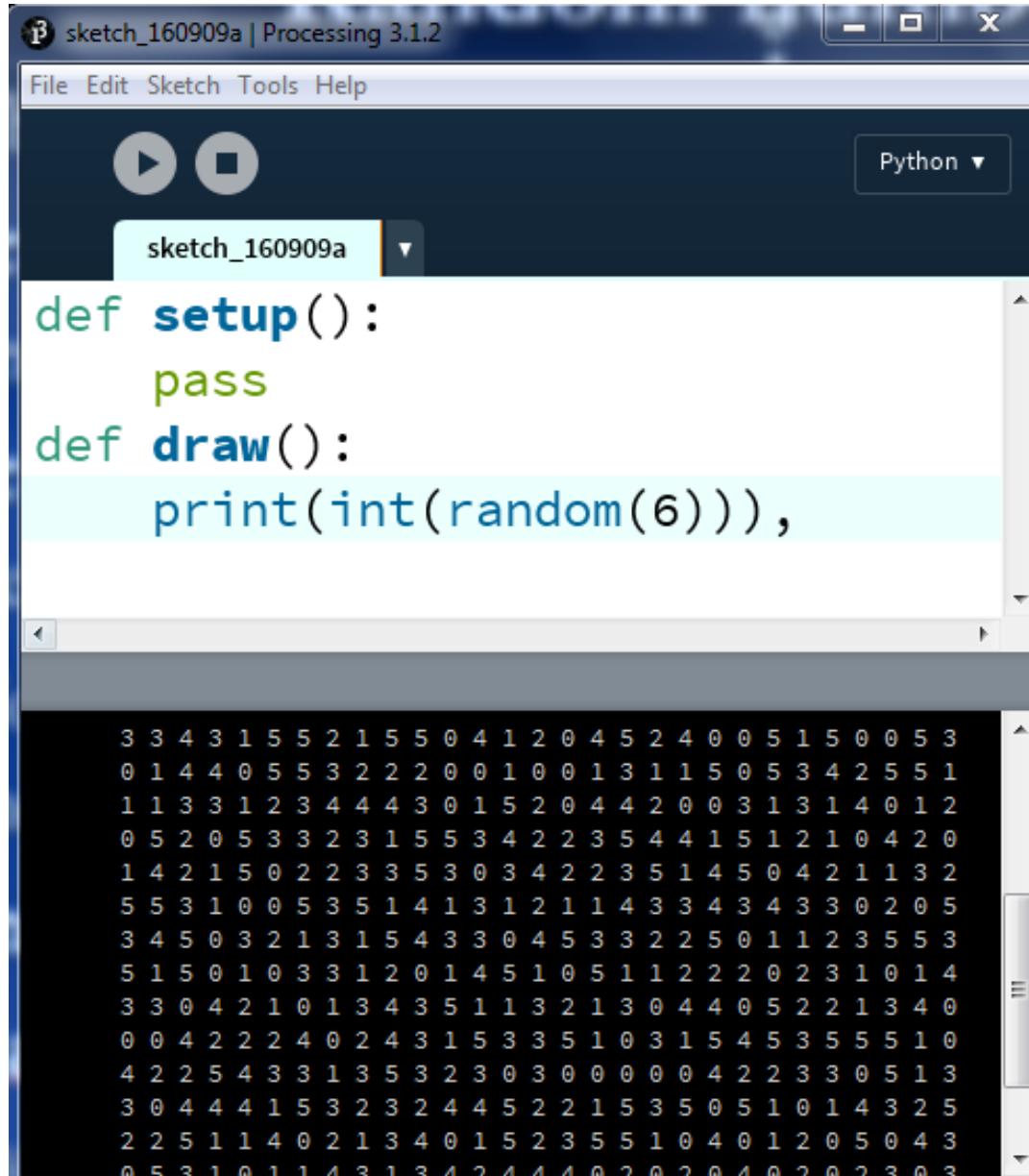
What is the output of the following program?

```
x = 13 / 5
y = 13 % 5
print("x") ,
print(x) ,
print("y")
print(y) ,
```

# Random Numbers

- The `random()` function has two versions:
- `random(5)` returns a decimal value between 0 and up to but not including 5.0
- `random(-5, 10.2)` returns a decimal value between -5.0 and up to but not including 10.2.
- To drop the decimal part of a random number, use the `int()` function.
- `int(random(6))` will generate a random integer from 0 up to but not including 6:  
`{0,1,2,3,4,5}`

# `int(random(6))`



The screenshot shows the Processing 3.1.2 software interface with a sketch titled "sketch\_160909a". The code in the editor is:

```
def setup():
 pass
def draw():
 print(int(random(6))),
```

The output window below displays a continuous stream of random integers from 0 to 5, separated by commas. A portion of the output is shown below:

```
3 3 4 3 1 5 5 2 1 5 5 0 4 1 2 0 4 5 2 4 0 0 5 1 5 0 0 5 3
0 1 4 4 0 5 5 3 2 2 2 0 0 1 0 0 1 3 1 1 5 0 5 3 4 2 5 5 1
1 1 3 3 1 2 3 4 4 4 3 0 1 5 2 0 4 4 2 0 0 3 1 3 1 4 0 1 2
0 5 2 0 5 3 3 2 3 1 5 5 3 4 2 2 3 5 4 4 1 5 1 2 1 0 4 2 0
1 4 2 1 5 0 2 2 3 3 5 3 0 3 4 2 2 3 5 1 4 5 0 4 2 1 1 3 2
5 5 3 1 0 0 5 3 5 1 4 1 3 1 2 1 1 4 3 3 4 3 4 3 3 0 2 0 5
3 4 5 0 3 2 1 3 1 5 4 3 3 0 4 5 3 3 2 2 5 0 1 1 2 3 5 5 3
5 1 5 0 1 0 3 3 1 2 0 1 4 5 1 0 5 1 1 2 2 2 0 2 3 1 0 1 4
3 3 0 4 2 1 0 1 3 4 3 5 1 1 3 2 1 3 0 4 4 0 5 2 2 1 3 4 0
0 0 4 2 2 2 4 0 2 4 3 1 5 3 3 5 1 0 3 1 5 4 5 3 5 5 5 1 0
4 2 2 5 4 3 3 1 3 5 3 2 3 0 3 0 0 0 0 0 4 2 2 3 3 0 5 1 3
3 0 4 4 4 1 5 3 2 3 2 4 4 5 2 2 1 5 3 5 0 5 1 0 1 4 3 2 5
2 2 5 1 1 4 0 2 1 3 4 0 1 5 2 3 5 5 1 0 4 0 1 2 0 5 0 4 3
0 5 3 1 0 1 1 4 3 1 3 4 3 4 4 4 0 2 0 2 0 4 0 2 0 2 3 0 3
```

# Dice Example

- Let's say I wanted to make a program that simulated rolling a single six sided die
- What numbers would be possible?

# Dice Example

- Let's say I wanted to make an applet that simulated rolling a single six sided die
- What numbers would be possible?

$\{1, 2, 3, 4, 5, 6\}$

# Dice Example

- Let's say I wanted to make an applet that simulated rolling a single six sided die.
- What numbers would be possible?

$\{1, 2, 3, 4, 5, 6\}$

**roll = ??**

# Dice Example

- Let's say I wanted to make an applet that simulated rolling a single six sided die.
- What numbers would be possible?

{1, 2, 3, 4, 5, 6}

roll = int(random( ?, ? ))

# Dice Example

- Let's say I wanted to make an applet that simulated rolling a single six sided die.
- What numbers would be possible?

The diagram illustrates the process of generating a random dice roll. At the top, a set of numbers  $\{1, 2, 3, 4, 5, 6\}$  is shown in yellow. A red arrow points from this set down to a red arrow pointing to the right. Above this second red arrow is the text  $+1$ . Below the red arrow pointing right is the Python code `roll = int(random( ?, ? ))`. The question marks in the code are highlighted in yellow, corresponding to the numbers in the set above. This visualizes how the random function generates a number between 0 and 6, and then adds 1 to get a number between 1 and 6.

```
roll = int(random(?, ?))
```

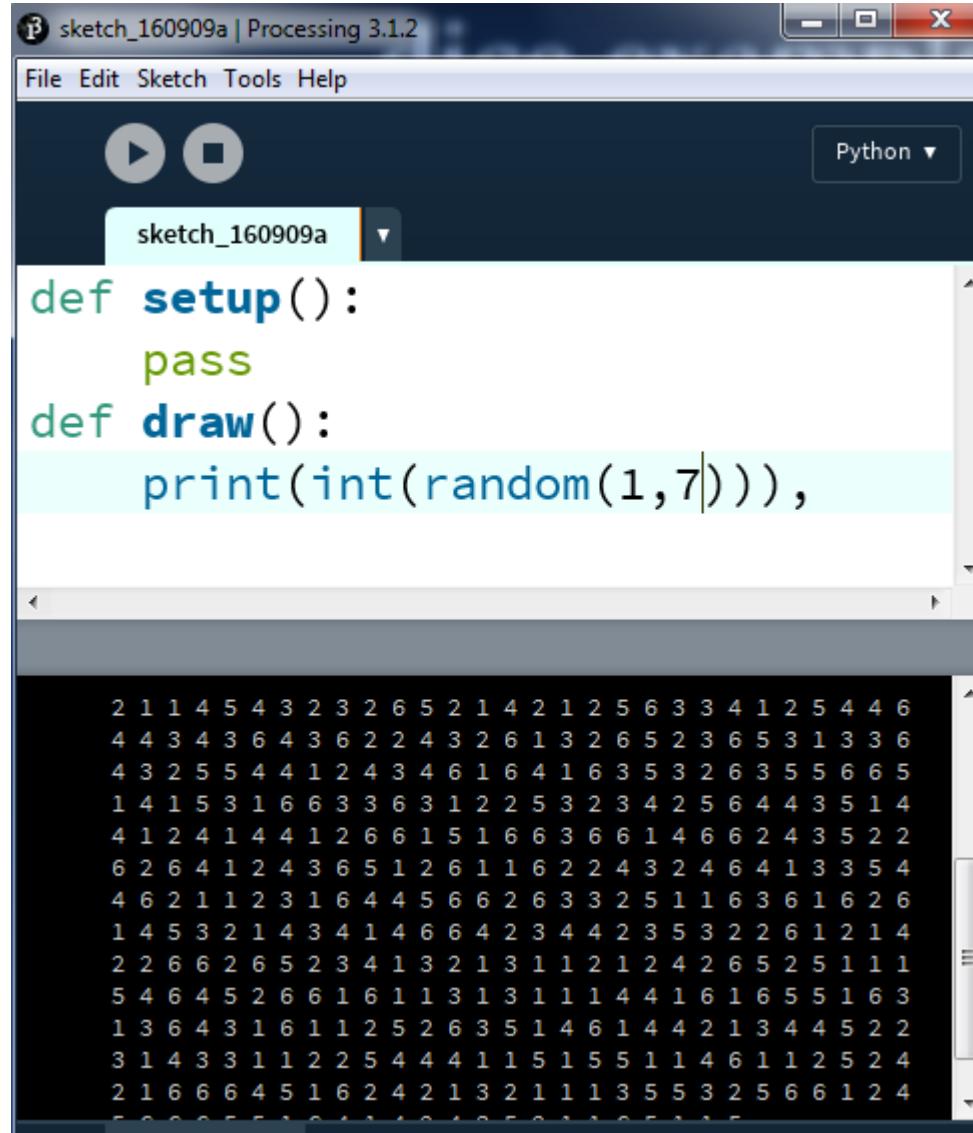
# Dice Example

- Let's say I wanted to make an applet that simulated rolling a single six sided die
- What numbers would be possible?

The diagram illustrates the process of generating a random integer between 1 and 7. At the top, a yellow curly brace encloses the integers 1, 2, 3, 4, 5, and 6. A red arrow points from this set down to a yellow curly brace enclosing the integers 1 and 7. Another red arrow points from the number 1 to the character '1' in the code below. The code itself is written in a stylized font and reads:

```
roll = int(random(1,7))
```

# Dice Example



The screenshot shows the Processing 3.1.2 software interface with a sketch titled "sketch\_160909a". The code defines a setup function that does nothing, and a draw function that prints a random integer between 1 and 7. The output window displays a large number of random integers, mostly between 1 and 6, with some 7s interspersed.

```
def setup():
 pass
def draw():
 print(int(random(1,7))),
```

|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 2 | 1 | 1 | 4 | 5 | 4 | 3 | 2 | 3 | 2 | 6 | 5 | 2 | 1 | 4 | 2 | 1 | 2 | 5 | 6 | 3 | 3 | 4 | 1 | 2 | 5 | 4 | 4 | 6 |   |
| 4 | 4 | 3 | 4 | 3 | 6 | 4 | 3 | 6 | 2 | 2 | 4 | 3 | 2 | 6 | 1 | 3 | 2 | 6 | 5 | 2 | 3 | 6 | 5 | 3 | 1 | 3 | 3 | 6 |   |
| 4 | 3 | 2 | 5 | 5 | 4 | 4 | 1 | 2 | 4 | 3 | 4 | 6 | 1 | 6 | 4 | 1 | 6 | 3 | 5 | 3 | 2 | 6 | 3 | 5 | 5 | 6 | 6 | 5 |   |
| 1 | 4 | 1 | 5 | 3 | 1 | 6 | 6 | 3 | 3 | 6 | 3 | 1 | 2 | 2 | 5 | 3 | 2 | 3 | 4 | 2 | 5 | 6 | 4 | 4 | 3 | 5 | 1 | 4 |   |
| 4 | 1 | 2 | 4 | 1 | 4 | 4 | 1 | 2 | 6 | 6 | 1 | 5 | 1 | 6 | 6 | 3 | 6 | 6 | 1 | 4 | 6 | 6 | 2 | 4 | 3 | 5 | 2 | 2 |   |
| 6 | 2 | 6 | 4 | 1 | 2 | 4 | 3 | 6 | 5 | 1 | 2 | 6 | 1 | 1 | 6 | 2 | 2 | 4 | 3 | 2 | 4 | 6 | 4 | 1 | 3 | 3 | 5 | 4 |   |
| 4 | 6 | 2 | 1 | 1 | 2 | 3 | 1 | 6 | 4 | 4 | 5 | 6 | 6 | 2 | 6 | 3 | 3 | 2 | 5 | 1 | 1 | 6 | 3 | 6 | 1 | 6 | 2 | 6 |   |
| 1 | 4 | 5 | 3 | 2 | 1 | 4 | 3 | 4 | 1 | 4 | 6 | 6 | 4 | 2 | 3 | 4 | 4 | 2 | 3 | 5 | 3 | 2 | 2 | 6 | 1 | 2 | 1 | 4 |   |
| 2 | 2 | 6 | 6 | 2 | 6 | 5 | 2 | 3 | 4 | 1 | 3 | 2 | 1 | 3 | 1 | 1 | 2 | 1 | 2 | 4 | 2 | 6 | 5 | 2 | 5 | 1 | 1 | 1 |   |
| 5 | 4 | 6 | 4 | 5 | 2 | 6 | 6 | 1 | 6 | 1 | 1 | 3 | 1 | 3 | 1 | 1 | 1 | 4 | 4 | 1 | 6 | 1 | 6 | 5 | 5 | 1 | 6 | 3 |   |
| 1 | 3 | 6 | 4 | 3 | 1 | 6 | 1 | 1 | 2 | 5 | 2 | 6 | 3 | 5 | 1 | 4 | 6 | 1 | 4 | 4 | 2 | 1 | 3 | 4 | 4 | 5 | 2 | 2 |   |
| 3 | 1 | 4 | 3 | 3 | 1 | 1 | 2 | 2 | 5 | 4 | 4 | 1 | 1 | 5 | 1 | 5 | 5 | 1 | 1 | 4 | 6 | 1 | 1 | 2 | 5 | 2 | 4 |   |   |
| 2 | 1 | 6 | 6 | 6 | 4 | 5 | 1 | 6 | 2 | 4 | 2 | 1 | 3 | 2 | 1 | 1 | 1 | 3 | 5 | 5 | 3 | 2 | 5 | 6 | 6 | 1 | 2 | 4 |   |
| 5 | 0 | 0 | 0 | 5 | 5 | 1 | 0 | 1 | 1 | 0 | 1 | 0 | 5 | 0 | 1 | 1 | 0 | 5 | 1 | 1 | 5 | 0 | 0 | 0 | 5 | 5 | 1 | 1 | 5 |

# A Bug with Negative Arguments

- What arguments would you use to get this range?

$$\{-3, -2, -1, 0, 1, 2, 3\}$$

```
roll = int(random(?, ?))
```

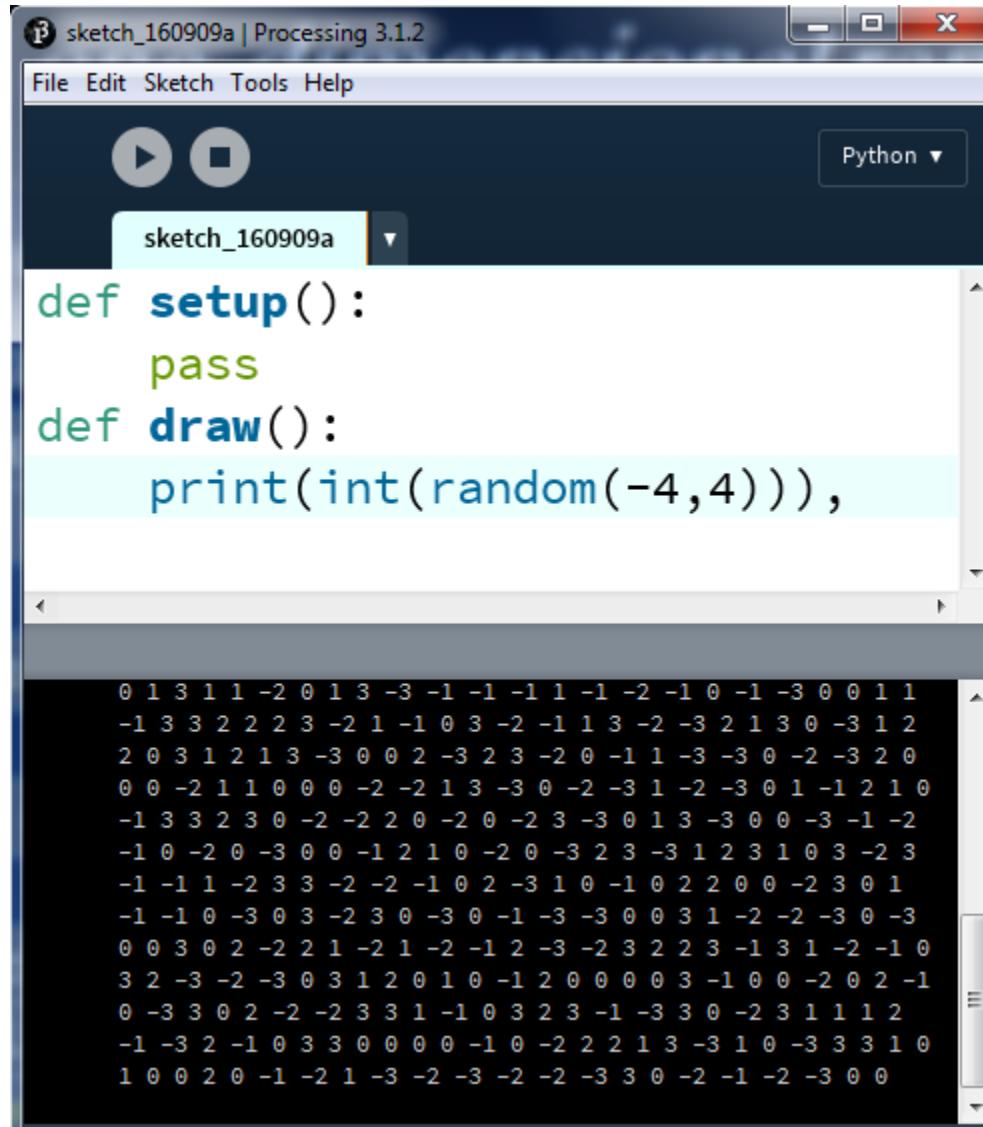
# A Bug with Negative Arguments

- For some reason the low limit is one less.

The diagram illustrates a bug in a programming code. At the top, a set of integers is shown in yellow:  $\{-3, -2, -1, 0, 1, 2, 3\}$ . Two red arrows point downwards from this set to a code snippet below. The left arrow points to the value  $-1$ , and the right arrow points to the value  $+1$ . Below the set, the code is written in black: `roll = int(random(-4, 4))`. In this code, the argument  $-4$  is highlighted in red, corresponding to the  $-1$  in the set above it. This visual cue highlights the discrepancy between the intended range and the actual range generated by the code.

```
roll = int(random(-4, 4))
```

# A Bug with Negative Arguments



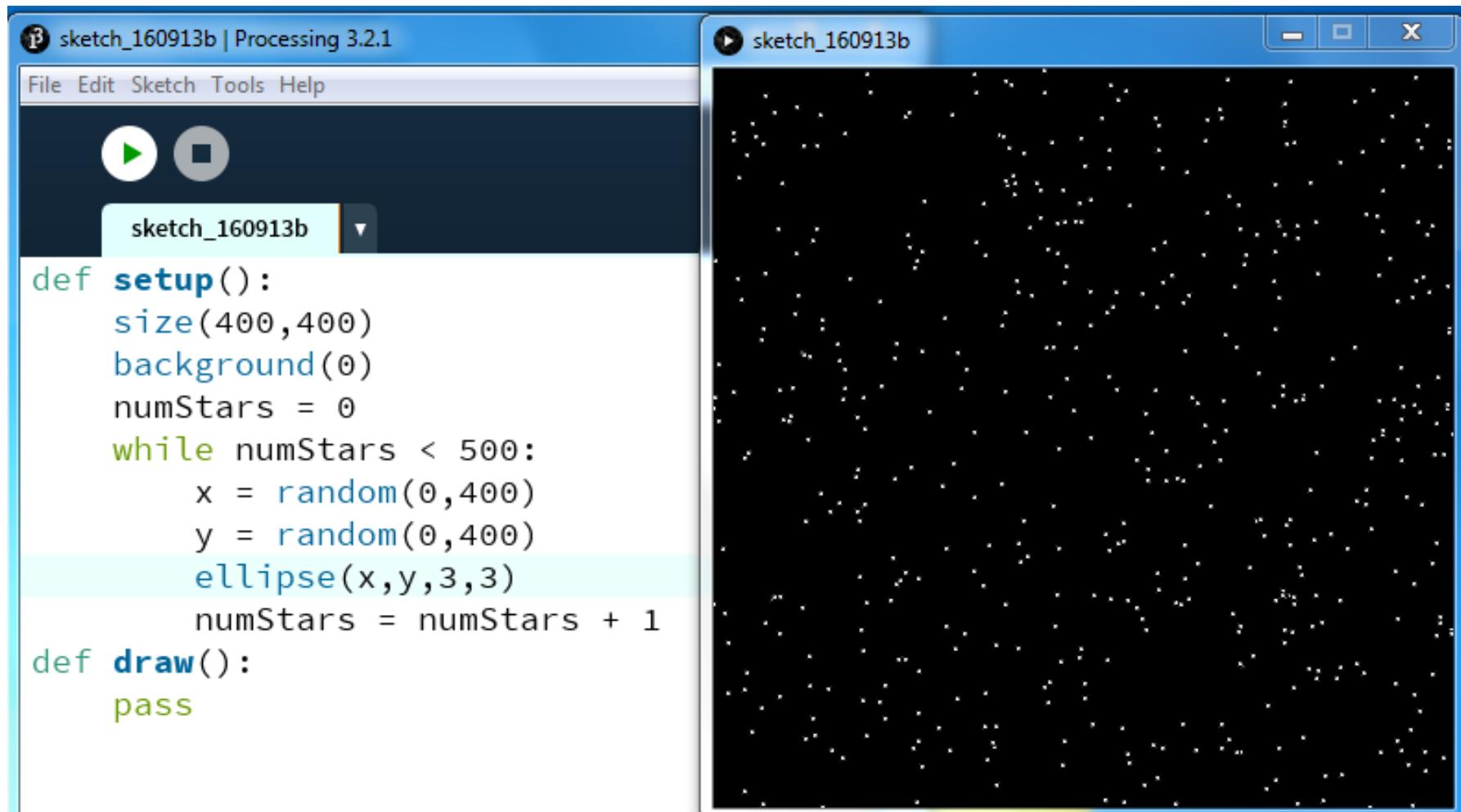
The screenshot shows the Processing 3.1.2 software interface with a sketch titled "sketch\_160909a". The code in the editor is as follows:

```
def setup():
 pass
def draw():
 print(int(random(-4,4))),
```

The output window displays a series of random integers ranging from -4 to 4, with each value followed by a comma. The output is as follows:

```
0 1 3 1 1 -2 0 1 3 -3 -1 -1 -1 1 -1 -2 -1 0 -1 -3 0 0 1 1
-1 3 3 2 2 2 3 -2 1 -1 0 3 -2 -1 1 3 -2 -3 2 1 3 0 -3 1 2
2 0 3 1 2 1 3 -3 0 0 2 -3 2 3 -2 0 -1 1 -3 -3 0 -2 -3 2 0
0 0 -2 1 1 0 0 0 -2 -2 1 3 -3 0 -2 -3 1 -2 -3 0 1 -1 2 1 0
-1 3 3 2 3 0 -2 -2 2 0 -2 0 -2 3 -3 0 1 3 -3 0 0 -3 -1 -2
-1 0 -2 0 -3 0 0 -1 2 1 0 -2 0 -3 2 3 -3 1 2 3 1 0 3 -2 3
-1 -1 1 -2 3 3 -2 -2 -1 0 2 -3 1 0 -1 0 2 2 0 0 -2 3 0 1
-1 -1 0 -3 0 3 -2 3 0 -3 0 -1 -3 -3 0 0 3 1 -2 -2 -3 0 -3
0 0 3 0 2 -2 2 1 -2 1 -2 -1 2 -3 -2 3 2 2 3 -1 3 1 -2 -1 0
3 2 -3 -2 -3 0 3 1 2 0 1 0 -1 2 0 0 0 0 3 -1 0 0 -2 0 2 -1
0 -3 3 0 2 -2 -2 3 3 1 -1 0 3 2 3 -1 -3 3 0 -2 3 1 1 1 2
-1 -3 2 -1 0 3 3 0 0 0 0 -1 0 -2 2 2 1 3 -3 1 0 -3 3 3 1 0
1 0 0 2 0 -1 -2 1 -3 -2 -3 -2 -2 -3 3 0 -2 -1 -2 -3 0 0
```

# A random() starry sky

A screenshot of the Processing 3.2.1 software interface. The left window shows the code for a sketch named "sketch\_160913b". The code uses the "random()" function to generate a starry sky. The right window shows the resulting visual output, which is a dark blue square filled with numerous small white dots of varying sizes, representing stars.

```
sketch_160913b | Processing 3.2.1
File Edit Sketch Tools Help
sketch_160913b
def setup():
 size(400,400)
 background(0)
 numStars = 0
 while numStars < 500:
 x = random(0,400)
 y = random(0,400)
 ellipse(x,y,3,3)
 numStars = numStars + 1
def draw():
 pass
```

# Practice Quiz Questions

Fill in the blanks to create the indicated random numbers.

A random integer from 0 to 10: (including 10)

**int(random(\_\_\_\_\_))**

A random integer from 1 to 100: (including 100)

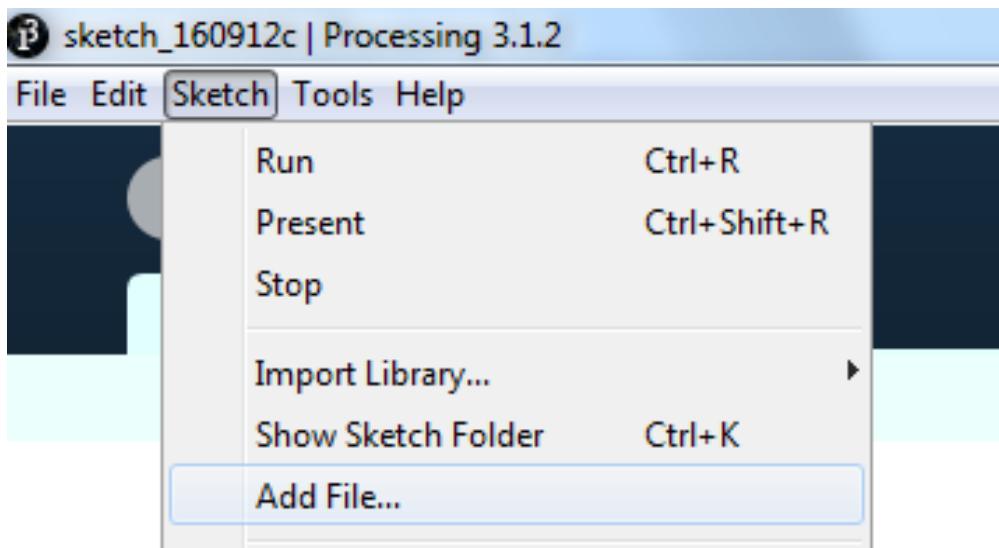
**int(random(\_\_\_\_\_))**

A random integer from -5 to 5: (including -5 & 5)

**int(random(\_\_\_\_\_))**

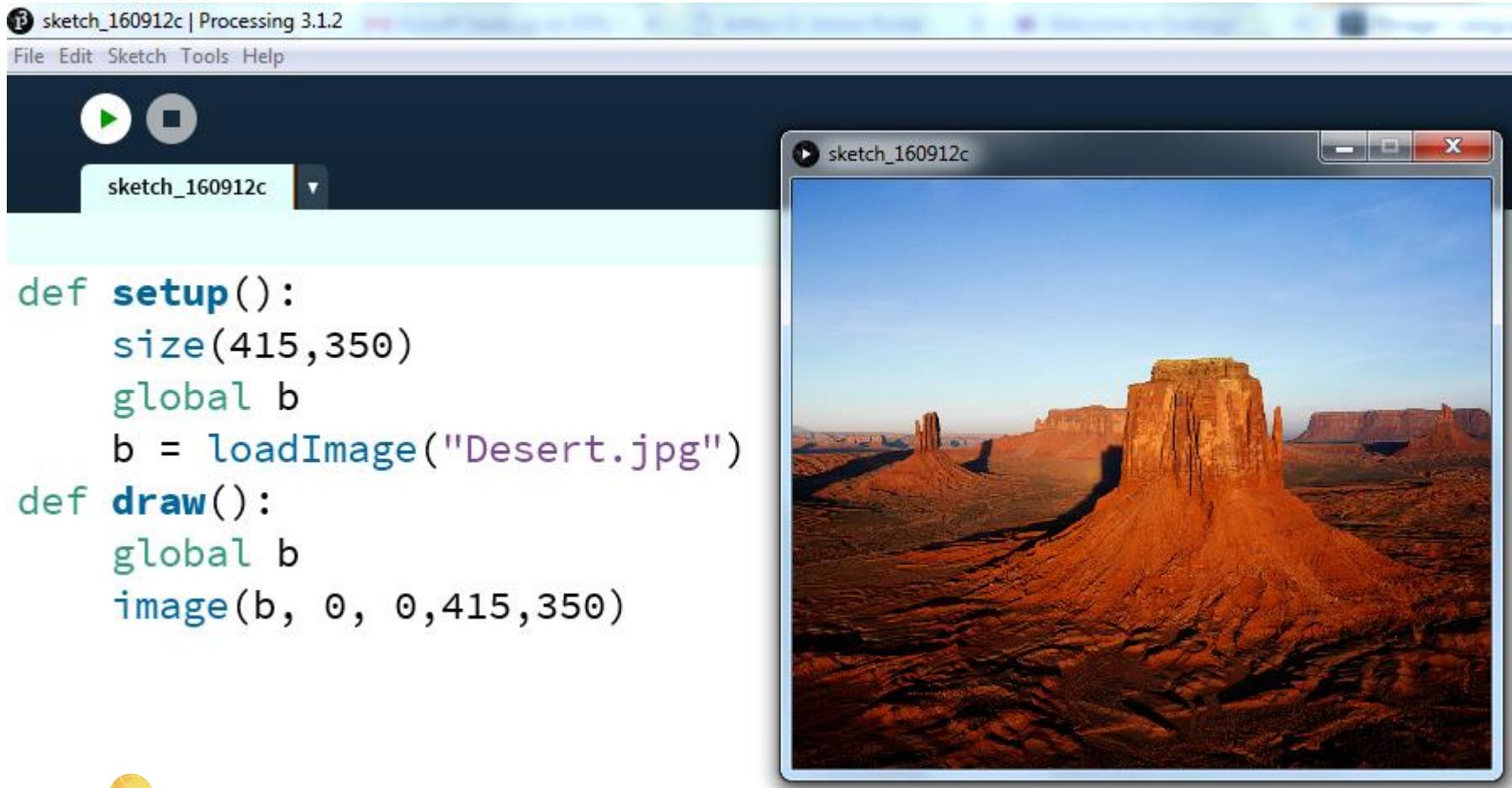
# Using Pictures and Images

- Processing can display .gif, .jpg, .tga, and .png image.
- First, choose ***Sketch*** / ***Add File*** to select the the image you want.



# Images

Now, write code like this, where **Desert.jpg** is the name of the picture.



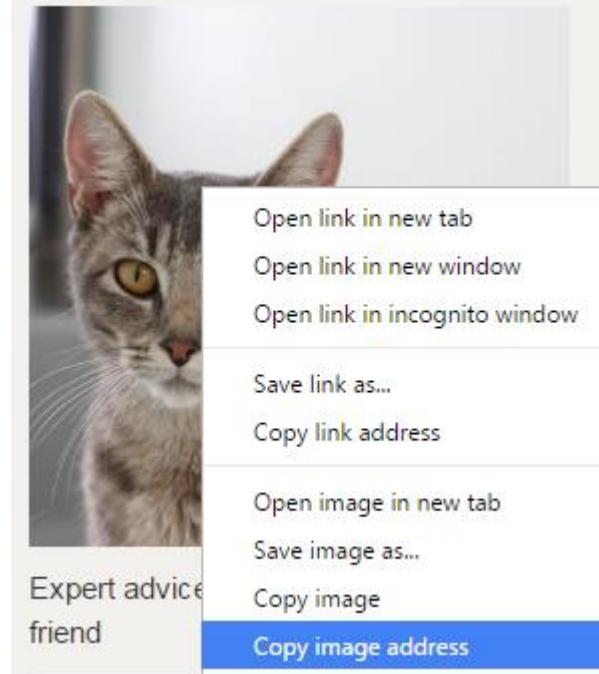
The image shows the Processing 3.1.2 software interface. The title bar says "sketch\_160912c | Processing 3.1.2". The menu bar includes File, Edit, Sketch, Tools, and Help. On the left, there are play and stop buttons, and a dropdown menu set to "sketch\_160912c". The main area contains the following code:

```
def setup():
 size(415,350)
 global b
 b = loadImage("Desert.jpg")
def draw():
 global b
 image(b, 0, 0,415,350)
```

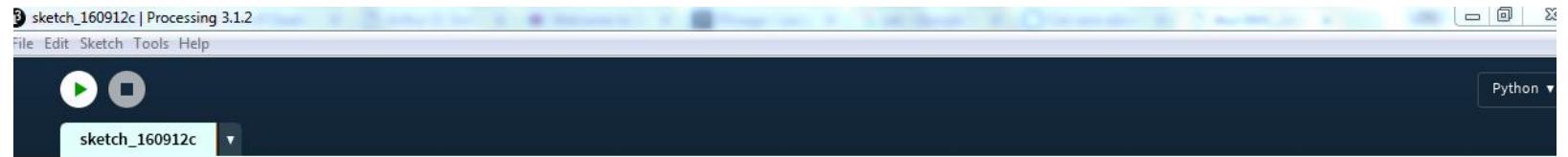
The right side of the image shows the resulting window titled "sketch\_160912c" displaying a desert landscape with large rock formations under a blue sky.

# Using a URL for an Image

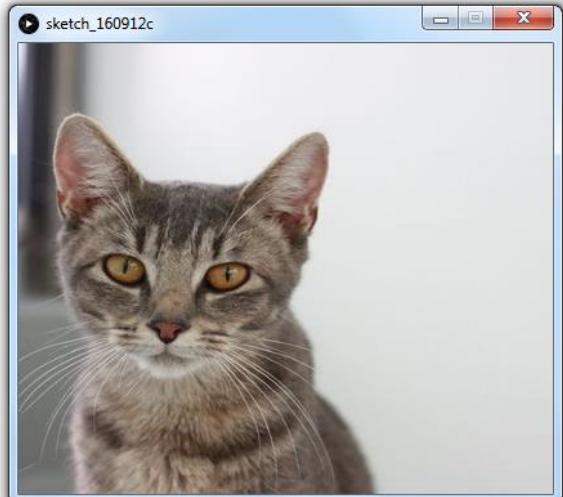
- ❑ Another way to load an image into your program is with a URL
- ❑ Right click on an image that's on the internet and select *Copy image address*



# Paste the url into loadImage



```
def setup():
 size(415,350)
 global b
 b = loadImage("https://d1ra4hr810e003.cloudfront.net/media/27FB7F0C-9885-42A6-9E0C19C:")
def draw():
 global b
 image(b, 0, 0,415,350)
```



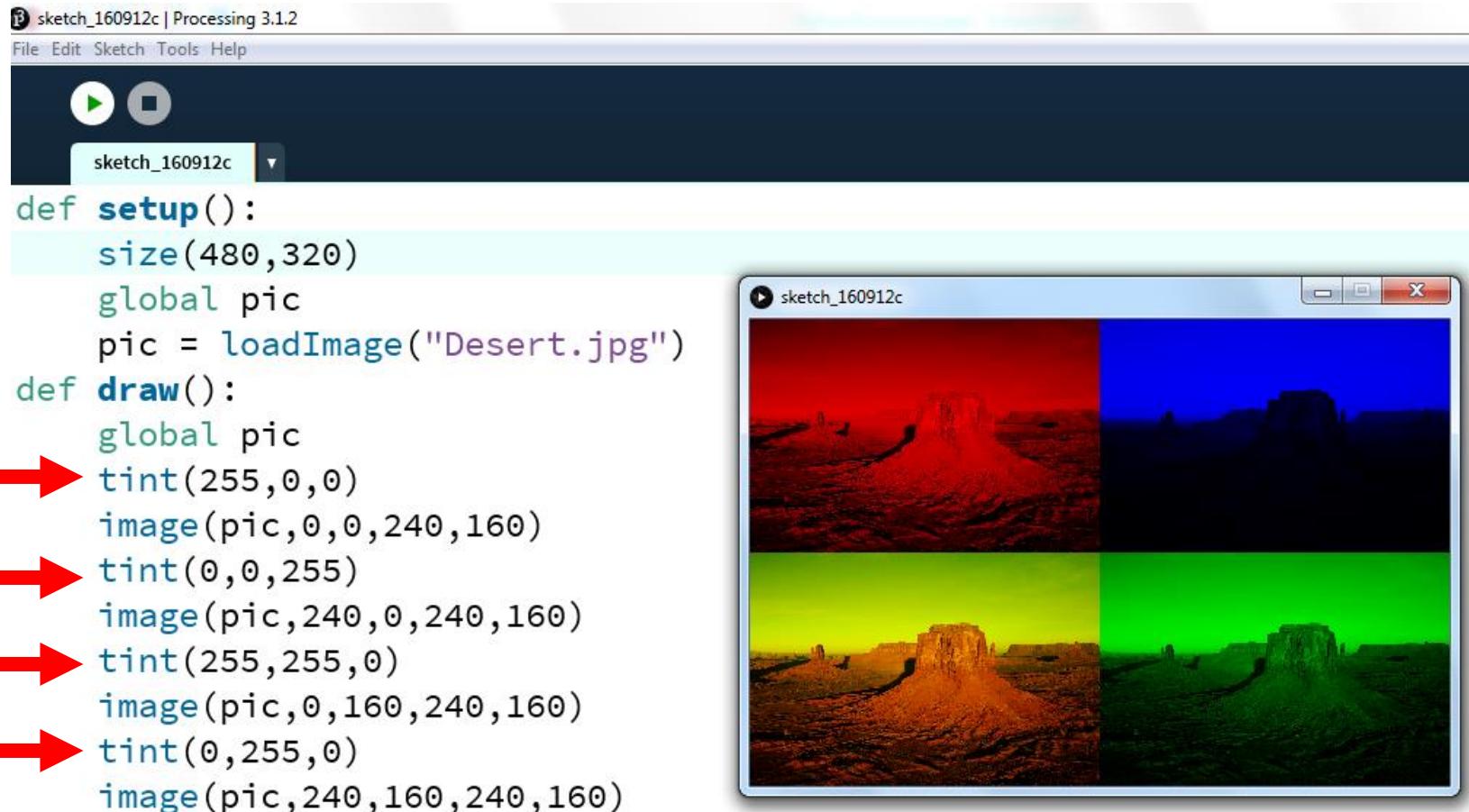
# Images

- **b** is the name of the *variable* that holds the image.
- **0,0** is the x and y of the top left hand corner.
- **415,350** is the **width** and **height**.

```
def setup():
 size(415,350)
 global b
 b = loadImage("Desert.jpg")
def draw():
 global b
 image(b, 0, 0 ,415,350)
```

# tint()

You can change the color and opacity of an image with the **tint()** function.



```
sketch_160912c | Processing 3.1.2
File Edit Sketch Tools Help
sketch_160912c

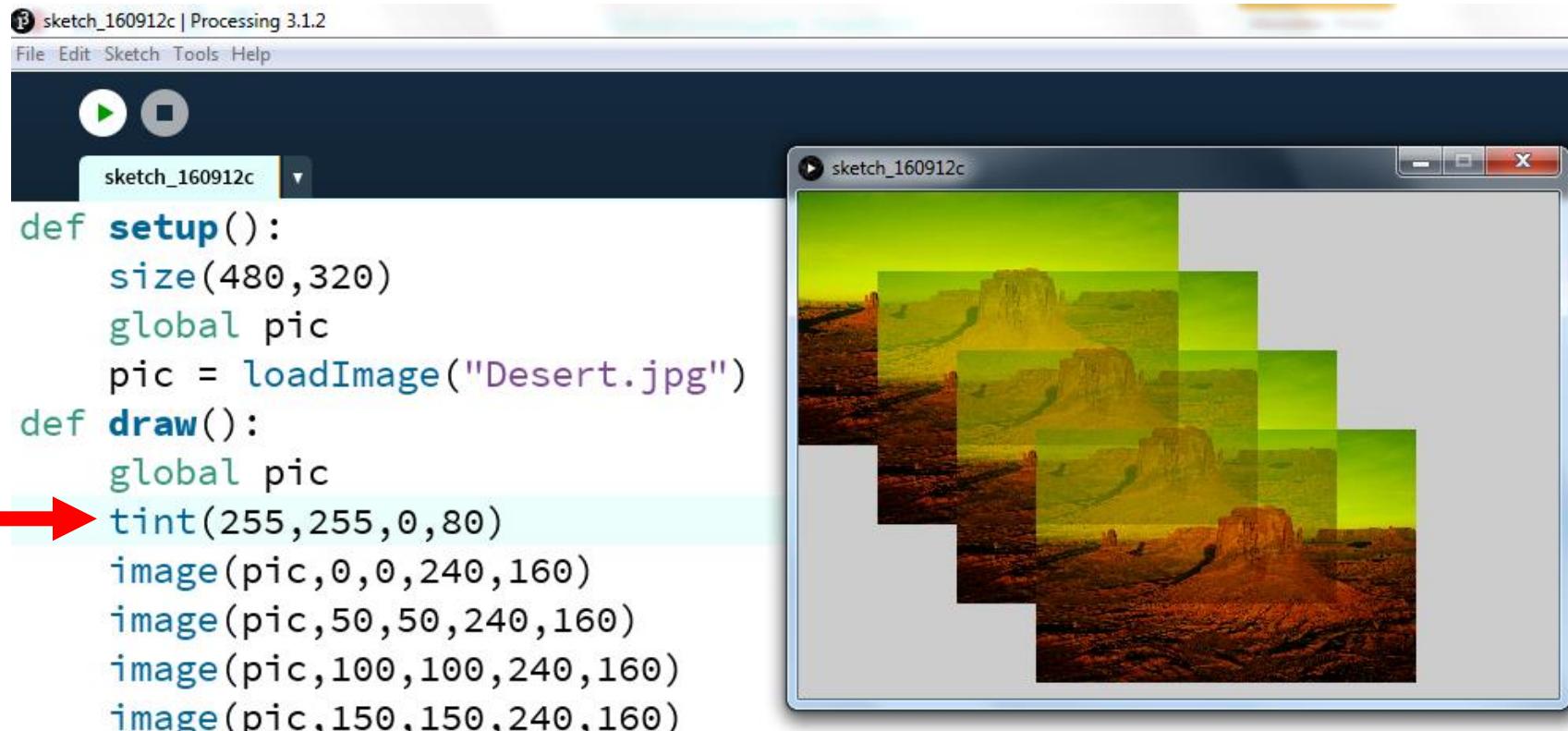
def setup():
 size(480,320)
 global pic
 pic = loadImage("Desert.jpg")
def draw():
 global pic
 tint(255,0,0)
 image(pic,0,0,240,160)
 tint(0,0,255)
 image(pic,240,0,240,160)
 tint(255,255,0)
 image(pic,0,160,240,160)
 tint(0,255,0)
 image(pic,240,160,240,160)
```

The code demonstrates the use of the `tint()` function within a `draw()` loop. It loads an image of a desert landscape and then applies different color transformations to its four quadrants. Red arrows point to the `tint()` calls at indices 2, 3, 4, and 5 in the `draw()` loop, which result in the image being rendered in red, blue, yellow, and green respectively.



# tint()

You can change the color and opacity of an image with the **tint()** function.



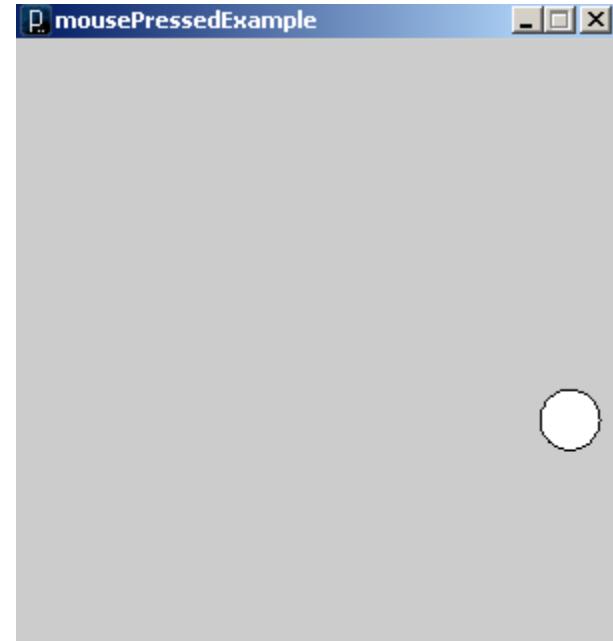
```
sketch_160912c | Processing 3.1.2
File Edit Sketch Tools Help
sketch_160912c
def setup():
 size(480,320)
 global pic
 pic = loadImage("Desert.jpg")
def draw():
 global pic
 tint(255,255,0,80)
 image(pic,0,0,240,160)
 image(pic,50,50,240,160)
 image(pic,100,100,240,160)
 image(pic,150,150,240,160)
```

# **mousePressed** function vs. variable

- Processing has both a **mousePressed()** function and a **mousePressed** system variable
- The thing to remember is that the **mousePressed()** function will run once every time the mouse is pressed.

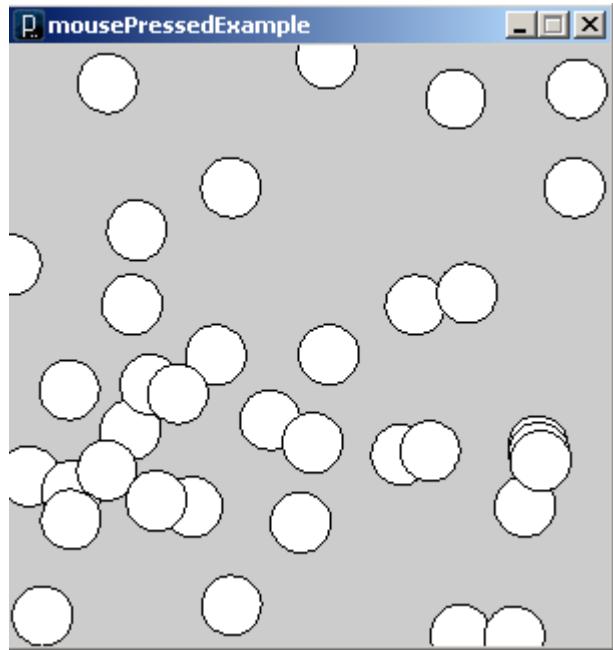
# If I use the `mousePressed` function, I get one ellipse with every press.

```
def setup():
 size(300,300)
def draw():
 pass
def mousePressed():
 ellipse(random(300),random(300),30,30)
```



If I use the `mousePressed` variable in `draw()`, I get more ellipses the longer I press.

```
def setup():
 size(300,300)
def draw():
 if mousePressed == True:
 ellipse(random(300),random(300),30,30)
```



# RANDOM on the AP Exam

- **RANDOM(a,b)** evaluates to a random integer from **a** to **b** including **a** and **b**
- For example **RANDOM(1,3)** could generate any of the values {1,2,3}

# What are the possible values for **x**?

```
x <- 0
```

```
REPEAT 3 TIMES
```

```
{
```

```
 x <- x + RANDOM(1,2)
```

```
}
```

- The smallest value would be 3.
- The largest value would be 6.
- So the possible values for x are {3, 4, 5, 6} .

```
x <- 0
REPEAT 2 TIMES
{
 x <- x + RANDOM(1,3)
}
y <- 0
REPEAT 3 TIMES
{
 y <- y + RANDOM(1,3)
}
```

# Practice Quiz

## Question

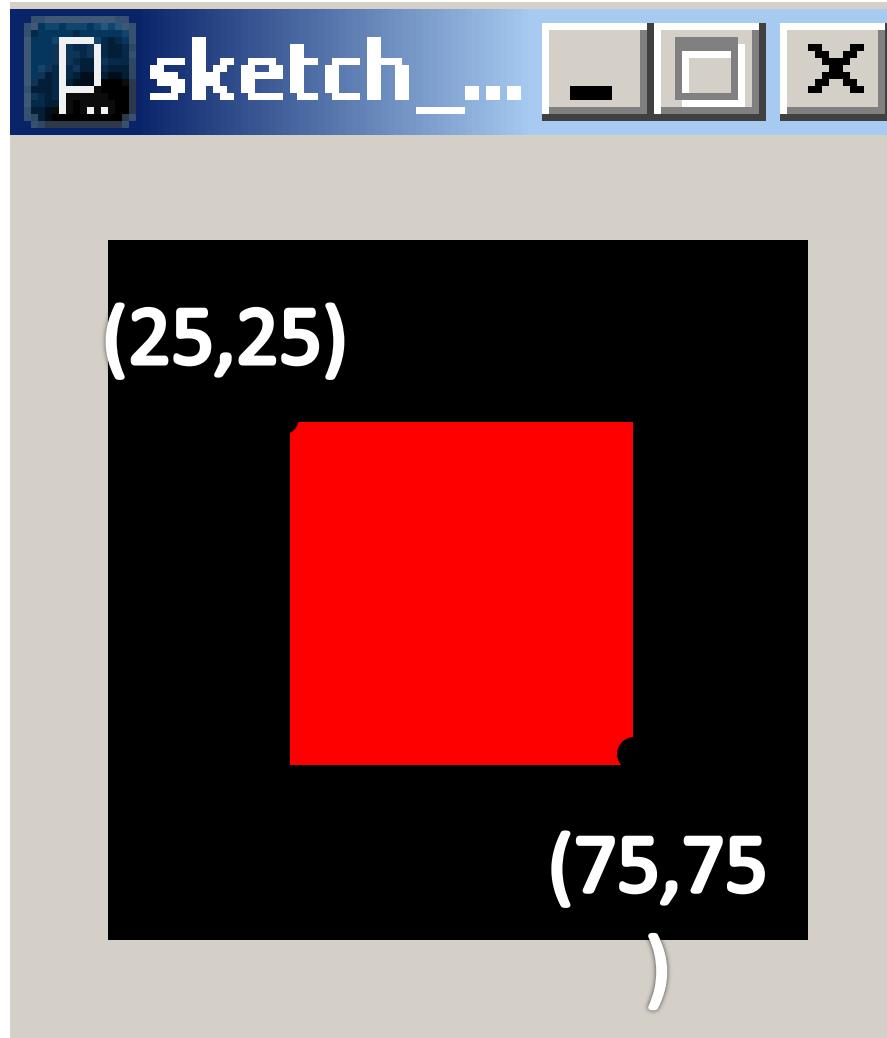
- Which of the following would NOT be a possible (x,y) coordinate pair after the program fragment executed? (Choose two)
- A. (1,1) B. (3,6) C.(4,5) D(6,3) E.(8,8)

# GUI

- GUI stands for Graphical User Interface.
- A GUI gets user input through objects like buttons, checkboxes and sliders.
- Most modern programs feature GUIs.
- It's reasonably easy to create your own GUI objects in Processing.

# Making a Button

```
rect(25,25,50,50)
```



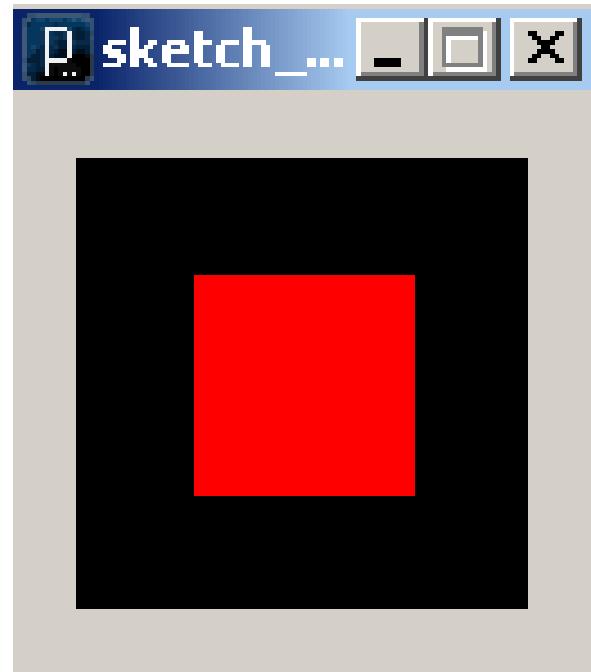
# Making a Button

```
def setup():
 noLoop()

def draw():
 background(0)
 rect(25,25,50,50)

def mousePressed():
 if mouseX >= 25 and mouseX <= 75 and \
 mouseY >= 25 and mouseY <= 75:
 fill(255,0,0)
 redraw()

def mouseReleased():
 fill(255)
 redraw()
```



# \ Breaks a Long Line of Code into Two Lines

```
def setup():
 noLoop()

def draw():
 background(0)
 rect(25,25,50,50)

def mousePressed():
 if mouseX >= 25 and mouseX <= 75 and mouseY >= 25 and mouseY <= 75:
 fill(255,0,0)
 redraw()

def mouseReleased():
 fill(255)
 redraw()
```



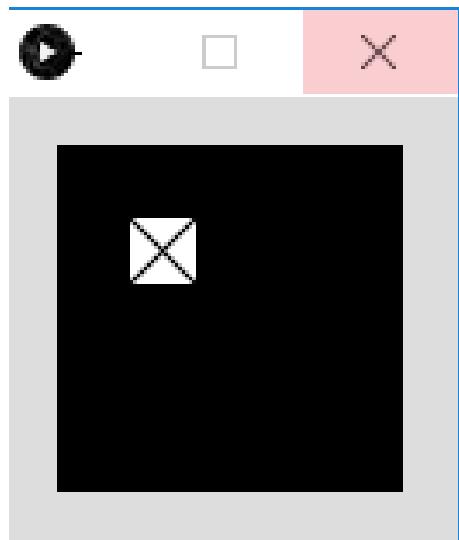
# Making a Checkbox

```
isChecked = True

def setup():
 pass # do nothing

def draw():
 global isChecked
 background(0)
 rect(20,20,20,20)
 if isChecked == True:
 line(20,20,40,40)
 line(20,40,40,20)

def mousePressed():
 global isChecked
 if mouseX >= 20 and mouseX <= 40 and \
 mouseY >= 20 and mouseY <= 40:
 isChecked = not isChecked #opposite
```

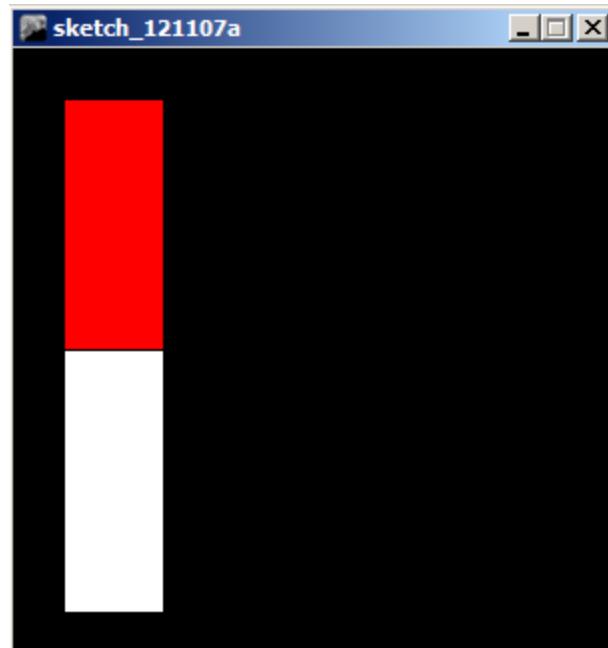


# Making a Slider

```
r = 127
def setup():
 size(300, 300)
 noLoop()

def draw():
 global r
 background(0)
 fill(255)
 rect(25, 25, 50, 256)
 fill(255, 0, 0)
 rect(25, 25, 50, r)

def mouseDragged():
 global r
 if mouseX > 25 and mouseX < 75 and \
 mouseY > 25 and mouseY < 25 + 256:
 r = mouseY-25;
 redraw()
```



# A Program that Uses a Simple **button** to Change the Fill and Stroke Colors

The image shows the Processing IDE interface. On the left is the code editor with the following pseudocode:

```
File Edit Sketch Tools Help
sketch_160914b
def setup():
 size(200,100)
 stroke(0)
 fill(0)
def draw():
 rect(10,10,10,10)
 if mousePressed == True:
 if mouseX >= 10 and mouseX <= 20 and mouseY >= 10 and mouseY <= 20:
 r = random(255)
 g = random(255)
 b = random(255)
 stroke(r,g,b)
 fill(r,g,b)
 else:
 ellipse(mouseX,mouseY,3,3)
```

On the right is the sketch window titled "sketch\_1609...". It displays a gray background with several colored lines and shapes. A small green square is at the top-left, followed by a blue wavy line, a cyan dashed line, a green dotted line, a pink solid line, and a cyan dashed line. Below these is a green rectangle. In the bottom-right corner, there is a small vertical black bar and a cyan circle.

Two yellow arrows point from the code editor to the sketch window: one points from the "rect" call in the "draw" function to the green rectangle in the sketch, and another points from the "if" condition in the "draw" function to the cyan circle in the sketch.

# if and if/else

```
num = 200
if num > 150:
 print("num is big")
```

- An **if** controls some code that either runs or doesn't.

```
num = 100
if num > 150:
 print("num is big")
else:
 print("num isn't that big")
```

- An **if/else** *always* runs the code in the **if** or the **else**, but never both.

# "Chained" elif What will be printed?

```
temp = 62
if temp > 80:
 print("Go swimming")
elif temp > 50:
 print("Go Fishing")
elif temp > 32:
 print("Go hot tubing")
else:
 print("Go sledding")
```

# "Chained" elif What will be printed now?

```
temp = 28
if temp > 80:
 print("Go swimming")
elif temp > 50:
 print("Go Fishing")
elif temp > 32:
 print("Go hot tubing")
else:
 print("Go sledding")
```

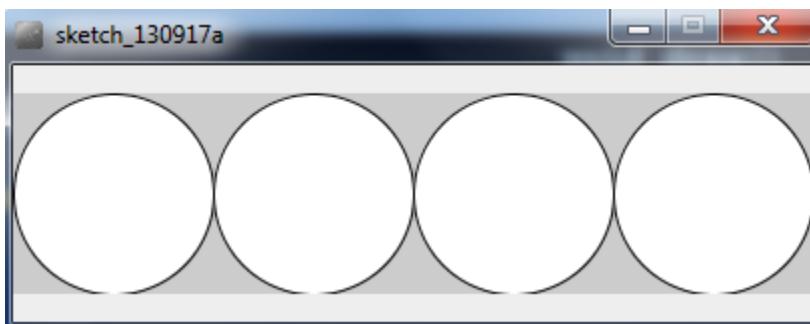
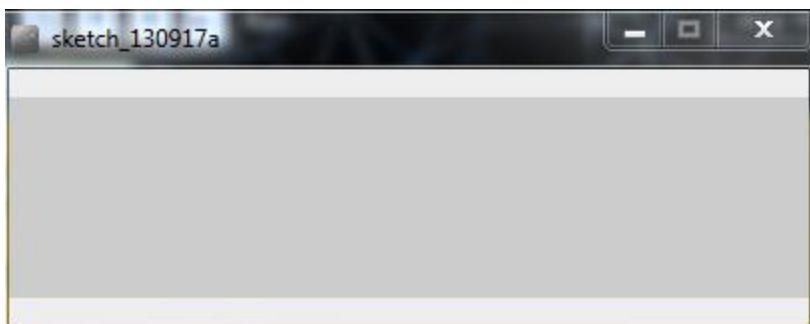
# if vs. if/else

- Here is a program that uses four separate ifs

```
def setup():
 size(400,100)
 frameRate(2)
def draw():
 background(204)
 if int(random(4)) == 0:
 ellipse(50,50,100,100)
 if int(random(4)) == 1:
 ellipse(150,50,100,100)
 if int(random(4)) == 2:
 ellipse(250,50,100,100)
 if int(random(4)) == 3:
 ellipse(350,50,100,100)
```

# if vs. if/else

Every time the screen is drawn I could see any number of ellipses from 0 to 4.



```
def setup():
 size(400,100)
 frameRate(2)
def draw():
 background(204)
 if int(random(4)) == 0:
 ellipse(50,50,100,100)
 if int(random(4)) == 1:
 ellipse(150,50,100,100)
 if int(random(4)) == 2:
 ellipse(250,50,100,100)
 if int(random(4)) == 3:
 ellipse(350,50,100,100)
```

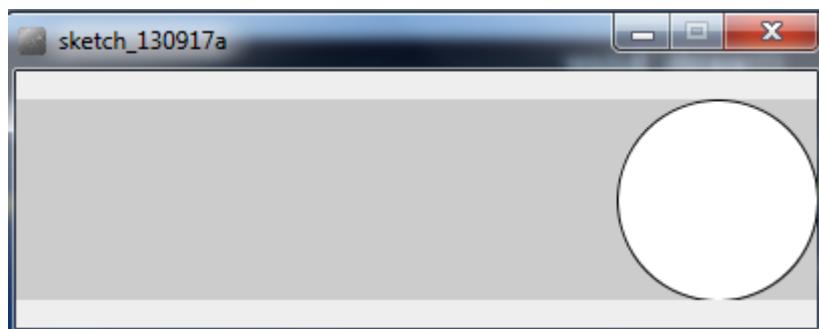
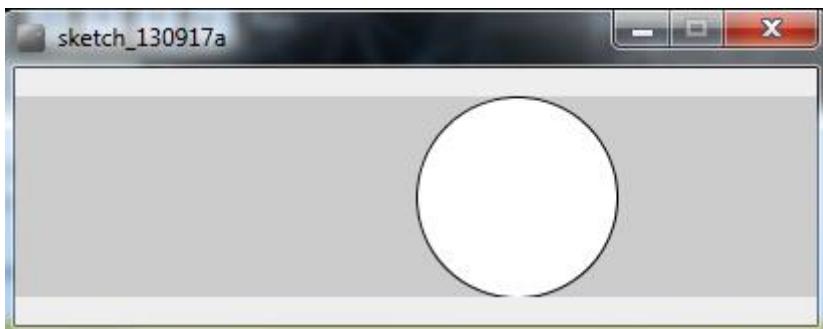
# if vs. if/else

- Here is a program that uses a **chained elif**

```
def setup():
 size(400,100)
 frameRate(2)
def draw():
 background(204)
 num = int(random(4))
 if num == 0:
 ellipse(50,50,100,100)
 elif num == 1:
 ellipse(150,50,100,100)
 elif num == 2:
 ellipse(250,50,100,100)
 else:
 ellipse(350,50,100,100)
```

# if vs. if/else

Every time the screen is drawn I will see *exactly 1* ellipse.



```
def setup():
 size(400,100)
 frameRate(2)
def draw():
 background(204)
 num = int(random(4))
 if num == 0:
 ellipse(50,50,100,100)
 elif num == 1:
 ellipse(150,50,100,100)
 elif num == 2:
 ellipse(250,50,100,100)
 else:
 ellipse(350,50,100,100)
```

# Practice Quiz Question: What is the Output of this Program?

```
num1 = 2
num2 = 7
decimal = 9

if num1 == 2:
 print("First")
elif num2 == 3:
 print("Second")
elif decimal == 9:
 print("Third")
else:
 print("Fourth")

if num2/num1 != 1:
 print("Fifth")

if num2/num1 != 3:
 print("Sixth")
else:
 print("Seventh")
```

# The Complete List of Input Functions and Variables is in the Processing API

| Mouse           | Keyboard      |
|-----------------|---------------|
| mouseButton     | key           |
| mouseClicked()  | keyCode       |
| mouseDragged()  | keyPressed()  |
| mouseMoved()    | keyPressed    |
| mousePressed()  | keyReleased() |
| mousePressed    | keyTyped()    |
| mouseReleased() |               |
| mouseWheel()    |               |
| mouseX          |               |
| mouseY          |               |
| pmouseX         |               |
| pmouseY         |               |

# Functions that Respond to Events Like User Input are Called “Event Handlers.”

| Mouse             | Keyboard        |
|-------------------|-----------------|
| mouseButton       | key             |
| → mouseClicked()  | keyCode         |
| → mouseDragged()  | keyPressed() ←  |
| → mouseMoved()    | keyPressed      |
| → mousePressed()  | keyReleased() ← |
| mousePressed      | keyTyped() ←    |
| → mouseReleased() |                 |
| → mouseWheel()    |                 |
| mouseX            |                 |
| mouseY            |                 |
| pmouseX           |                 |
| pmouseY           |                 |

# Processing's Predefined “System Variables” that are Used for User Input

| Mouse           | Keyboard      |
|-----------------|---------------|
| mouseButton     | key           |
| mouseClicked()  | keyCode       |
| mouseDragged()  | keyPressed()  |
| mouseMoved()    | keyPressed    |
| mousePressed()  | keyReleased() |
| mousePressed    | keyTyped()    |
| mouseReleased() |               |
| mouseWheel()    |               |
| mouseX          |               |
| mouseY          |               |
| pmouseX         |               |
| pmouseY         |               |

# True/False Question

- True / False  $(-2 < 1)$  and  $(3 == 3)$

# True/False Question

- True / False  $(-2 < 1)$  and  $(3 == 3)$

# True/False Question

- True / False **True** and  $(3 == 3)$

# True/False Question

- True / False **True** and **(3 == 3)**

# True/False Question

- True / False **True** and **True**

# True/False Question

- True / False **True** and **True**

# The answer is True

- True / False  $(-2 < 1)$  and  $(3 == 3)$

# Practice Quiz Questions

1. True / False  $(3 \neq 4)$  and  $(2 < 2)$
2. True / False  $(3 \neq 4)$  or  $(2 < 2)$
3. Name three "system variables" that are used for user input:
  - a) \_\_\_\_\_
  - b) \_\_\_\_\_
  - c) \_\_\_\_\_
4. Name two functions that are used for user input:
  - a) \_\_\_\_\_
  - b) \_\_\_\_\_

# Functions in Python

Functions need to be **defined** before they can be **called**

```
def printTwice(x):
 print(x)
 print(x)
```

```
def doubleIt(y):
 return 2 * y
```

```
printTwice("Hello")
print(doubleIt(42))
```

# Functions in Python

```
def printTwice(x):
```

```
 print(x)
```

```
 print(x)
```

```
def doubleIt(y):
```

```
 return 2 * y
```

```
Hello
```

```
Hello
```

```
84
```

```
printTwice("Hello")
```

```
print(doubleIt(42))
```

# Functions in Python

- Function definitions begin with the keyword **def** followed by the **function name** and parentheses/
- The function may zero or more **arguments**/
- The code block within every function starts with a colon and is indented.
- **return** exits a function, optionally passing back an expression to where the function was called/

```
def printTwice(x) :
 print(x)
 print(x)
```

```
def doubleIt(y) :
 return 2 * y
```

# Functions in Python

The *order* in which functions are **defined** doesn't matter, what does matter is the *order* in which they are **called**

```
def mystery1(x):
 return 2 * x
```

```
def mystery2(y):
 return y / 2
```

```
a = mystery2(5)
a = mystery1(a)
print(a)
```

# Functions with Optional Arguments

```
def someFunction(x, y=2, z=3):
 return x + y + z

print(someFunction(1))
print(someFunction(1,y=1,z=1))
print(someFunction(1,z=0))
print(someFunction(1,y=0))
```

# Functions with Optional Arguments

```
def someFunction(x, y=2, z=3):
 return x + y + z
```

```
print(someFunction(1))
print(someFunction(1,y=1,z=1))
print(someFunction(1,z=0))
print(someFunction(1,y=0))
```

6

3

3

4

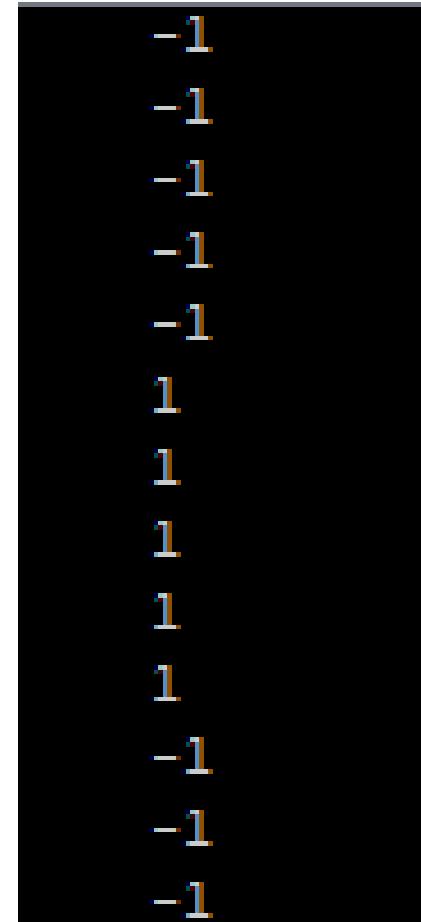
# mouseWheel () & event.count

- `mouseWheel()` has an argument `event` that has information about the mouse wheel.
- If you scroll the mouse wheel one way `event.count` is 1
- The other way is -1.

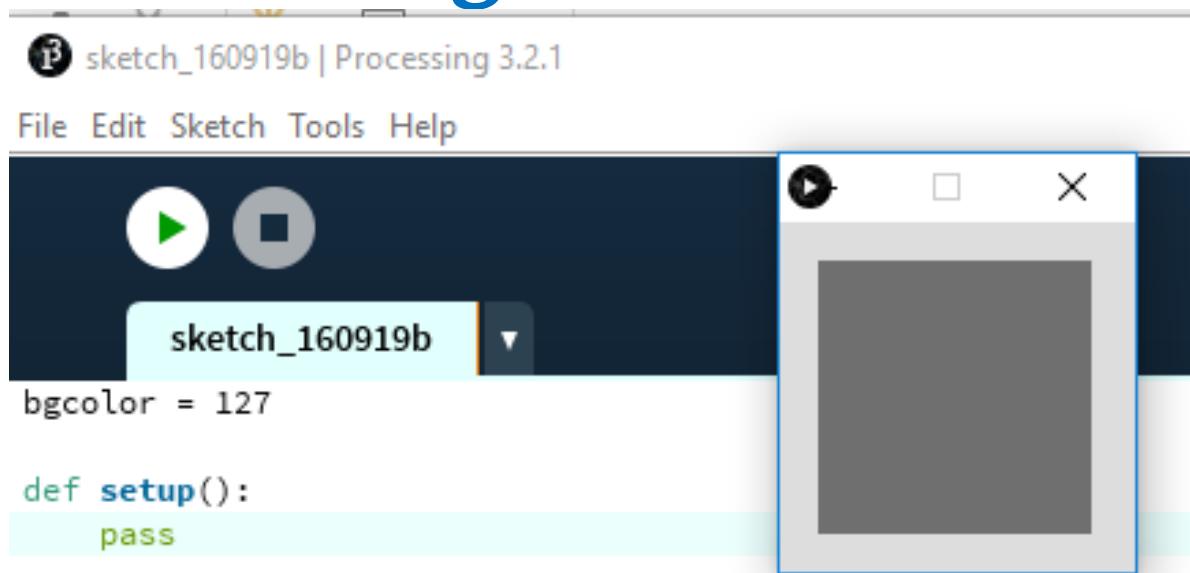
```
def setup():
 pass
```

```
def draw():
 pass
```

```
def mouseWheel(event):
 print(event.count)
```



# Moving the Mouse Wheel Darkens and Lightens the Background



sketch\_160919b | Processing 3.2.1

File Edit Sketch Tools Help

```
bgcolor = 127

def setup():
 pass

def draw():
 background(bgcolor)

def mouseWheel(event):
 global bgcolor
 bgcolor = bgcolor + event.count
 if bgcolor < 0:
 bgcolor = 0
 if bgcolor > 255:
 bgcolor = 255
```

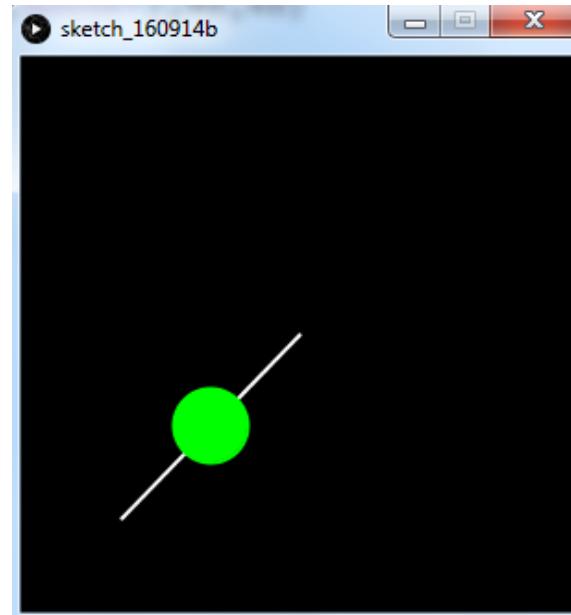
# Practice Quiz Question

Copy and paste following program into Processing. Replace the `return 0` with code that returns the average of the **two arguments**. The correct code will place the green ellipse at the midpoint of the line.

```
def setup():
 size(300,300)
 strokeWeight(2)

def draw():
 background(0)
 stroke(255)
 line(150,150,mouseX,mouseY)
 stroke(0,255,0)
 fill(0,255,0)
 ellipse(average(150,mouseX),average(150,mouseY),40,40)

def average(x,y):
 return 0 # replace with your code
```

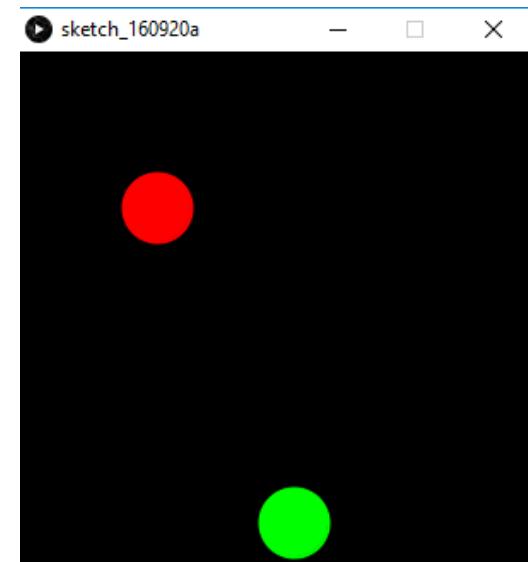


Copy and paste following program into Processing. Replace the two **return 0** statements with code that returns the half and the third of each **argument**. The correct code will place the red ellipse at  $\frac{1}{2}$  of the x and  $\frac{1}{3}$  of the y of the green circle.

```
def setup():
 size(300,300)

def draw():
 background(0)
 stroke(0,255,0)
 fill(0,255,0)
 ellipse(mouseX,mouseY,40,40)
 stroke(255,0,0)
 fill(255,0,0)
 ellipse(half(mouseX),third(mouseY),40,40)

def half(num):
 return 0 # replace 0 with half of argument
def third(num):
 return 0 # replace 0 with third of argument
```



# Practice Quiz Questions

1. true/false     $5 \% 2 = 1$
2. true/false    **NOT**  $(2 = 2)$
3. true/false     $(4 > 3)$  **AND**  $(3 \neq 3)$
4. true/false     $(4 > 3)$  **OR**  $(3 \neq 3)$
5. true/false    **mouseMoved()** is an example of a predefined system variable in Processing

- Note that on the AP exam:
  - != will be written  $\neq$
  - ! will be written **NOT**
  - == will be written  $=$

# tron



- Returns the color of one pixel
- **(10,10)** is on the left (black) side of the canvas, so **get()** returns a value equal to **color(0)** (black)
- **(90,10)** is on the right (white) side of the canvas, so **get()** returns a value that is NOT equal to **color(0)** (black)

**get()**



sketch\_160913a | Processing 3.2.1

File Edit Sketch Tools Help

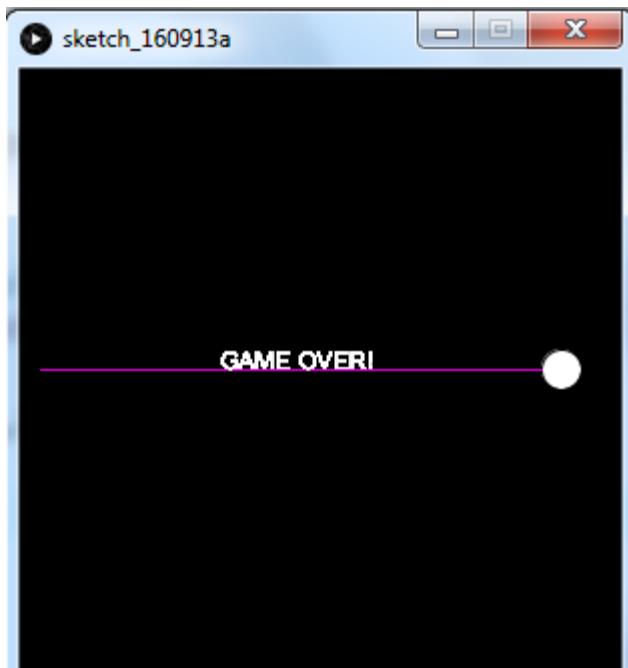
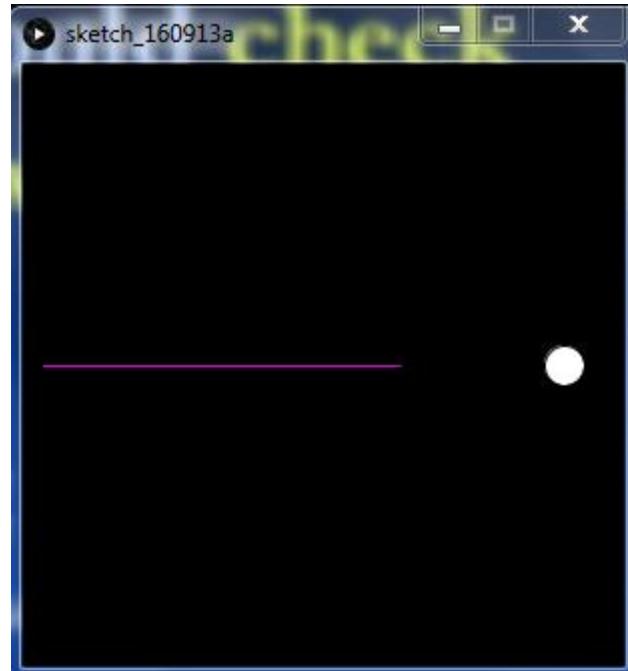
sketch\_160913a

```
background(0)
fill(255)
rect(50,0,50,100)
print(get(10,10) == color(0))
print(get(90,10) == color(0))
```

True  
False

# How we Check for a Crash with get ()

```
x = 10
y = 150
def setup():
 size(300,300)
 background(0)
 fill(255)
 ellipse(270,150,20,20)
def draw():
 global x,y
 # check for crash
 if get(x,y) != color(0):
 text("GAME
OVER!",100,150)
 else:
 stroke(255,0,255)
 point(x,y)
 x=x+1
```

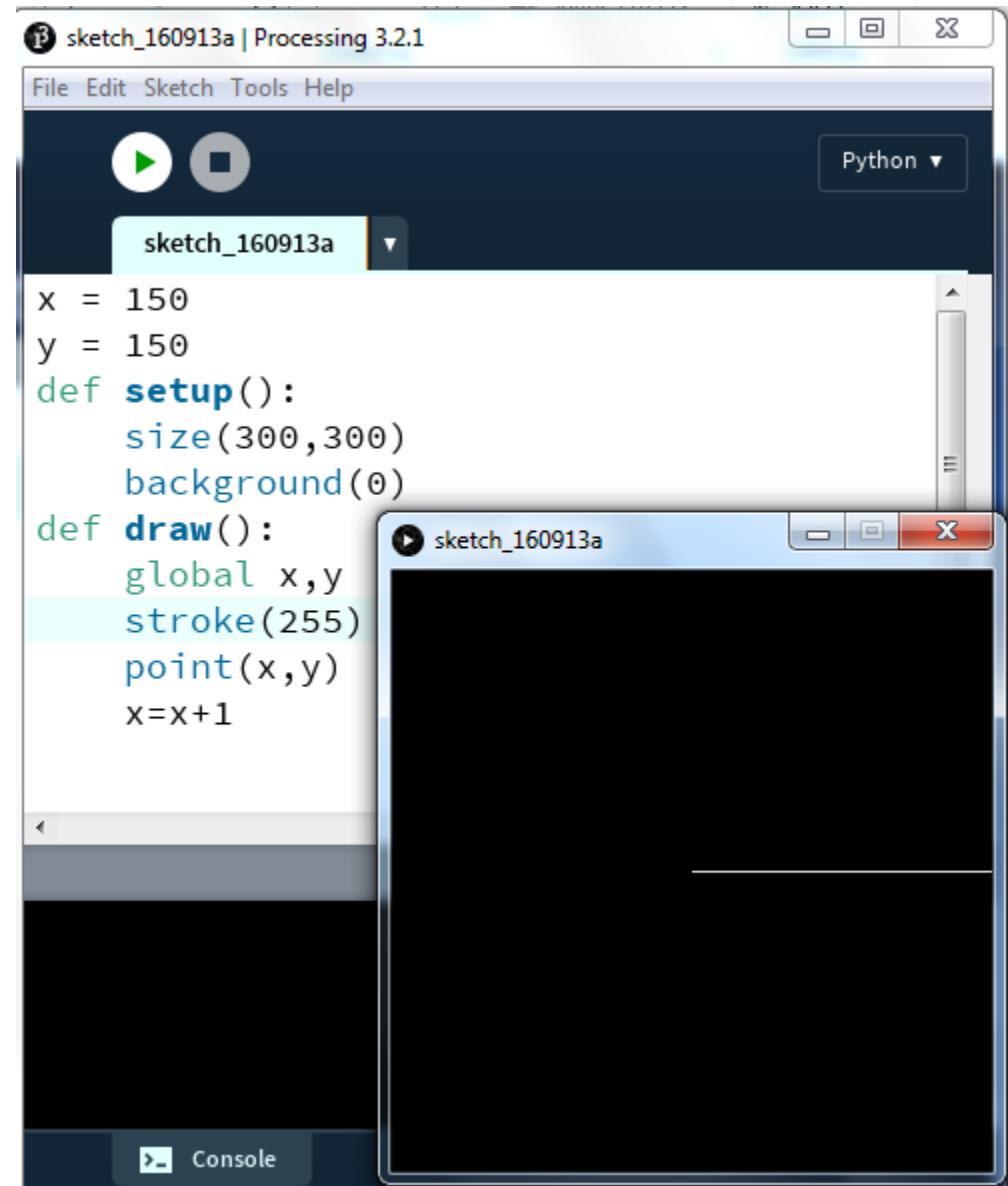


# Writing tron

- Don't try to write the program all at once.
- Get one detail working and then move on to the next thing.
- Let's start by making the light trail.

# Writing tron

Next, create a variable for our **direction** so we can *change* our direction.



```
x = 150
y = 150
def setup():
 size(300,300)
 background(0)
def draw():
 global x,y
 stroke(255)
 point(x,y)
 x=x+1
```

# Adding Code to draw()

Now, we'd probably want to make a **keyPressed()** function so that we can change direction with the keyboard.



```
p sketch_160913a | Processing 3.2.1
File Edit Sketch Tools Help
sketch_160913a ▾
x = 150
y = 150
direction = RIGHT
def setup():
 size(300,300)
 background(0)
def draw():
 global x,y
 stroke(255)
 point(x,y)
 if direction == RIGHT:
 x=x+1
 elif direction == LEFT:
 x=x-1
```

# Changing Direction in keyPressed()

```
def keyPressed():
 global direction
 if key == 'a':
 direction = LEFT
 elif key == 'd':
 direction = RIGHT
 elif key == 'w':
 direction = UP
 elif key == 's':
 direction = DOWN
```

# Using keyCode for the Arrow Keys in keyPressed()

```
def keyPressed():
 global direction
 if keyCode == LEFT:
 direction = LEFT
 elif keyCode == RIGHT:
 direction = RIGHT
 elif keyCode == DOWN:
 direction = DOWN
 else:
 direction = UP
```

# A gameOver Variable

- Keeping track of whether the game is over is important.
- One way to do that is with a variable.

```
x = 150
y = 150
direction = RIGHT
gameOver = False
```



# A Boolean gameOver Variable

- A boolean is either **True or False**
- Either the game is over or not
- When a player crashes we will change the value of the boolean to **True**

```
if get(x,y) != color(0):
 gameOver = True
```

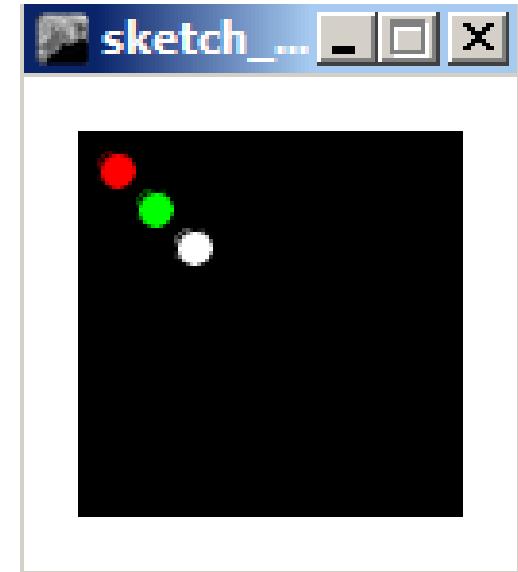
# A Boolean `gameOver` Variable

Once the game is over, we'll display an appropriate message.

```
def draw():
 if gameOver == True:
 text("GAME OVER",250,250)
 else:
 if get(x,y) != color(0):
 gameOver = True
 stroke(255,255,255)
 point(x,y)
```

# Practice Quiz Question: Find the Output

```
background(0)
fill(255,0,0)
ellipse(10,10,10,10)
fill(0,255,0)
ellipse(20,20,10,10)
fill(255)
ellipse(30,30,10,10)
print(get(10,10) != color(0))
print(get(20,20) == color(255))
print(get(30,30) == color(255))
print(get(50,50) != color(0))
```



# Adding a Computer Opponent

- Before I add the computer, let's clean up the code
- **draw()** is getting ugly
- let's move the human code into its own function

```
def draw():
 global x,y
 if get(x,y) != color(0):
 fill(255)
 textSize(24)
 text("GAME OVER",75,150)
 else:
 stroke(0,255,255)
 point(x,y)
 if direction == RIGHT:
 x = x + 1
 elif direction == LEFT: x = x - 1
```

# Adding a Computer Opponent

- There, that's better!
- Now we can use the **human()** function as a guide.
- **computer()** will be similar.

```
def draw():
 global x,y
 if get(x,y)!= color(0):
 fill(255)
 textSize(24)
 text("GAME OVER",75,150)
 else:
 human()
def human():
 global x,y
 stroke(0,255,255)
 point(x,y)
 if direction == RIGHT:
 x = x + 1
 elif direction == LEFT:
```

# Adding a Computer Opponent

The computer opponent will need it's own set of variables to keep track of its location and direction.

```
x = 10
y = 150
direction = RIGHT
compX = 290
compY = 150
compDir = LEFT
def setup():
 size(300, 300)
```

# Adding a Computer Opponent

- We will write a function that is very similar to `human()` called `computer()`
- In `computer()`, will use `compX`, `compY`, etc., instead of `x` & `y`

```
def draw():
 global x,y
 if get(x,y)!= color(0):
 fill(255)
 textSize(24)
 text("GAME OVER",75,150)
 else:
 human()
 computer()

def human():
```

# Adding a Computer Opponent

- The problem now is that our computer won't turn.
- We need to add code to make it "smart."
- Sometimes, this is called "AI" for artificial intelligence.

# Adding a Computer Opponent

- In each of the **four if** statements of the computer, we will **look ahead 1 place** to see if we are going to run into something
- If we are, we'll turn by **changing the value of compDir**

```
def computer():
 global compX,compY,compDir
 if compDir == RIGHT:
 compX = compX + 1
 if get(compX+1,compY) != color(0):
 compDir = DOWN
 elif compDir == LEFT:
```

# Practice Quiz Question

Find the output.

```
def mystery1(x):
 return x + 2

def mystery2(y):
 return y * 3

a = mystery1(mystery2(5))
print(a)
```

# Ending the Game

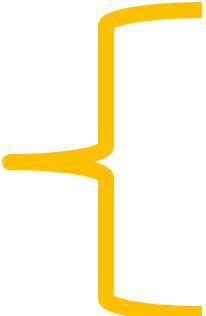
- I need to stop the game as soon as one player loses, otherwise things will get messed up
- That's what the **gameOver** variable is for

---

```
x = 160
y = 150
direction = RIGHT
compX = 490
compY = 250
compDir = LEFT
gameOver = False
def setup():
```

# Ending the Game

- I'll move the code that checks to see if the human crashes to the human function
- If the human crashes I'll change the variable to show the game is over



```
def human():
 global x,y,gameOver
 if get(x,y) != color(0):
 fill(255)
 textSize(24)
 text("GAME OVER \n HUMAN LOSES",75,150)
 gameOver = True
 stroke(0,255,255)
 point(x,y)
 if direction == DIRECTION.
```

# Ending the Game

I'll do the same in the computer function.

```
def computer():
 global compX,compY,compDir,gameOver
 if get(compX,compY) != color(0):
 fill(255)
 textSize(24)
 text("GAME OVER \n COMPUTER LOSES",75,150)
 gameOver = True
 stroke(0,0,255)
 point(compX,compY)
 if compDir == RTGHT:
```

# Ending the Game

- In `draw()`, I'll make sure that the human and the computer only move if the game isn't over.
- I test them separately so that they don't both lose on the same turn.

```
def draw():
 global x,y,gameOver
 if gameOver == False:
 human()
 if gameOver == False:
 computer()
```

# Practice Quiz Question: What is the Output of this Program?

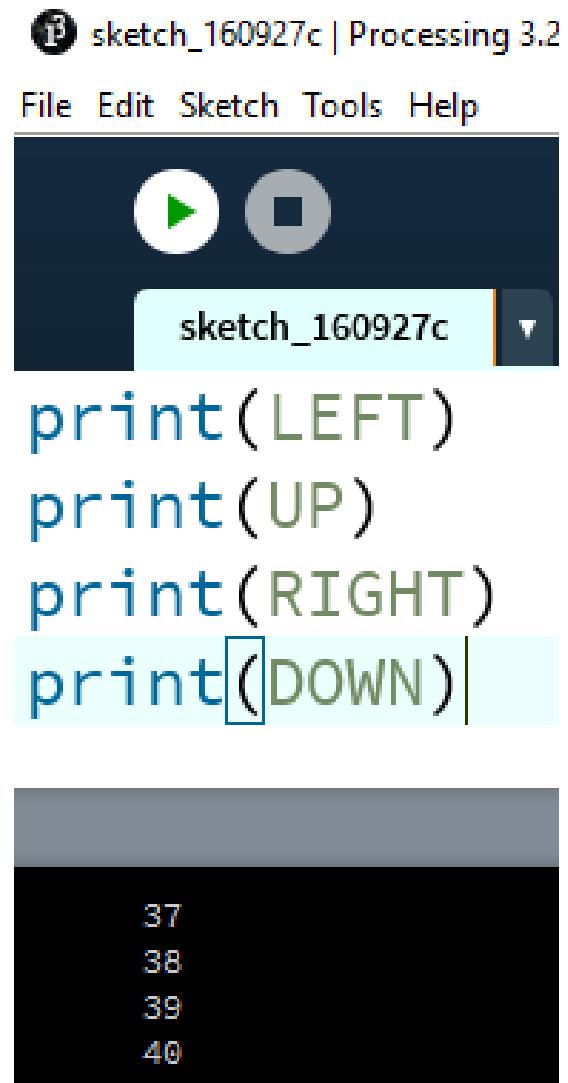
```
def setup():
 num1 = mystery(1,2)
 num2 = mystery(num1,1)
 num3 = mystery(mystery(1,1),num2)
 print(num3)

def draw():
 pass

def mystery(n1,n2):
 return n1 + (2*n2)
```

# A Clever Way to Handle Turning

- **LEFT**, **UP**, **RIGHT** and **DOWN** are the integers 37, 38, 39, 40.
- Turning to the left is decreasing the direction by one.
- Turning to the right is increasing the direction by one.



The screenshot shows the Processing IDE interface. At the top, there's a toolbar with File, Edit, Sketch, Tools, and Help. Below the toolbar, there are two buttons: a green play button and a grey square button. The title bar says "sketch\_160927c | Processing 3.2". The main code area contains the following code:

```
print(LEFT)
print(UP)
print(RIGHT)
print(DOWN)
```

The word "DOWN" is currently selected, indicated by a light blue background. In the bottom half of the window, the output of the code is displayed in a black text area:

```
37
38
39
40
```

# A Clever Way to Handle Turning

Now the human only needs two keys, turn right and turn left!

```
def keyPressed():
 global direction
 if keyCode == LEFT:
 direction = direction - 1
 if direction < 37:
 direction = 40
 if keyCode == RIGHT:
 direction = direction + 1
 if direction > 40:
 direction = 37
```

# Making the Line Thicker

- We can make a bigger dot by **increasing the `strokeWeight`**
- We'll need to **increase the amount we move** to compensate
- We'll also need to **slow down the `frameRate`**



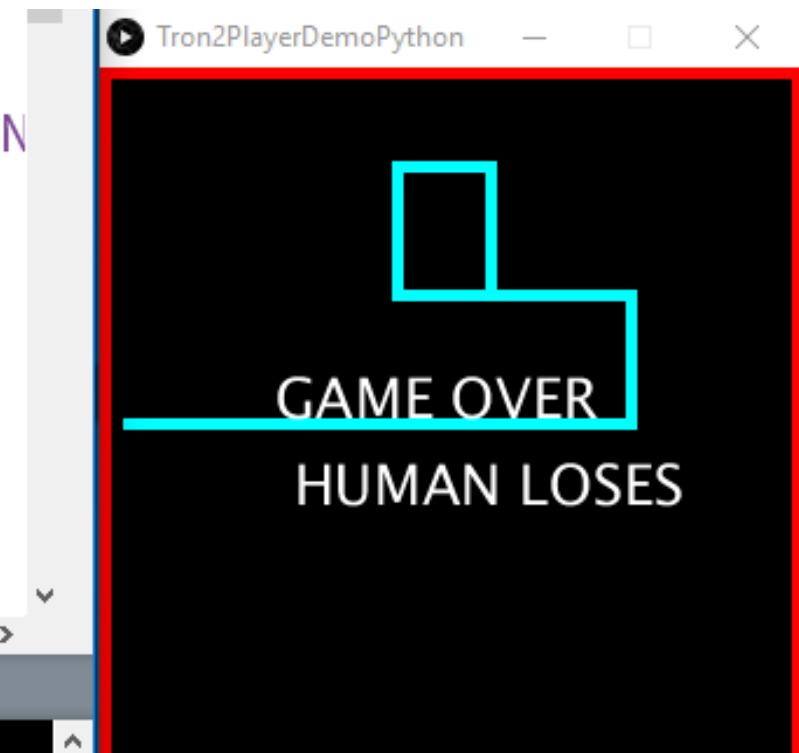
```
def setup():
 size(300,300)
 background(255,0,0)
 fill(0)
 rect(5,5,290,290)
 frameRate(10)
 strokeWeight(5)
```

```
text("GAME OVER \n HUMAN LOSES")
gameOver = True
stroke(0,255,255)
point(x,y)
if direction == RIGHT:
 x = x + 5
elif direction == LEFT:
 x = x - 5
```

# Making the Line Thicker

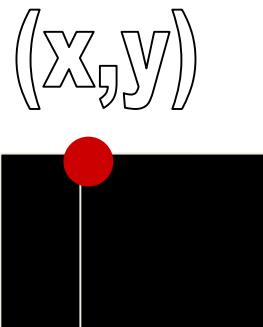
- Using a `rect` instead of a `point` creates a smoother line.
- Notice that the size of the `rect` is one less than the amount we move.

```
textSize(24)
text("GAME OVER \n HUMAN")
gameOver = True
stroke(0,255,255)
fill(0,255,255)
rect(x,y,4,4)
if direction == RIGHT:
 x = x + 5
```



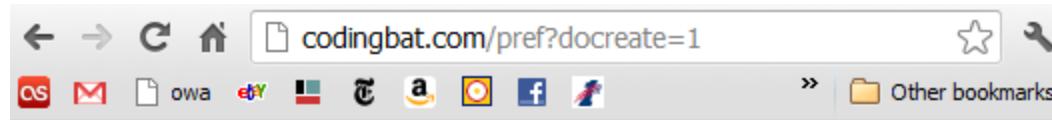
# Making the Line Thicker

- I need to increase the amount I move, otherwise  $(x,y)$  lands on part of the previous rectangle
- It's not black, so I crash and die : (



# Codingbat.com

## Create an account



CodingBat code practice

### Create Account

Please enter information to create a new account. We use your email address as your id just so it's memorable and for password reset, not for spamming. New accounts automatically pick up any saved or done problems of the current session. The name field is not required, but with the Teacher Share feature, it can help the teacher see who is who. The password should have at least 6 characters.

|                                                |                                               |
|------------------------------------------------|-----------------------------------------------|
| Email (used as account id)                     | <input type="text"/>                          |
| Password                                       | <input type="password"/>                      |
| Confirm password                               | <input type="password"/>                      |
| Name (last, first)<br>e.g. <b>Smith, Alice</b> | <input type="text"/> (optional)               |
|                                                | <input type="button" value="Create Account"/> |

## Python> Warmup-1 > sleepIn and monkeyTrouble

Make sure you are logged in!

[Java](#)[Python](#)

### Warmup-1 > sleep\_in

[prev](#) | [next](#) | [chance](#)

The parameter `weekday` is `True` if it is a weekday, and the parameter `vacation` is `True` if we are on vacation. We sleep in if it is not a weekday or we're on vacation. Return `True` if we sleep in.

`sleep_in(False, False) → True`  
`sleep_in(True, False) → False`  
`sleep_in(False, True) → True`

[Go](#)[...Save, Compile, Run](#)[Show Solution](#)

```
def sleep_in(weekday, vacation):
```

# Making the Computer Unpredictable

- Here the cyan human created a trap for the blue computer.
- The computer pattern is a predictable spiral, so it is easy to trap.



# Making the Computer Unpredictable

- The following code checks to see if the **UP** direction **is blocked**.

```
compDir = UP
elif compDir == UP:
 compY = compY - 5
 if get(compX, compY-5) != color(0):
 compDir = RIGHT
else:
```

# Making the Computer Unpredictable

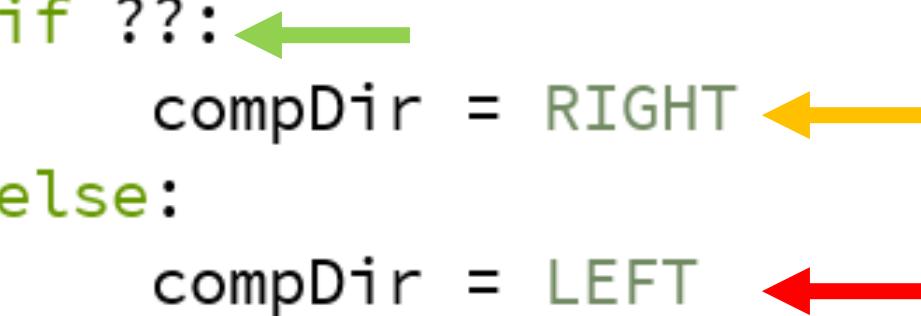
- The problem is that the computer predictably turns **RIGHT** if **UP** is blocked

```
 compDir = UP
elif compDir == UP:
 compY = compY - 5
 if get(compX, compY-5) != color(0):
 compDir = RIGHT
else:
```

# Making the Computer Unpredictable

- What could we put here so that the computer might unpredictably turn RIGHT or LEFT?

```
compDir = UP
elif compDir == UP:
 compY = compY - 5
 if get(compX,compY-5) != color(0):
 if ?: ←
 compDir = RIGHT ←
 else:
 compDir = LEFT ←
else:
 compY = compY + 5
```



# Making the Computer Unpredictable

- What if LEFT was blocked?
- What if both directions were open?

```
 compDir = UP
elif compDir == UP:
 compY = compY - 5
 if get(compX,compY-5) != color(0):
 if get(compX-5,compY) != color(0): #left blocked
 compDir = RIGHT
 elif get(compX+5,compY) != color(0): #right blocked
 compDir = LEFT
 elif random(0,2) == 1: # neither blocked
 compDir = RIGHT # so we can
 else: # randomly choose
 compDir = LEFT
else:
```

## Python> Warmup-1 > sum\_double and diff21

Make sure you are logged in!

Java

Python

---

### Warmup-1 > sum\_double

[prev](#) | [next](#) | [chance](#)

Given two int values, return their sum. Unless the two values are the same, then return double their sum.

sum\_double(1, 2) → 3  
sum\_double(3, 2) → 5  
sum\_double(2, 2) → 8

# Algorithm

- An algorithm is a step by step process.
- Every computer program has at least one algorithm.
- Most have multiple algorithms.
- Here is an algorithm, a step by step process for choosing the direction of our computer tron opponent:
  - If traveling **UP** and **UP** is blocked turn **RIGHT**
  - If traveling **RIGHT** and **RIGHT** is blocked turn **DOWN**
  - If traveling **DOWN** and **DOWN** is blocked turn **LEFT**
  - If traveling **LEFT** and **LEFT** is blocked turn **UP**

# Heuristics

- A heuristic is NOT a step by step process
- A heuristic is an alternative to an algorithm
- It's another way for solving a problem, but it may not give you the best result
- Examples of heuristics:
  - Guess and Check
  - Rule of Thumb
  - Common sense
  - Educated Guess
  - Process of elimination
  - Draw a picture
  - Work backwards

# Heuristics

- $3^x + 5^x = 70$  is an example of a math problem that requires a heuristic approach.
- It's impossible to solve for  $x$  and get an exact answer.
- The best solution will be a close approximation.

# Disadvantages of Heuristics

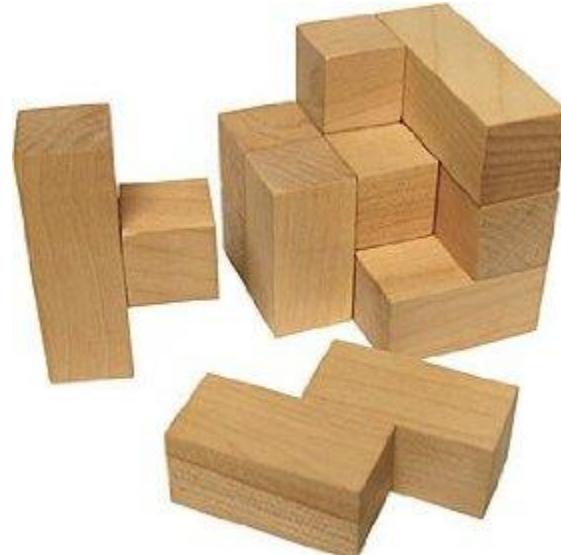
- No guarantee it will solve the problem.
- May not lead to the best solution.

# Disadvantages of Heuristics

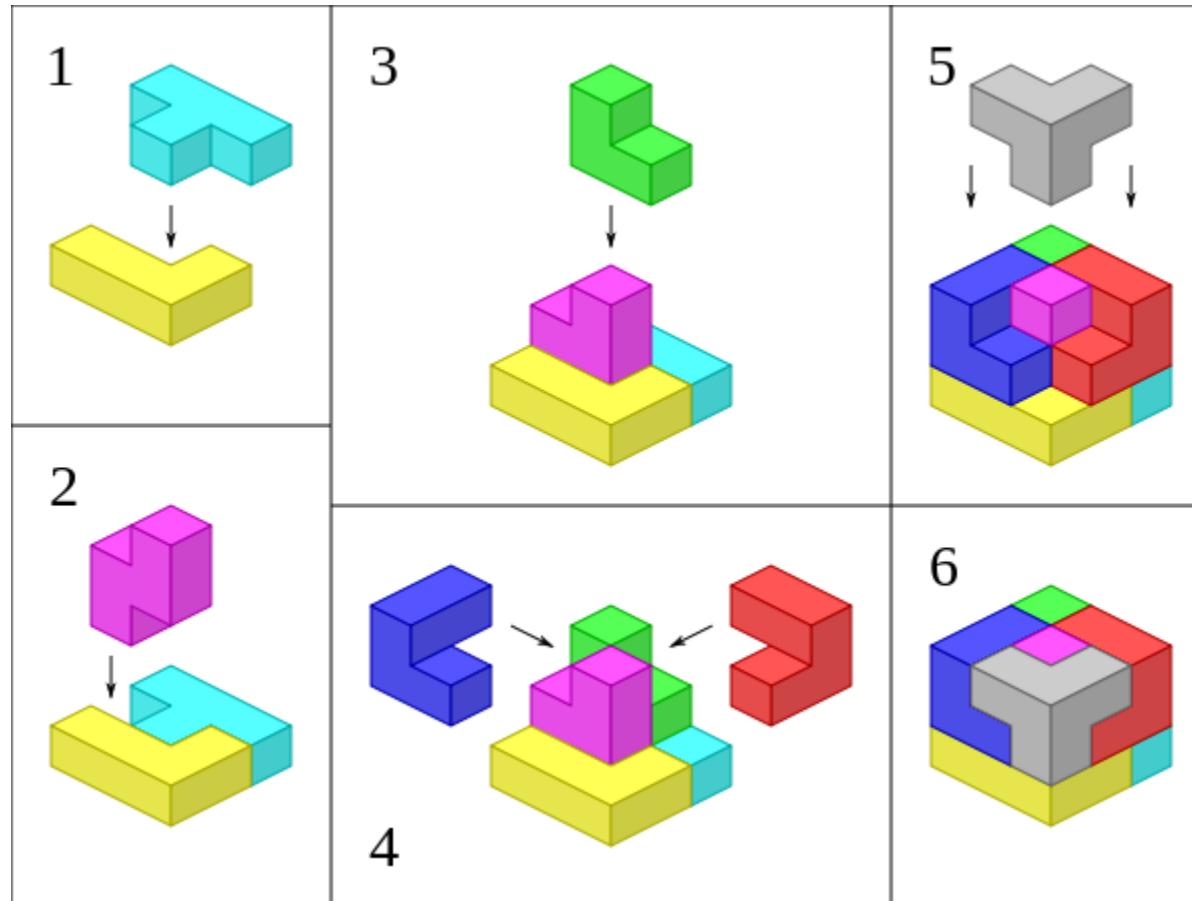
- May be quicker and easier to use than an algorithm.
- There may not be an algorithmic solution.
- Some problems can not be solved in a reasonable amount of time using an algorithm.
- Allows for more creativity and originality than a algorithm.

# Heuristics

- The soma cube puzzle can be solved using a heuristic of a few “rules of thumb” or guidelines:
  - Start with the biggest pieces
  - Never arrange more than three units in a row
  - Don’t leave a 1x1x1 hole
- <https://www.youtube.com/watch?v=jJ3CV3yhajM>

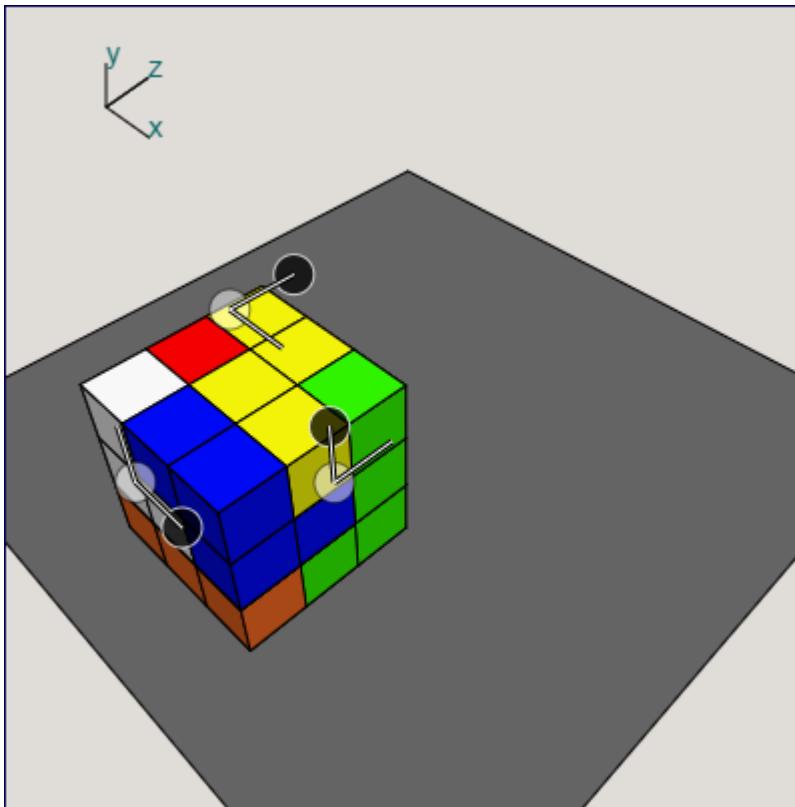


# An Algorithm to Solve a Soma Cube



# Online Soma Cube

- <http://www.mathythings.com/soma4.htm>
- Or Google “mathy things soma puzzle”



# Practice Quiz Question

Which of the following programs is most likely to benefit from a heuristic?

- A. A program that calculates a student's body mass index based on their height and weight.
- B. A program that calculates the interest on a loan.
- C. A program that predicts which movie a person will enjoy based on their previous viewing history.
- D. A program that draws the midpoint on a line segment.

# Practice Quiz Question

Which one of the following programs is LEAST likely to benefit from a heuristic approach?

- A. A program that predicts which song a person will enjoy based on their previous listening history
- B. A program that predicts the date of a future terrorist attack
- C. A program that predicts the quickest driving route from home to school
- D. A program that predicts a person's pant size based on the measurements of their waist and inseam

Optional: Try solving the [mathy things soma puzzle](#)

# Functions on the AP exam

- Python function definitions start with **def**

```
def diff21(n):
 if n <= 21:
 return 21 - n
 else:
 return (n -
21) * 2
```

- The AP Exam uses **PROCEDURE**

```
PROCEDURE diff21(n)
{
 IF (n <= 21)
 {
 RETURN 21 - n
 }
 ELSE
 {
 RETURN (n - 21) * 2
 }
}
```

# Practice Quiz Question

When is it appropriate to use a heuristic to solve a problem?  
Choose the best answer.

- A. When an approximate solution is good enough.
- B. When there is no single correct solution to a problem.
- C. When it is impossible or difficult to calculate an exact answer in a reasonable amount of time.
- D. When an original, unique and/or creative solution is desired.
- E. All of the above.

# Problems are Either. . .

Decidable:

algorithm(s) can be constructed to answer “yes” or “no” for all inputs...

Is this an even number?

Undecidable:

No algorithm can be constructed that always leads to a correct yes or no answer. (key word = always!) There may be instances that have an algorithmic solution, but there is no algorithmic solution that solves all instances of the problem.

# A Problem Computers Can't Solve

<https://www.youtube.com/watch?v=xiMl5mUESX>

M

YouTube #VOTE IRL

Search

§1 Input: Finite string,  
using a constant  
number of symbols

§2 Output: Finite string,  
using a constant  
number of symbols

§3 Output is an  
objectively correct  
and definitive  
answer

The diagram shows a rectangular window with a title bar containing '30%' in red. Below the title bar is a blue progress bar with a black slider. In the bottom right corner of the window is an oval button labeled 'Cancel'. Below the window is a small hourglass icon.

The Halting Problem - Intro to Theoretical Computer Science

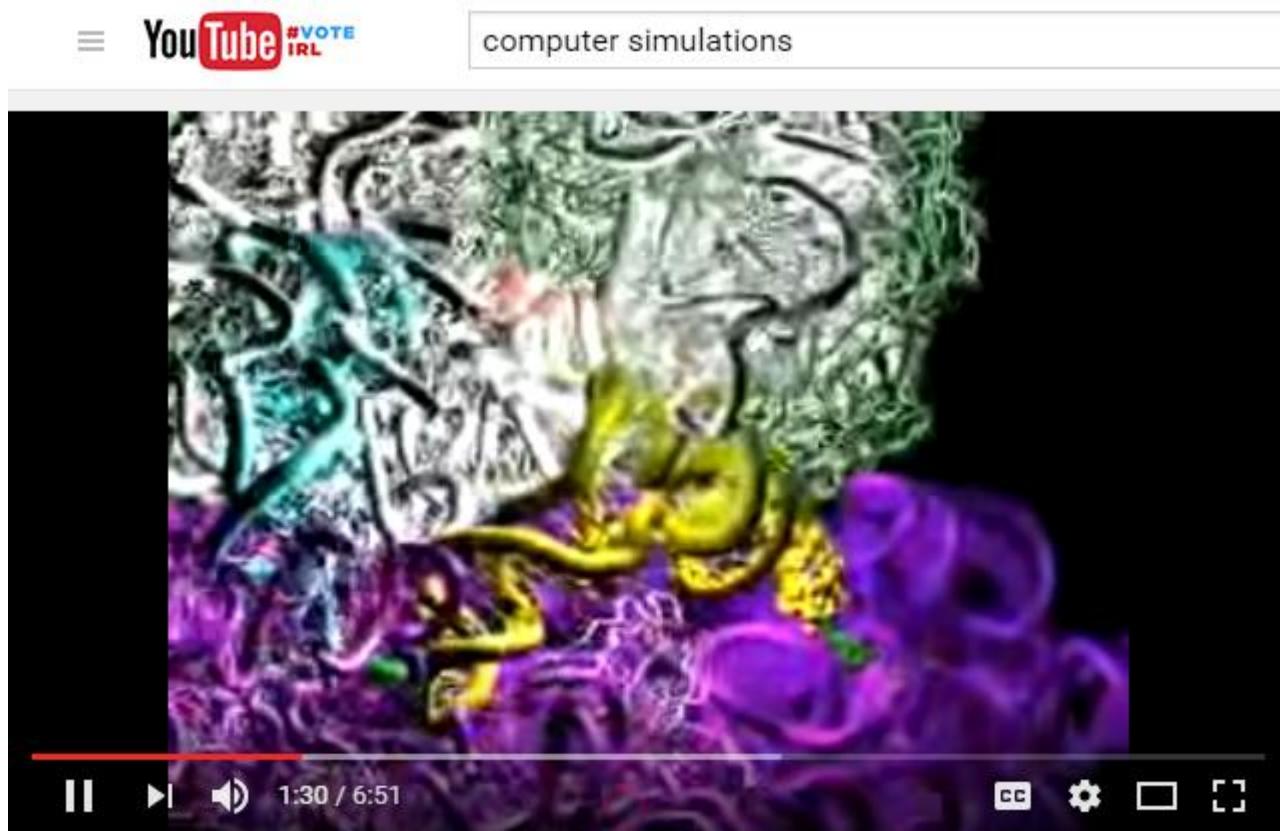
# Practice Quiz Question

Which of the following statements is true?

- A. Every problem can be solved with an algorithm for all possible inputs, in a reasonable amount of time, using a modern computer.
- B. Every problem can be solved with an algorithm for all possible inputs, but some will take more than 100 years, even with the fastest possible computer.
- C. Every problem can be solved with an algorithm for all possible inputs, but some of these algorithms have not been discovered yet.
- D. There exist problems that no algorithm will ever be able to solve for all possible inputs.

# Video: Computer Simulations

<https://www.youtube.com/watch?v=yEP-pLvSdXs>



Cool-Amazing Scientific Computer Simulations

# Computer Simulations

- Simulations are simplified representations of more complex problems.
- Simulations can reduce the time, cost and danger of building and testing in the “real world.”
- The results of simulations *may* generate new knowledge of the problem being modeled.
- A simulation will probably not produce an exact answer or perfect solution to a problem.

# How could you answer the question : “On average, how many times do you need to flip a coin to get 3 heads in a row?”

- You could solve an equation like
$$E(X_3) = 0.5 \cdot (1 + E(X_3)) + 0.25 \cdot (0.5 \cdot 3 + 0.5 \cdot (3 + E(X_3))) + 0.25 \cdot (2 + E(X_3)) \\ E(X_3) = 0.5 \cdot (1 + E(X_3)) + 0.25 \cdot (0.5 \cdot 3 + 0.5 \cdot (3 + E(X_3))) + 0.25 \cdot (2 + E(X_3))$$
- You could hire someone to flip a coin for a few thousand times.
- You could write a computer simulation.

**How could you answer the question :  
“On average, how many times do you need  
to flip a coin to get 3 heads in a row?”**

**Advantages of a computer simulation:**

- Faster and cheaper than performing “real life” experiments.
- Can give insight into a problem that be too long or difficult to solve with a mathematical equation.

**How could you answer the question :  
“On average, how many times do you need  
to flip a coin to get 3 heads in a row?”**

**Disadvantages** of a computer simulation:

- Simplifications may make results inaccurate or unhelpful.
- Won’t produce the exact answer a mathematical calculation could.

# Computer Simulations

- Write a computer simulation to determine the odds of flipping 3 heads in a row.
- Check your answer at the site below.
- <https://www.quora.com/What-is-the-expected-number-of-coin-flips-until-you-get-3-heads-in-a-row#>

# Start by Simulating a Coin Flip

- We'll say **True** is heads and **False** is tails.
- Run the program until you are satisfied that there is a 50/50 chance of **True** or **False**.



The image shows a Scratch-like programming interface. At the top, there are two circular buttons: a green play button on the left and a grey stop button on the right. Below these buttons is a text input field containing the code:

```
sketch_161107a
flip1 = random(1) < .5
print(flip1)
```

Below the code area is a preview window. The preview window has a light blue header bar. The main preview area is black and displays the word "True" in orange text. There is also a small white arrow icon in the top-left corner of the preview area.

# We Need to Store Three Coin Flips and Check to See if they are all Heads

- It's unusual to get three heads in a row
- We'll need to run the program 20 or 30 times to make sure it prints **True** every now and then

```
flip1 = random(1) < .5
flip2 = random(1) < .5
flip3 = random(1) < .5
print(flip1 == True and flip2 == True and flip3 == True)
```

# Count How Many Flips it Takes to get 3 Heads in a Row.

- We'll need a variable **numFlips** to keep track
- Why is **numFlips** initialized with 3?

```
flip1 = random(1) < .5
flip2 = random(1) < .5
flip3 = random(1) < .5
numFlips = 3
while not(flip1 == True and flip2 == True and flip3 == True):
 flip1 = flip2
 flip2 = flip3
 flip3 = random(1) < .5
 numFlips = numFlips + 1
print(numFlips)
```

# How it Might Work

|       |       |       |
|-------|-------|-------|
| False | False | True  |
| flip1 | flip2 | flip3 |

|          |
|----------|
|          |
| numFlips |

```
flip1 = random(1) < .5
flip2 = random(1) < .5
flip3 = random(1) < .5
numFlips = 3
while not(flip1 == True and flip2 == True and flip3 == True):
 flip1 = flip2
 flip2 = flip3
 flip3 = random(1) < .5
 numFlips = numFlips + 1
print(numFlips)
```

# How it Might Work

|       |       |         |
|-------|-------|---------|
| False | True  | True    |
| flip1 | flip2 | flip3   |
|       |       | numFlip |
|       |       |         |

```
flip1 = random(1) < .5
flip2 = random(1) < .5
flip3 = random(1) < .5
numFlips = 3
while not(flip1 == True and flip2 == True and flip3 == True):
 flip1 = flip2
 flip2 = flip3
 flip3 = random(1) < .5
 numFlips = numFlips + 1
print(numFlips)
```

Here it took 5 flips to get three heads.  
That was just one trial, we'll need to  
repeat this many times.

|       |       |       |
|-------|-------|-------|
| True  | True  | True  |
| flip1 | flip2 | flip3 |

|          |
|----------|
|          |
| numFlips |

```
flip1 = random(1) < .5
flip2 = random(1) < .5
flip3 = random(1) < .5
numFlips = 3
while not(flip1 == True and flip2 == True and flip3 == True):
 flip1 = flip2
 flip2 = flip3
 flip3 = random(1) < .5
 numFlips = numFlips + 1
print(numFlips)
```

# Create a Function Definition

```
def oneTrial():
```

- We'll **return numFlips** rather than print it.
- Printing 1000 or more numbers would be a mess!

```
def oneTrial():
 flip1 = random(1) < .5
 flip2 = random(1) < .5
 flip3 = random(1) < .5
 numFlips = 3
 while not(flip1 == True and flip2 == True and flip3 == True):
 flip1 = flip2
 flip2 = flip3
 flip3 = random(1) < .5
 numFlips = numFlips + 1
 return numFlips
```

# Create two Variables sum and count

- To calculate the average number of flips, we'll divide **sum** of all flips by the **count** of trials.

---

```
def oneTrial():
 flip1 = random(1) < .5
 flip2 = random(1) < .5
 flip3 = random(1) < .5
 numFlips = 3
 while not(flip1 == True and flip2 == True and flip3 == True):
 flip1 = flip2
 flip2 = flip3
 flip3 = random(1) < .5
 numFlips = numFlips + 1
 return numFlips
sum = 0.0
count = 0
```

# Create two Variables sum and count

- To calculate the average number of flips, we'll divide sum of all flips by the count of trials.
- For example, if we did three trials that took
  - 35 flips
  - 14 flips
  - 11 flips
- The average would be  $(35+14+11)/3$
- $=60/3$
- $=20$

# Write a loop to do 1000 Trials

Increase the number of trials until you get a stable result.

---

```
def oneTrial():
 flip1 = random(1) < .5
 flip2 = random(1) < .5
 flip3 = random(1) < .5
 numFlips = 3
 while not(flip1 == True and flip2 == True and flip3 == True):
 flip1 = flip2
 flip2 = flip3
 flip3 = random(1) < .5
 numFlips = numFlips + 1
 return numFlips

sum = 0.0
count = 0
while count < 1000:
 sum = sum + oneTrial()
 count = count + 1
print(sum/count)
```

# Practice Quiz Question #1

The DMV wants to minimize the amount of time customers wait in line. It hires computer researchers to create a simulation of both a single line where the customer at the front waits for the next available window, and of separate lines for each window. Which of the following is NOT true about the simulation?

- A. The simulation can also test how the number of windows effects wait time.
- B. The DMV can use the simulation to investigate these two options without causing inconvenience for customers.
- C. The DMV may consider new alternatives based on the simulation results.
- D. The simulation will not produce usable results because actual customer data are not available.

# Practice Quiz Question #2

Which of the following are reasons for the DMV to use simulation software in this context?

- I. Using simulation software can save the DMV time and money vs. measuring wait times in the real world.
  - II. Using simulation software guarantees an optimal solution to the problem.
  - III. The DMV can use the simulation software to identify problem employees.
- 
- A. I only
  - B. II only
  - C. III only
  - D. I and II only
  - E. I, II, and III

# Practice Quiz Question #3: Choose two

Of the following, which two outcomes are most likely to be results of the DMV simulation?

- A. Better understanding of the effect of temporarily closing DMV windows.
- B. Better understanding of the effect of different placements of informational literature and brochures.
- C. Better understanding of the impact of widening the front doors of public buildings.
- D. Better understanding of the impact of decreasing the average time a person spends at a DMV window.

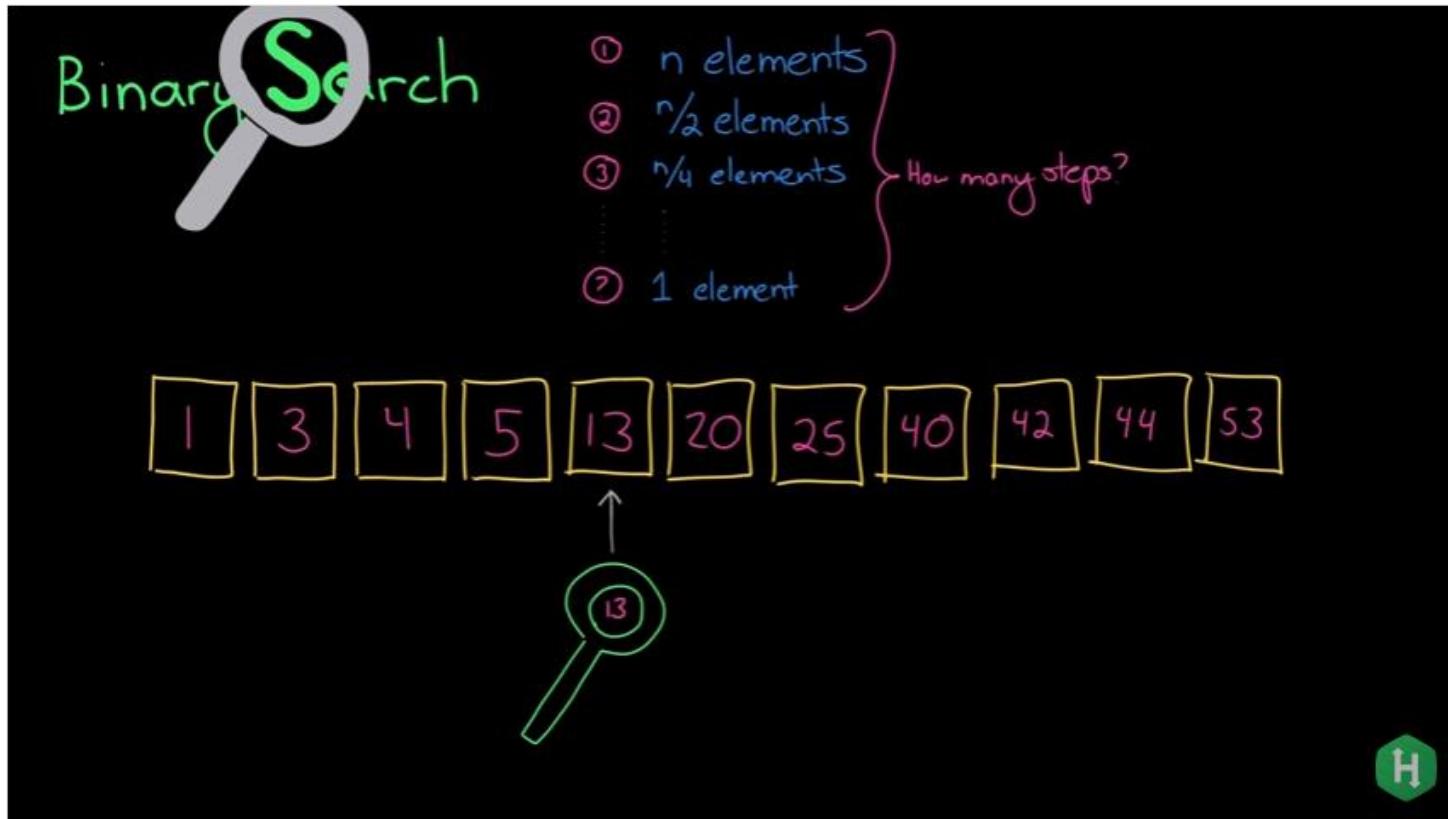
# A Guessing Game

- I have 15 numbered cups
- I'm going to hide a chocolate under one cup
- I'll give you ***four guesses*** to find the chocolate
- With each guess, I'll tell you:
  - If you found it
  - If your guess is too high
  - If your guess is too low.



# Video: Binary Search

- <https://www.youtube.com/watch?v=P3YID7liBug>
- Just watch the first 2:42.



# Linear Search

- If the elements of a list aren't in order, you need to do a linear search
- Since the number could be anywhere in the list (**if** it is even in the list) the order we click on the indexes isn't important

## Linear Search Widget

|                   |     |     |     |     |     |     |     |     |     |      |      |      |      |      |      |      |      |      |      |      |  |  |
|-------------------|-----|-----|-----|-----|-----|-----|-----|-----|-----|------|------|------|------|------|------|------|------|------|------|------|--|--|
| Number at [index] |     |     |     |     |     | 64  |     |     |     |      |      |      |      |      |      |      |      |      |      |      |  |  |
| [index]           | [1] | [2] | [3] | [4] | [5] | [6] | [7] | [8] | [9] | [10] | [11] | [12] | [13] | [14] | [15] | [16] | [17] | [18] | [19] | [20] |  |  |

Try to find 798

Number of Guesses 1

Click on an [index] to reveal the number stored at that index  
*Refresh your browser to play again*

# Binary Search

- If the numbers are in order, I can do a binary search.
- If I'm careful to divide the list into half with each guess, I can determine if the number is in the list much more quickly.

## Binary Search Widget

Number at [index]

|         |     |     |     |     |     |     |     |     |     |      |      |      |      |      |      |      |      |      |      |      |     |     |
|---------|-----|-----|-----|-----|-----|-----|-----|-----|-----|------|------|------|------|------|------|------|------|------|------|------|-----|-----|
| <22     | <22 | <22 | <22 | 22  |     |     |     |     | 70  | >70  | >70  | >70  | >70  | >70  | >70  | >70  | >70  | >70  | >70  | >70  | >70 | >70 |
| [index] | [1] | [2] | [3] | [4] | [5] | [6] | [7] | [8] | [9] | [10] | [11] | [12] | [13] | [14] | [15] | [16] | [17] | [18] | [19] | [20] |     |     |

Try to find 40

Number of Guesses 2

Click on an [index] to reveal the number stored at that index  
*Refresh your browser to play again*

# Linear and Binary Search

- With a partner or two, use the two widgets to complete the worksheet.
  - Each person should submit a completed worksheet.

## *Linear and Binary Search*

In this assignment you and a partner or two will use two “widgets” to explore the different searching techniques of linear and binary search.

## Linear Search

The first searching technique we will explore is linear search. Start by opening a web browser and visiting <https://apcsprinciples.github.io/LinearSearch/>. You should see a page that looks like this:

# Linear Search Widget

Number at [index]

|         |     |     |     |     |     |     |     |     |     |      |      |      |      |      |      |      |      |      |      |      |
|---------|-----|-----|-----|-----|-----|-----|-----|-----|-----|------|------|------|------|------|------|------|------|------|------|------|
|         |     |     |     |     |     |     |     |     |     |      |      |      |      |      |      |      |      |      |      |      |
| [index] | [1] | [2] | [3] | [4] | [5] | [6] | [7] | [8] | [9] | [10] | [11] | [12] | [13] | [14] | [15] | [16] | [17] | [18] | [19] | [20] |

Try to find 798

Number of Guesses 0

# map ()

- The **map ()** function converts from **one scale** to **another**.

```
yellowNum
```

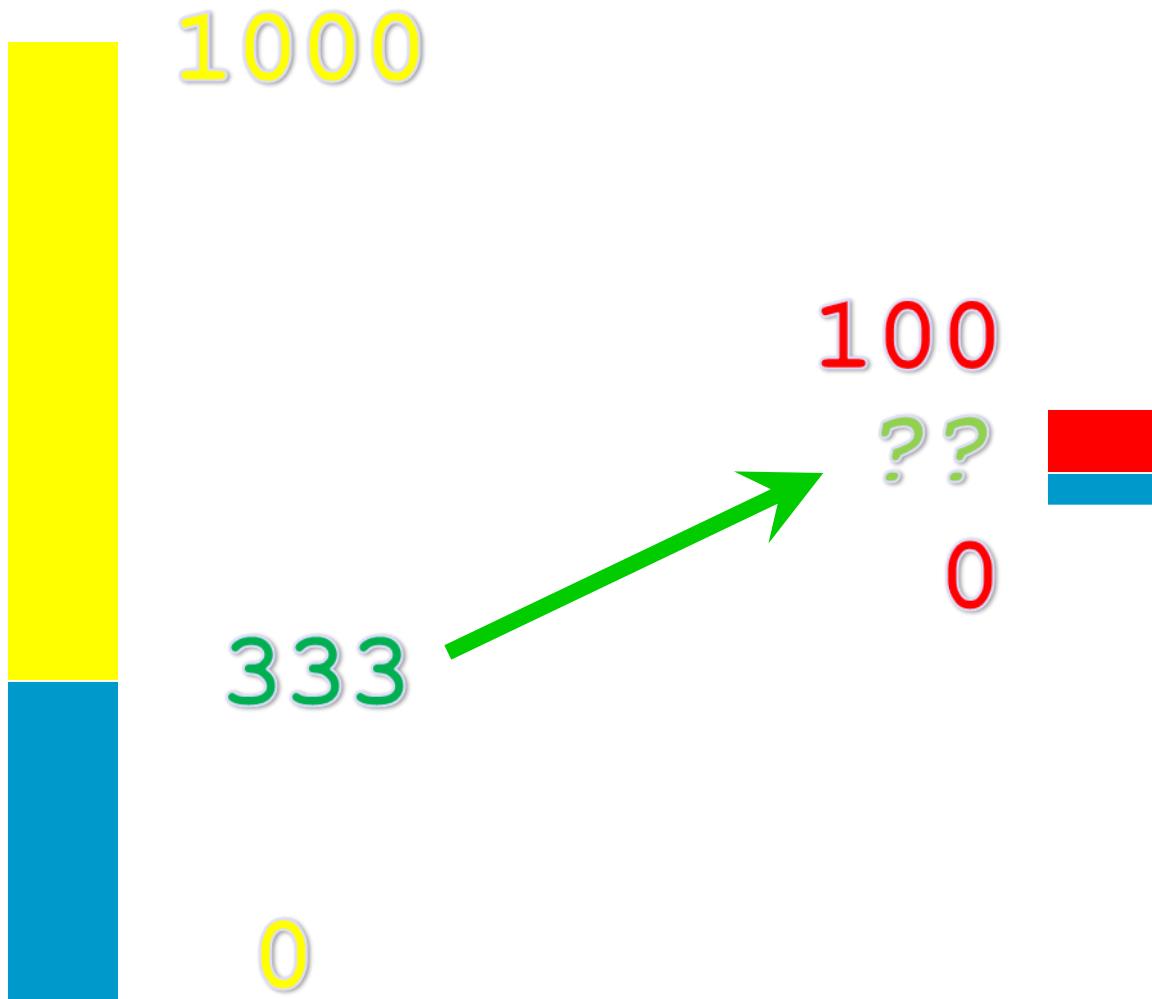
```
redNum = map(yellowNum, 0, 1000, 0, 30)
```

```
print(redNum)
```

- In this example, **redNum** will be assigned the value 15.
- **yellowNum** is “half way” between **0** and **1000** so **map ()** returns a value halfway between **0** and **30**.

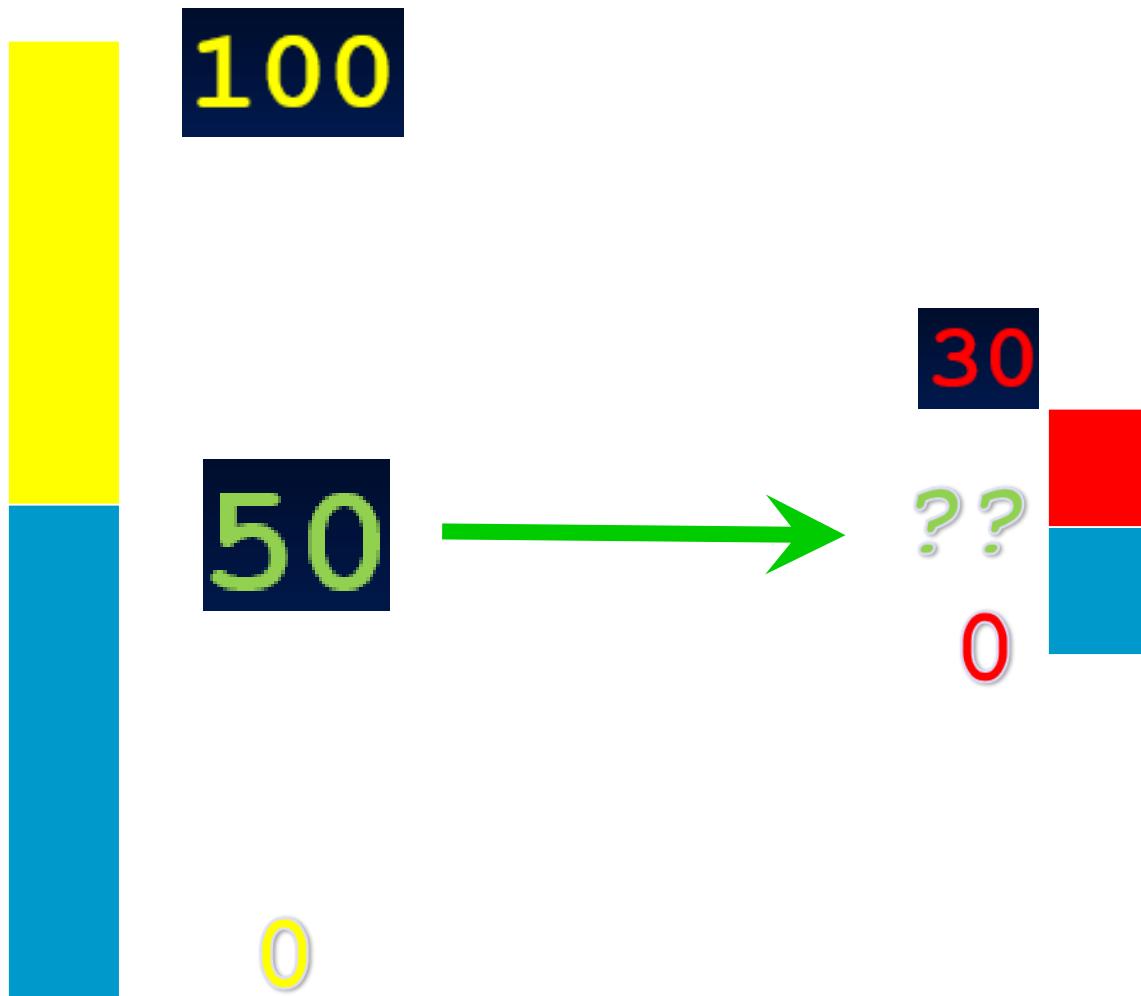
What would this display?

```
mystery = map(333,0,1000,0,100)
print(mystery)
```



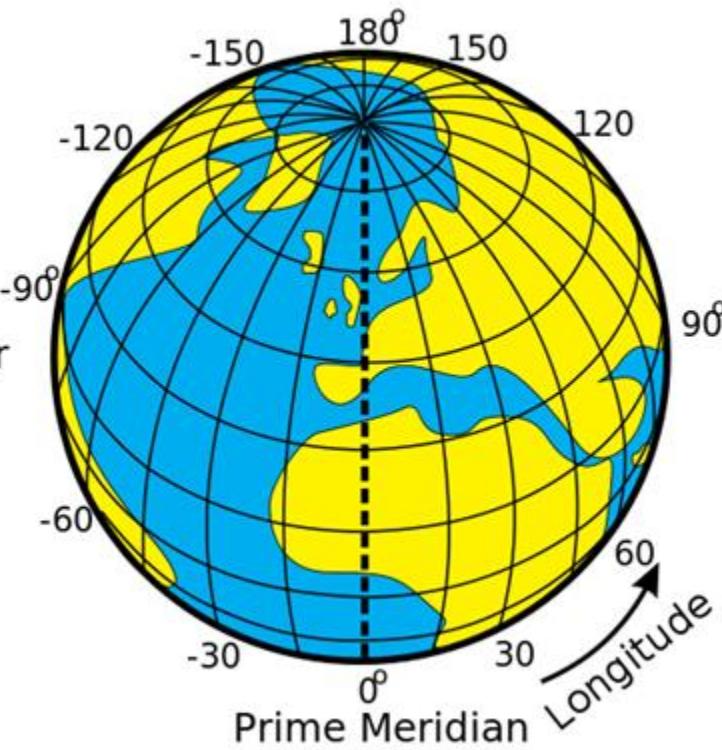
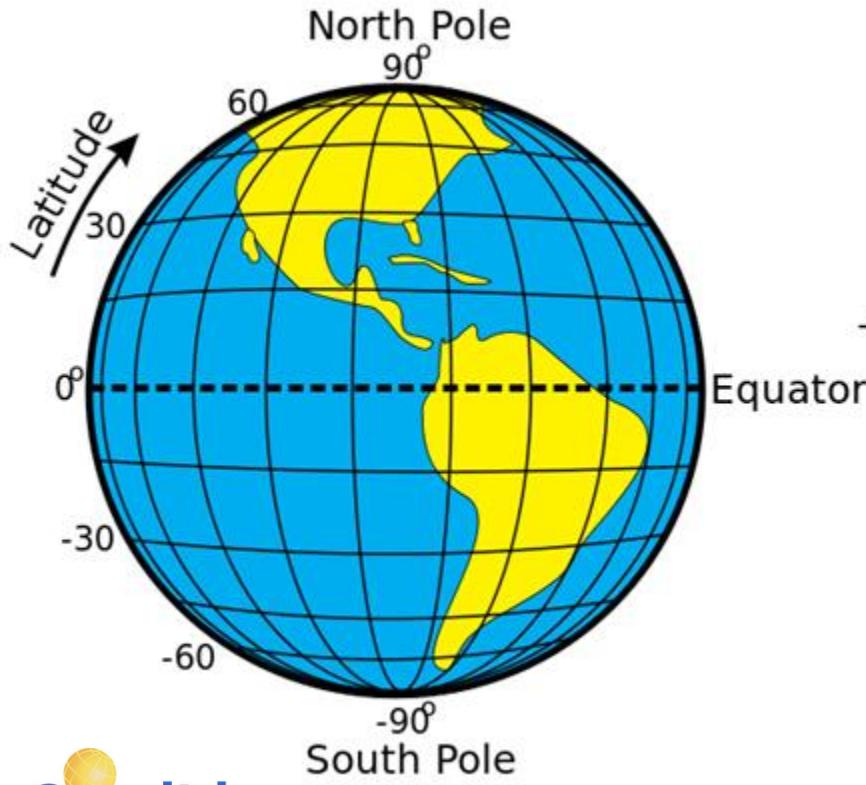
# What would this display?

```
mystery = map(50, 0, 100, 0, 30)
```



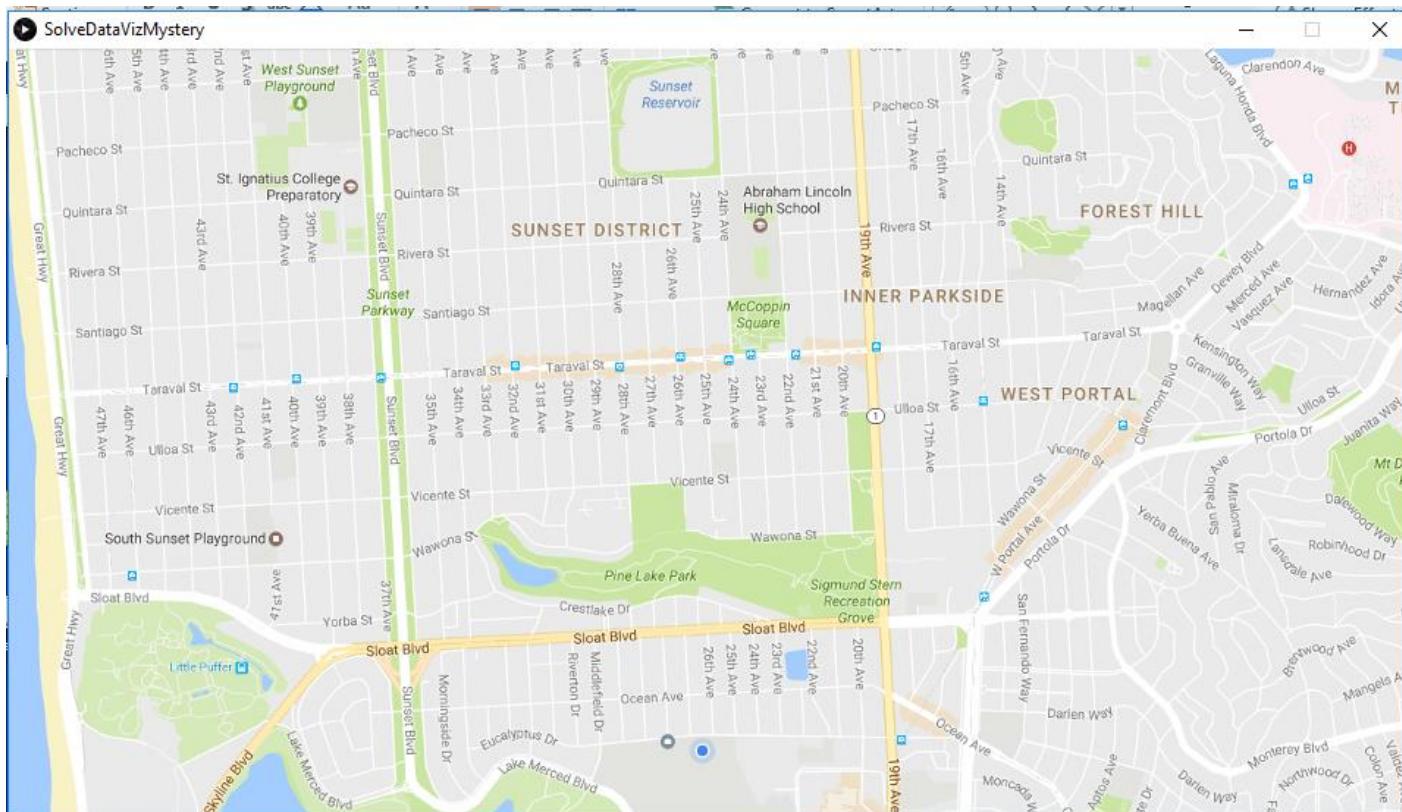
# Latitude and Longitude

- GPS coordinates come in pairs with latitude first followed by longitude.



# Using map() with Latitude and Longitude

- Let's say that I want to put the location (37.747201,-122.503728) on this map



# Using map() with Latitude and Longitude

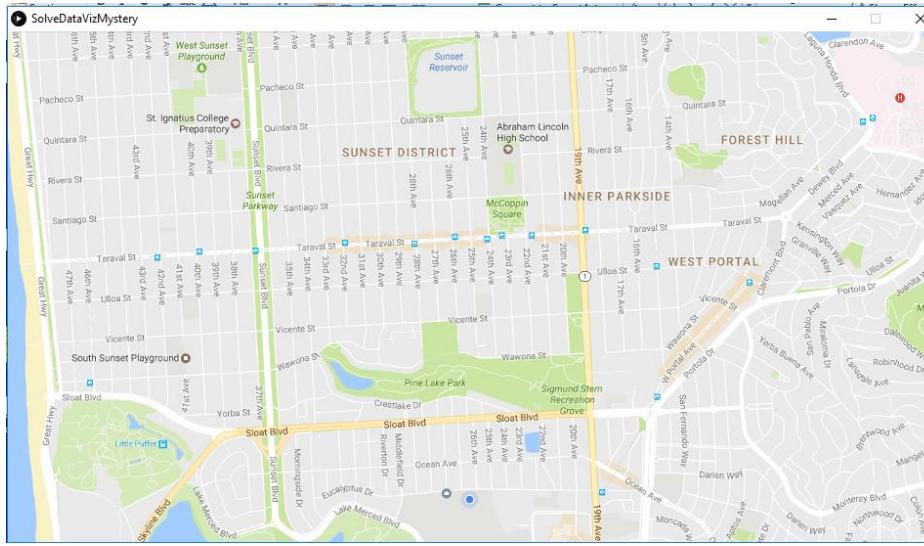
- I need to know the minimum and maximum latitude and longitude on my map.

```
minLat = 37.752242 # minimum latitude in map
```

```
maxLat = 37.728825 # maximum latitude in map
```

```
minLong = -122.509245 # minimum longitude in map
```

```
maxLong = -122.454380 # maximum longitude in map
```



# Using map() with Latitude and Longitude

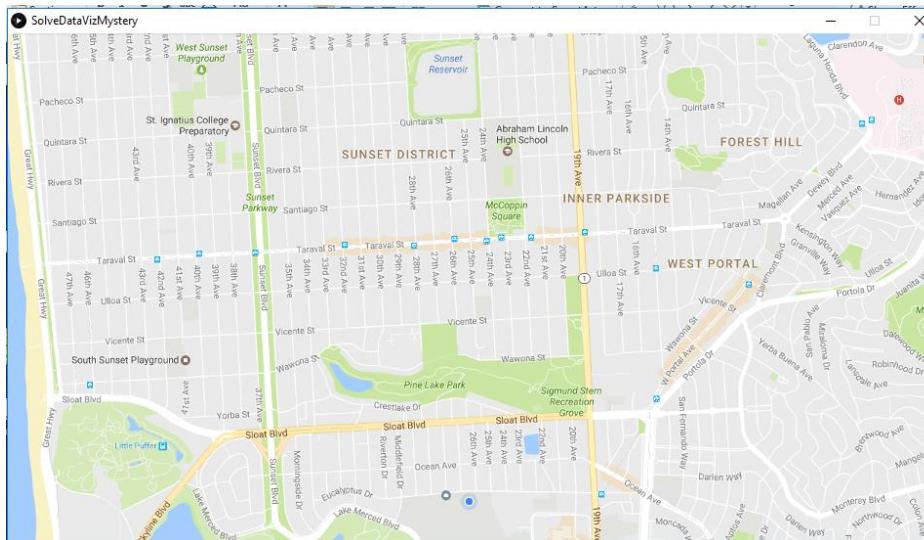
- Latitude goes up and down so it maps to y
- Longitude maps to x.

lat= 37.747201

lng = -122.503728

y = map(lat, minLat,maxLat , 0 , height)

x = map(lng, minLong, maxLong, 0, width)



# X Marks the Spot

- We can use the `text()` function to make an X

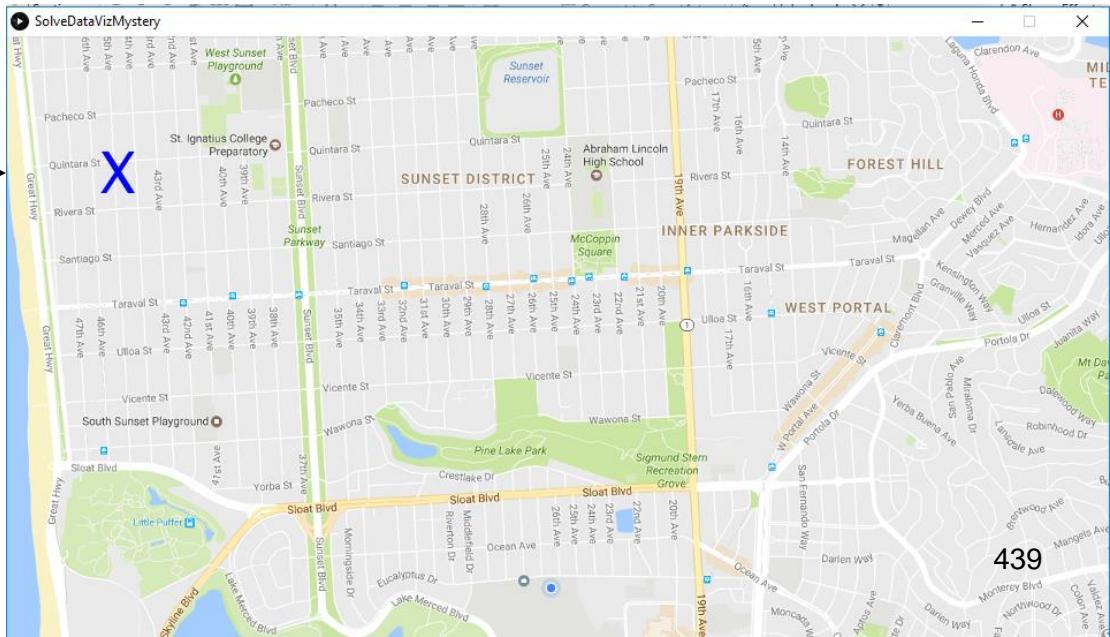
```
lat= 37.747201
```

```
lng = -122.503728
```

```
y = map(lat, minLat,maxLat , 0, height)
```

```
x = map(lng, minLong, maxLong, 0, width)
```

```
text("X",x,y)
```



# Tuples

- Our GPS data in the data visualization problem comes in groups of three numbers in parenthesis.
- In Python that is called a “Tuple.”

```
dat = [(37.747201,-122.503728,"13:00"),
 (37.747201,-122.503728,"13:10"),
 (37.747201,-122.503728,"13:20"),
 (37.747201,-122.503728,"13:30"),
 (37.7457331,-122.5001465,"13:40"),
 (37.7459013,-122.4954998,"13:50"),
 (37.7429473,-122.4875535,"14:00"),
 (37.7430309,-122.4767629,"14:10"),
 (37.7434253,-122.474164,"14:20"),
 (37.7434253,-122.474164,"14:30"),
 (37.7434253,-122.474164,"14:40"),
 (37.7448167,-122.4756721,"14:50"),
 (37.7469956,-122.475964,"15:00"),
 (37.7496281,-122.4764707,"15:10"),
 (37.7486281,-122.4760707,"15:20"),
 (37.7482044,-122.4858554,"15:30"),
 (37.7456825,-122.4985913,"15:40"),
 (37.747201,-122.503728,"15:50"),
 (37.747201,-122.503728,"16:00")]
```

# Tuples

- This code will “unpack” the first tuple into the three variables `lat`, `lng` and `tme`

```
lat,lng,tme =
dat[0]
```

```
dat = [(37.747201,-122.503728,"13:00"),
 (37.747201,-122.503728,"13:10"),
 (37.747201,-122.503728,"13:20"),
 (37.747201,-122.503728,"13:30"),
 (37.7457331,-122.5001465,"13:40"),
 (37.7459013,-122.4954998,"13:50"),
 (37.7429473,-122.4875535,"14:00"),
 (37.7430309,-122.4767629,"14:10"),
 (37.7434253,-122.474164,"14:20"),
 (37.7434253,-122.474164,"14:30"),
 (37.7434253,-122.474164,"14:40"),
 (37.7448167,-122.4756721,"14:50"),
 (37.7469956,-122.475964,"15:00"),
 (37.7496281,-122.4764707,"15:10"),
 (37.7486281,-122.4760707,"15:20"),
 (37.7482044,-122.4858554,"15:30"),
 (37.7456825,-122.4985913,"15:40"),
 (37.747201,-122.503728,"15:50"),
 (37.747201,-122.503728,"16:00")]
```

# Practice Quiz Question

Find the output

```
dat = [(37.747201,-122.503728,"13:00"),
 (37.747201,-122.503728,"13:10"),
 (37.747201,-122.503728,"13:20"),
 (37.747201,-122.503728,"13:30"),
 (37.747201,-122.503728,"16:00")]
lat,lng,tme = dat[4]
print(tme),
print(lng)
```

