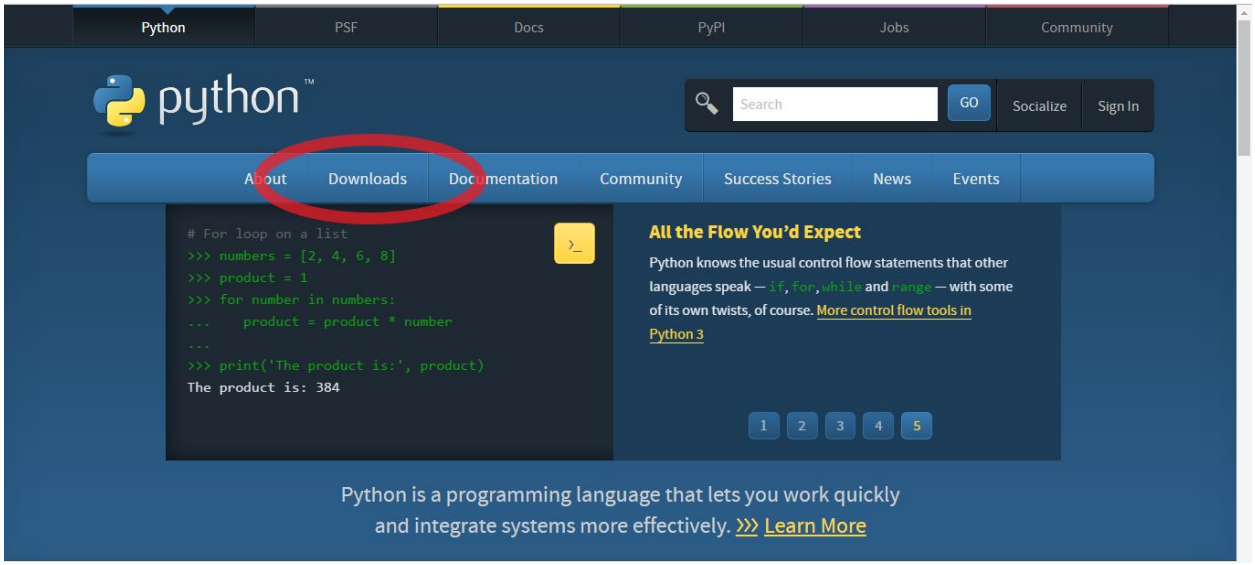
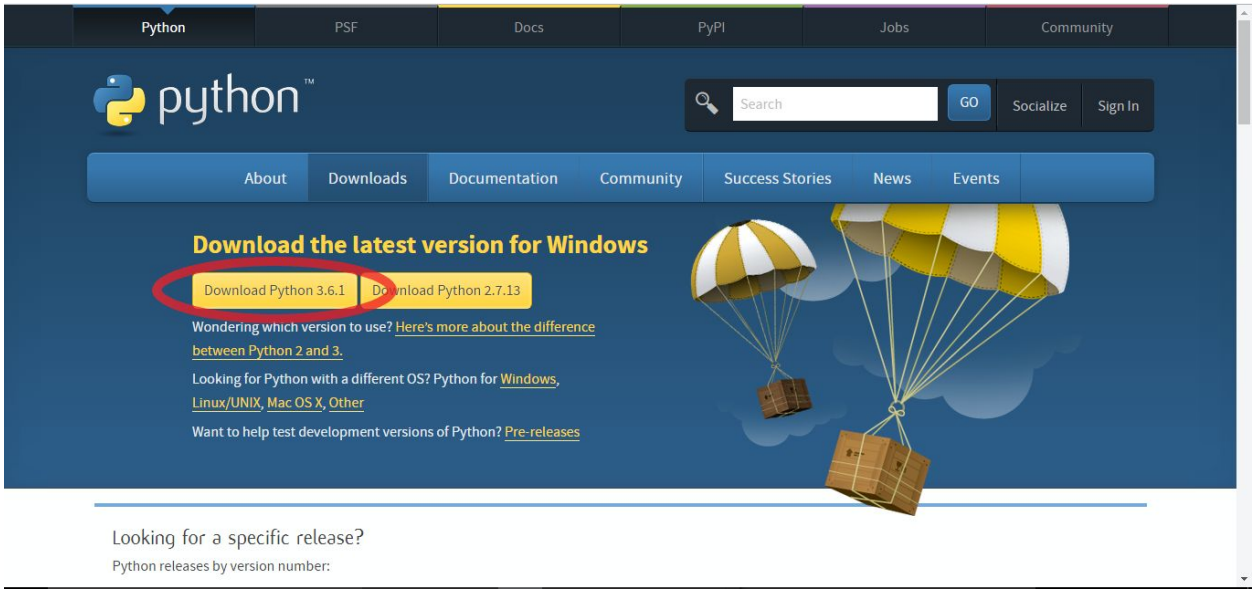
Setting Up Python on Windows

1. Go to www.python.org.
2. Click on “Downloads.”

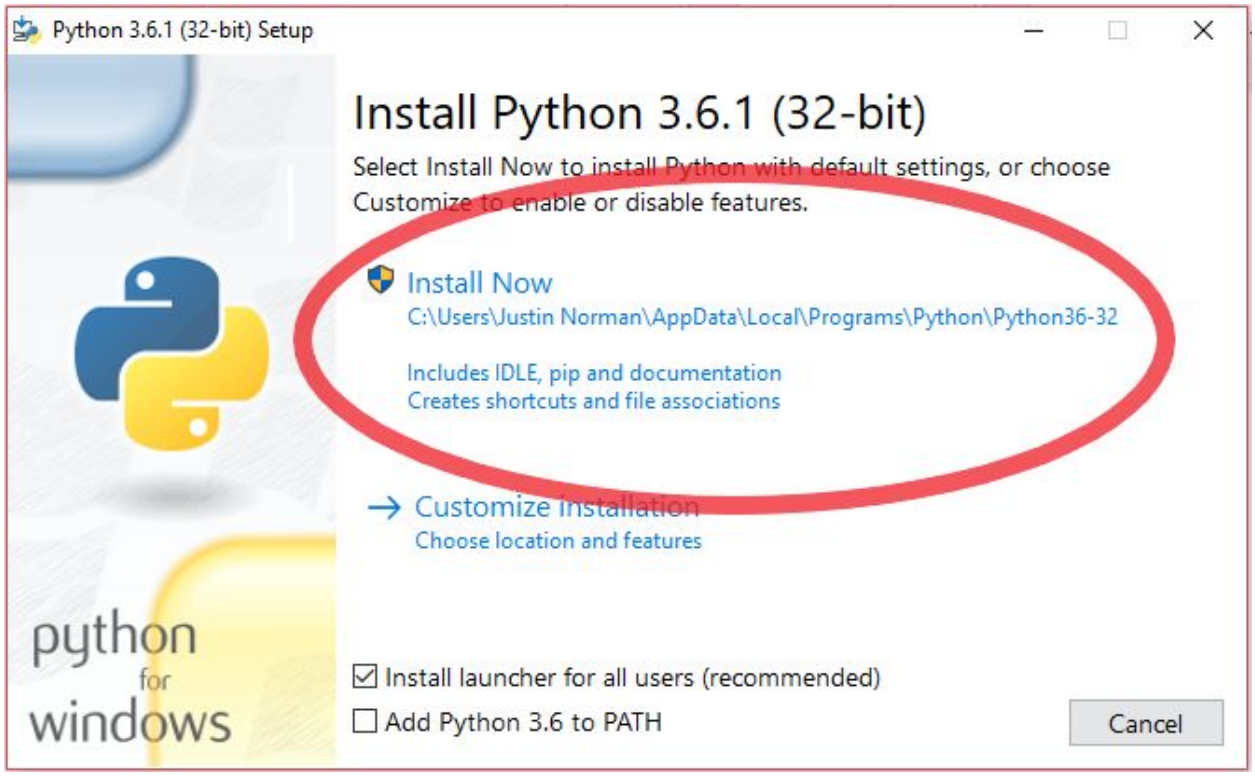


3. Download the most recent Python 3 installer.

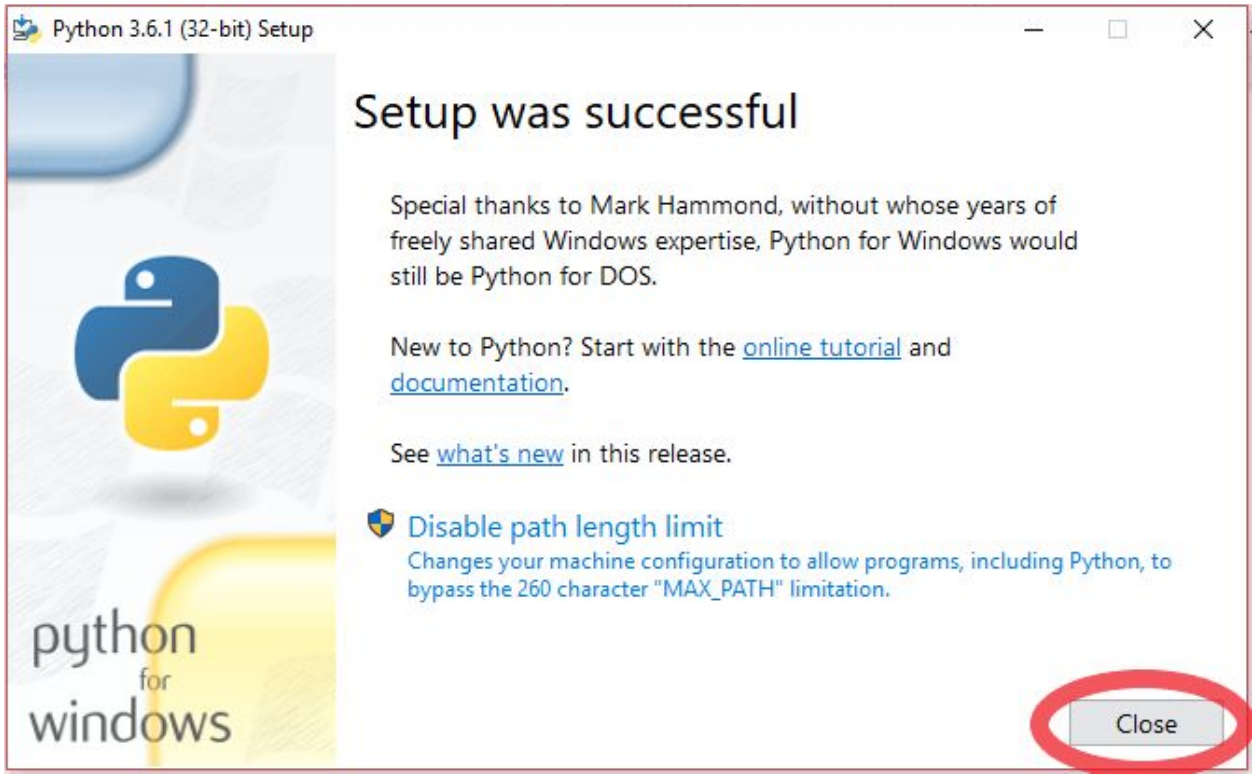


4. Run the downloaded installer.

5. Click “Install Now” and wait for the installer to finish.

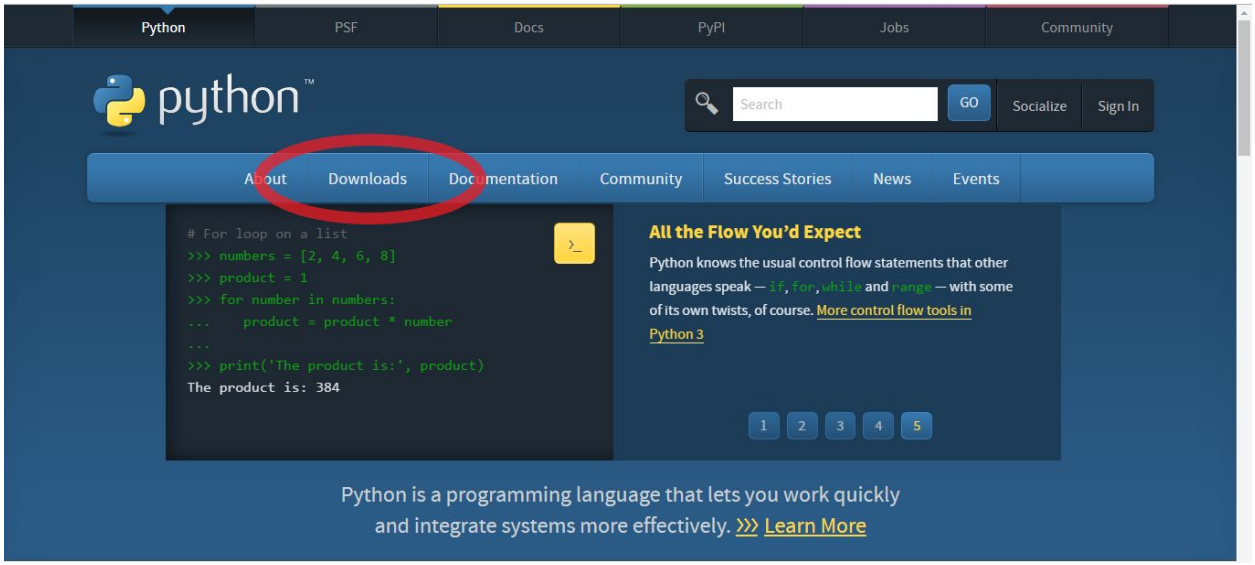


6. Close the installer.

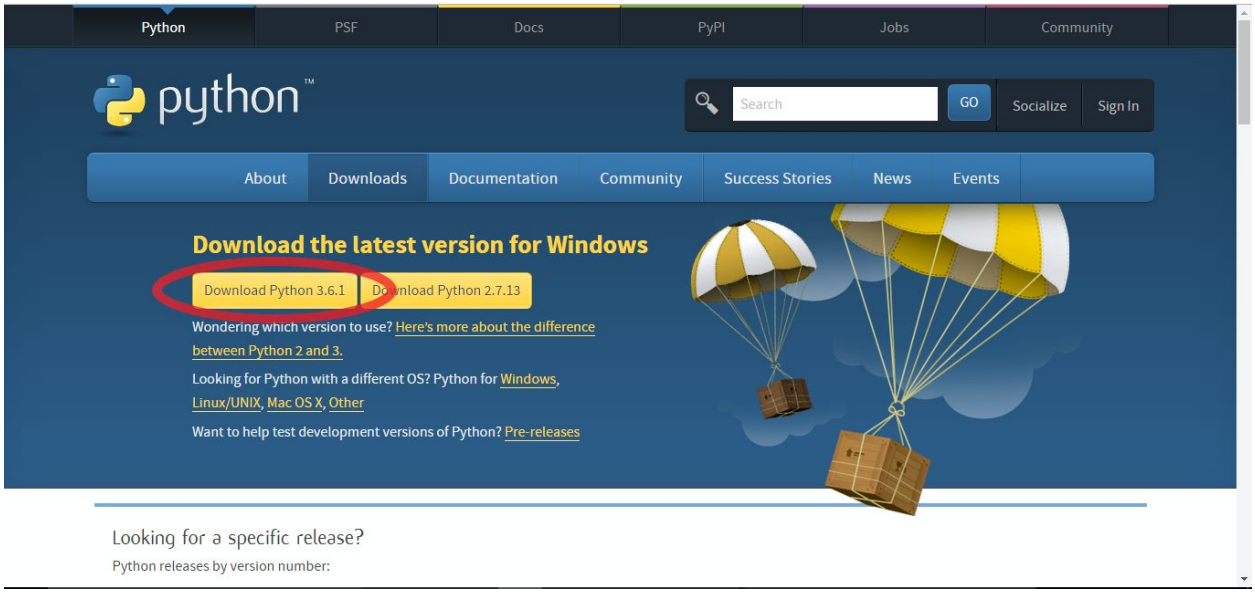

Python 3.6.1 (32-bit) Setup
Setup was successful
Special thanks to Mark Hammond, without whose years of freely shared Windows expertise, Python for Windows would still be Python for DOS.
New to Python? Start with the online tutorial and documentation.
See what's new in this release.
Disable path length limit
Changes your machine configuration to allow programs, including Python, to bypass the 260 character "MAX_PATH" limitation.
python for windows
Close

Setting Up Python on macOS

1. Go to www.python.org.
2. Click on “Downloads.”

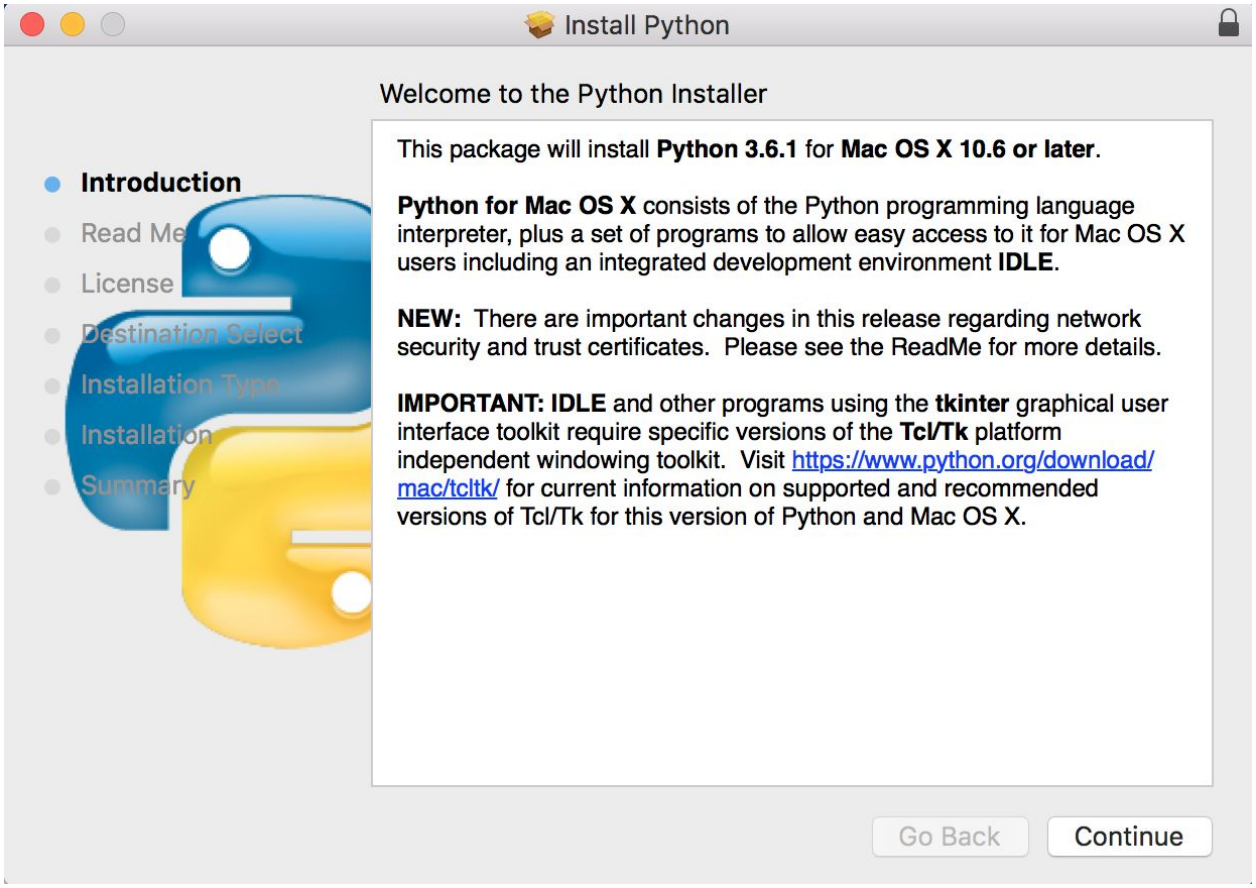


3. Download the most recent Python 3 installer.



4. Run the python.3.x.x-macosx10.6.pkg program in your downloads folder.

5. Click continue and carefully read through the license agreement before accepting it.

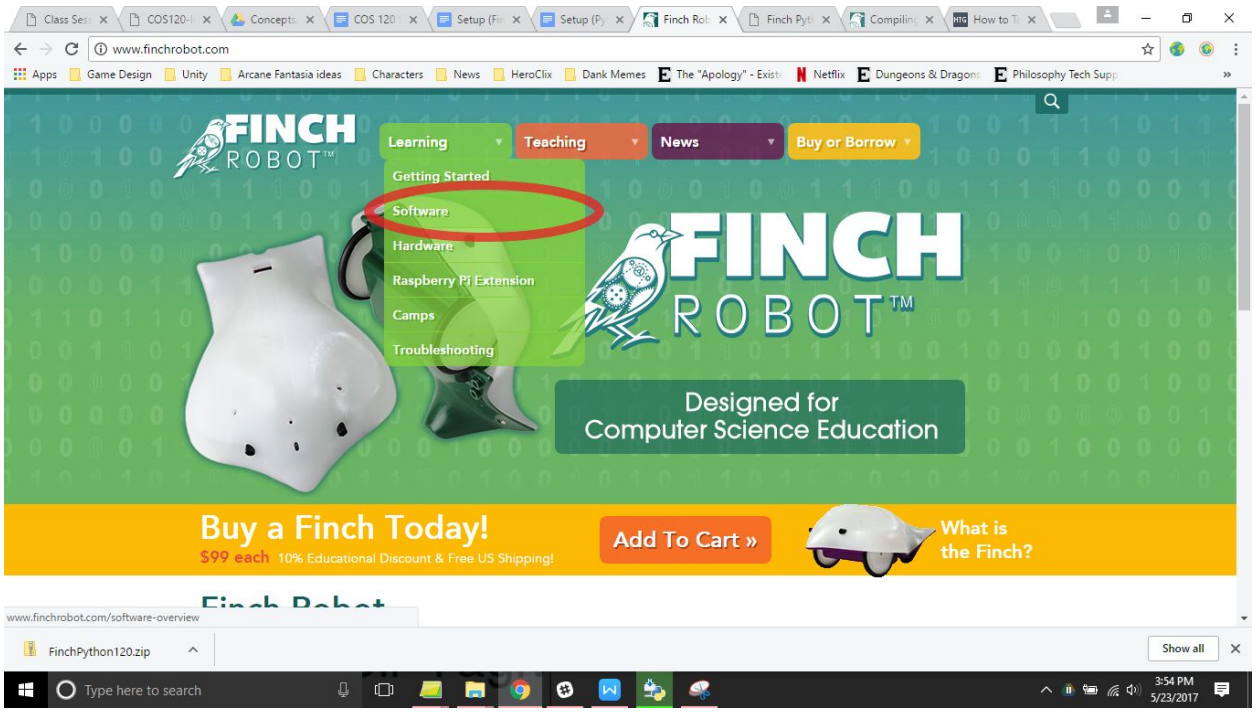


6. Choose to install python on your boot drive (for this example it's Macintosh HD).

7. Click install and enter your password to install the program.

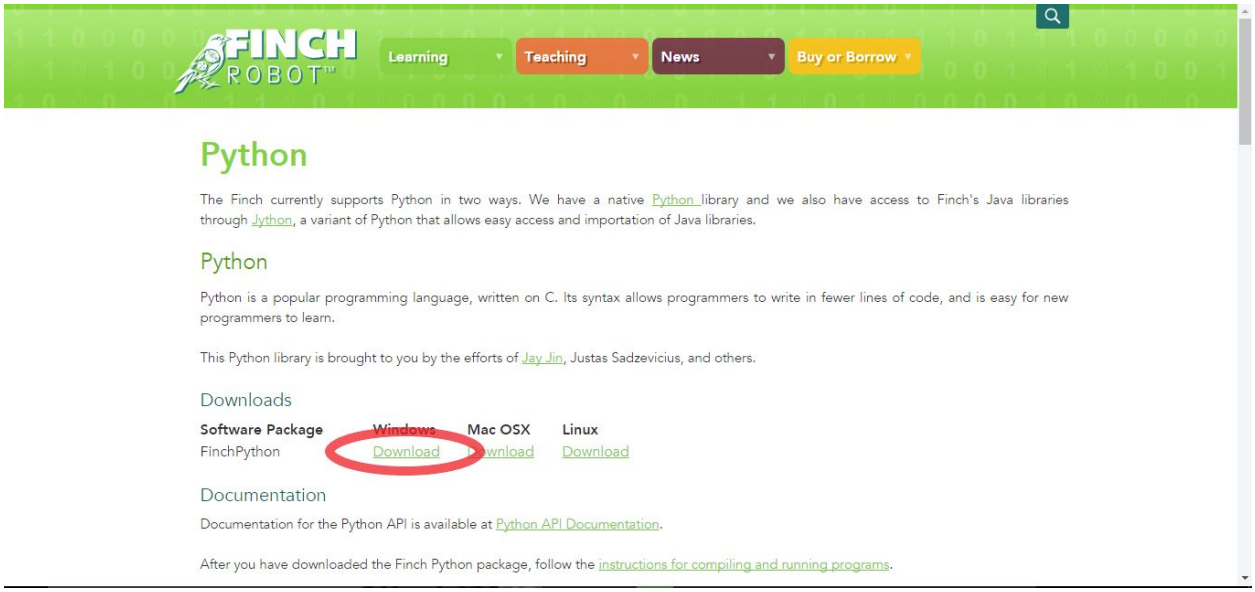
Setting Up the Finch Robot on Windows

1. Go to www.finchrobot.com.
2. Hover over “Learning” and click “Software.”

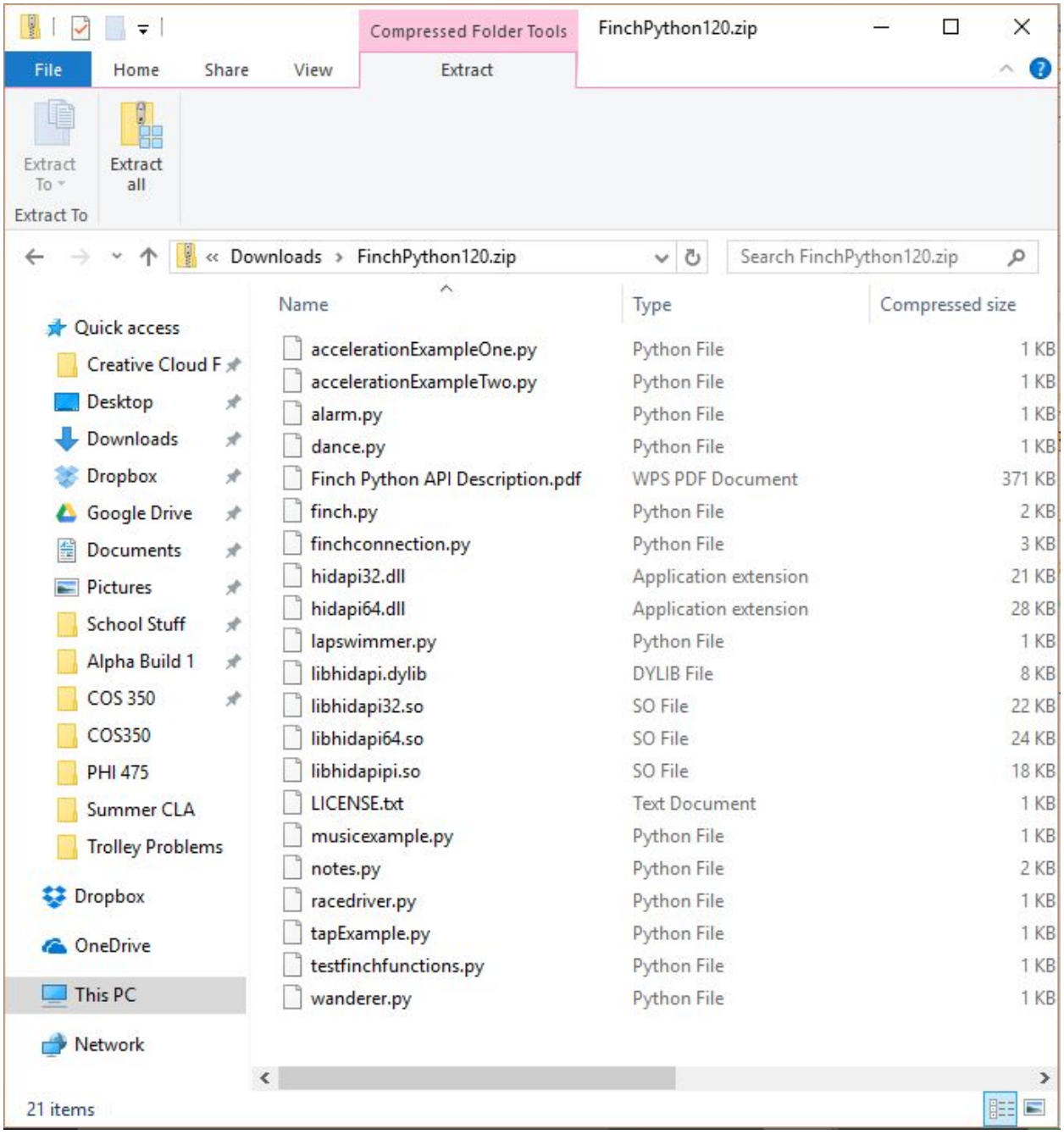


3. Click on “Python/Jython.”

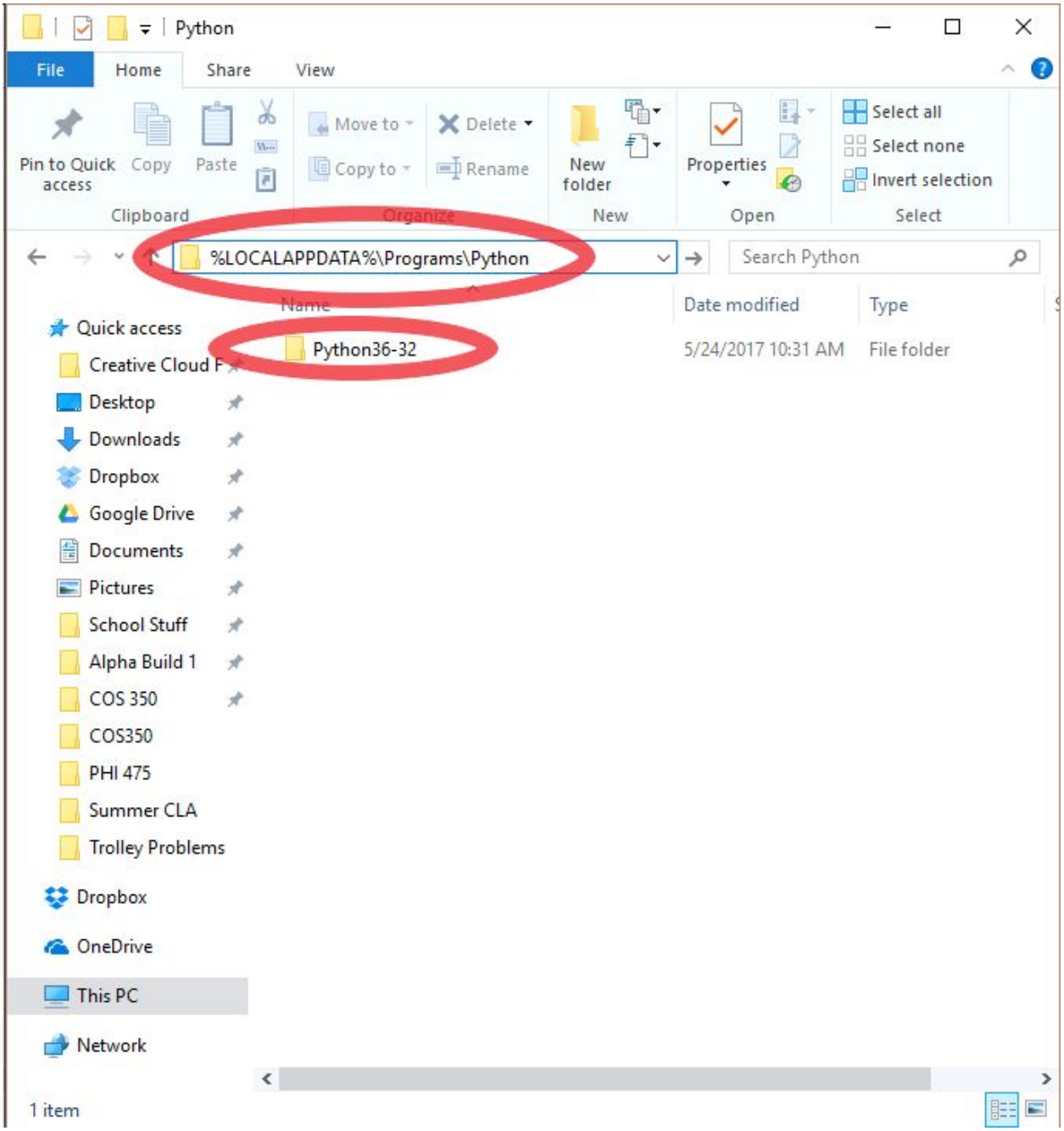
4. Click on “Download” under “Windows.”



5. Open the zip file. The contents should look like this.



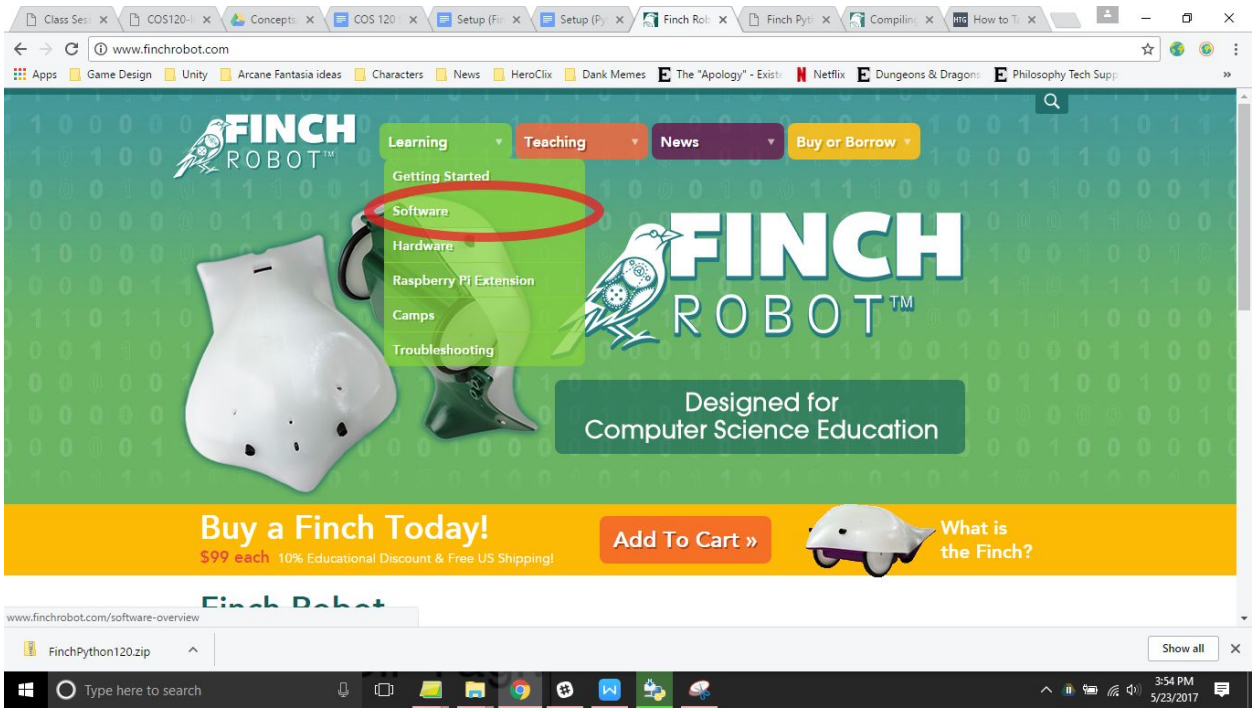
6. Open a new file explorer (WIN + E) and go to the folder that Python is in. This is %LOCALAPPDATA%\Programs\Python\ by default. There will be a single folder in the "Python" folder. Open that.



7. If you are on Windows 10, drag everything from the zip folder to the Python folder. If you are on Windows 8 or 7, extract the contents of the zip file to the Python folder.
8. (Optional) Create a new folder anywhere you'd like and drag everything from the zip folder to that folder.
9. Close the zip folder.
10. Go to https://drive.google.com/open?id=0BxZlk0Jh261pdTBlWGNPQWtrMkk
11. Download the file.
12. Open your downloads and drag "finchAPI.py" from downloads to the Python folder.
13. Close all explorers.

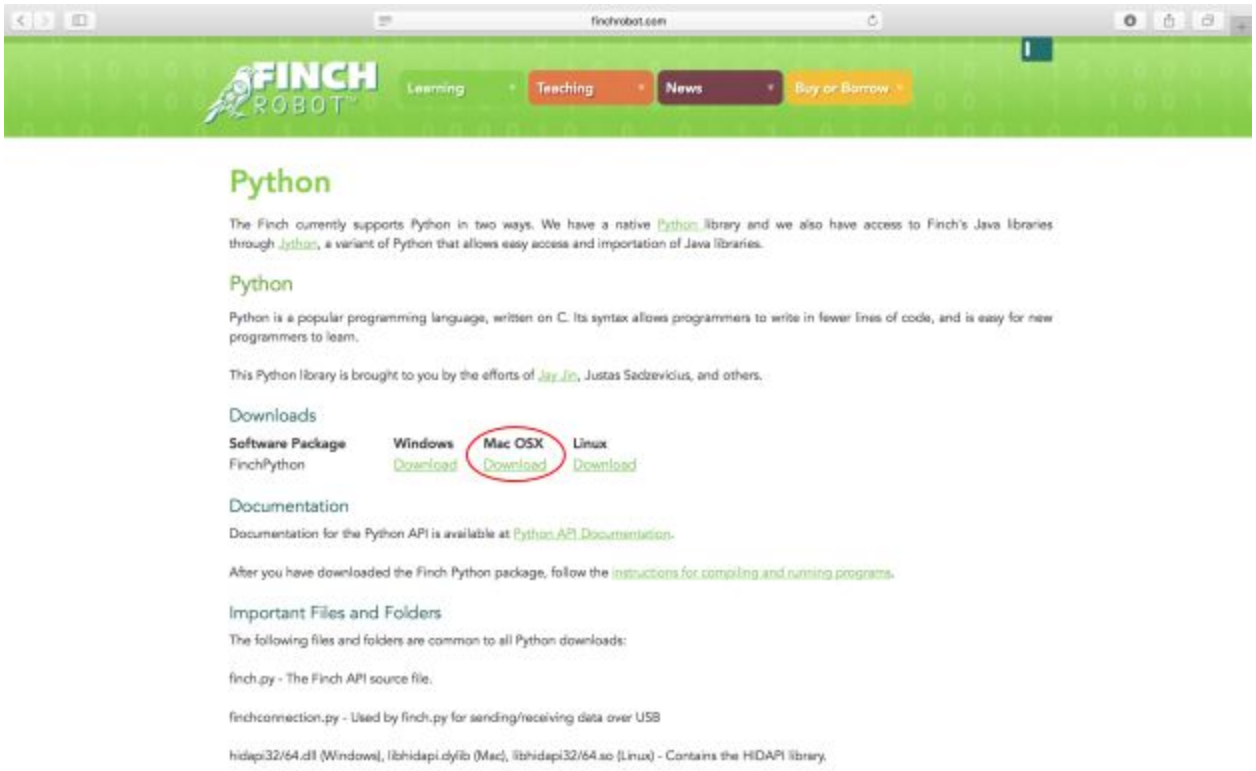
Setting Up the Finch Robot on macOS

1. Go to www.finchrobot.com.
2. Hover over “Learning” and click “Software.”

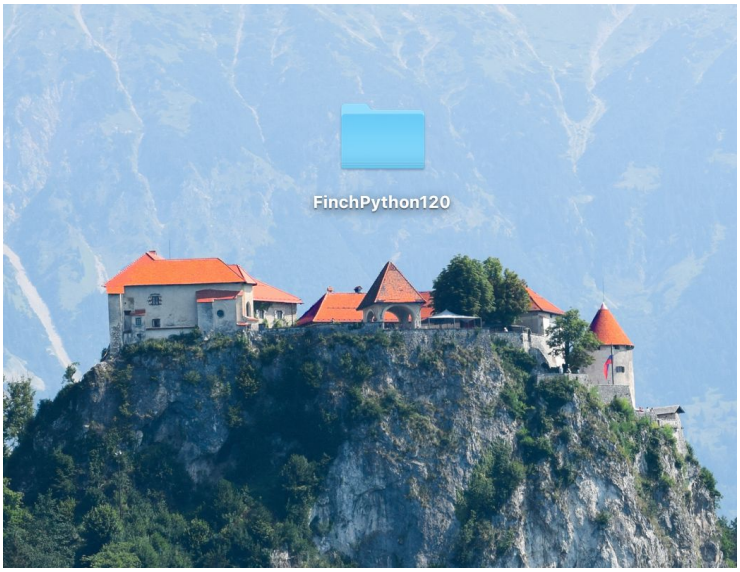


3. Click on “Python/Jython.”

4. Click on “Download” under “Mac OSX.”



5. Remove the downloaded file from your “Downloads” folder, and place it on your Desktop.



6. Go to https://drive.google.com/open?id=0BxZlk0Jh261pdTBlWGNPQWtrMkk, download the file.
7. Drag the file from your downloads to the FinchPython120 folder.
8. Close any finders.

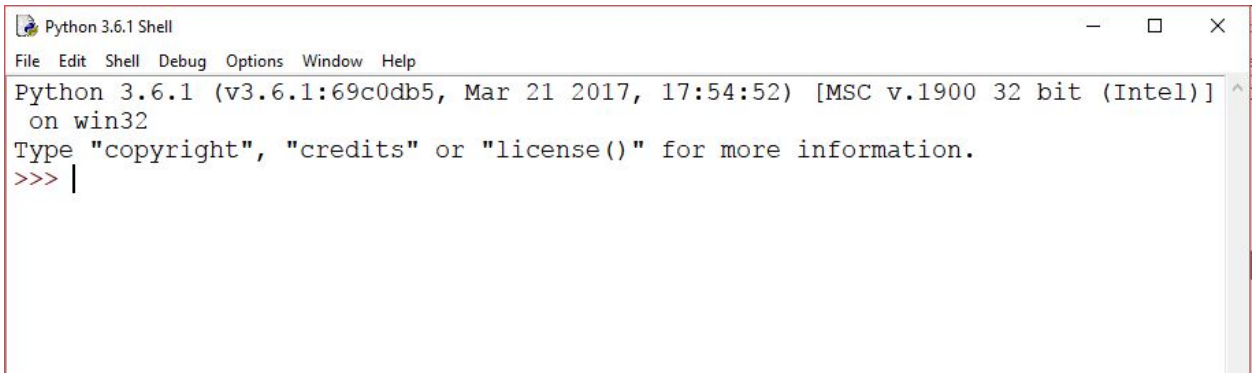
Using IDLE

What is IDLE

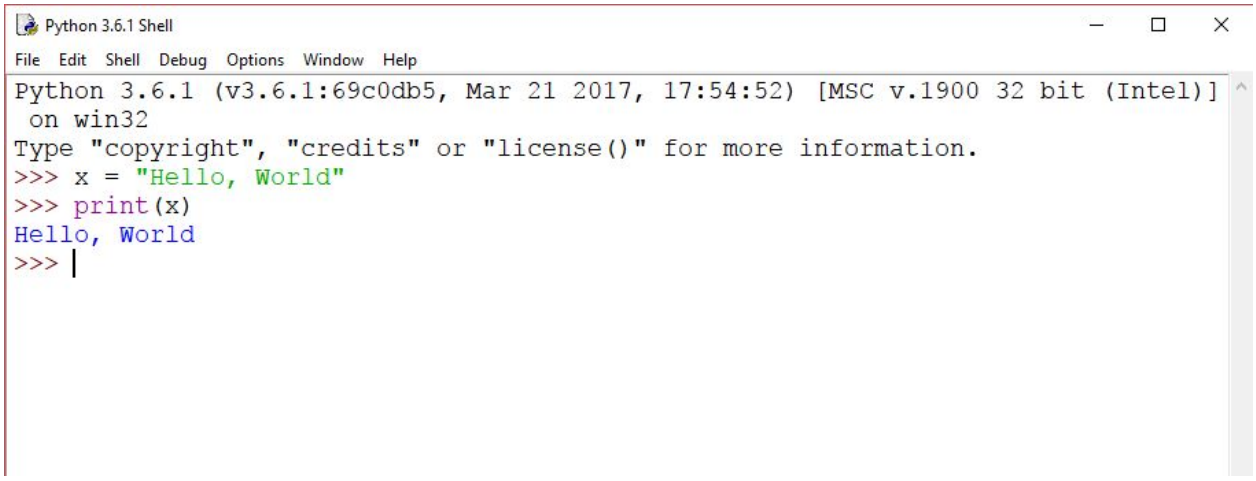
IDLE is the **interactive development environment (IDE)** that is used by default with Python. An IDE is a text editor specifically made for programming. It often comes with other tools, like a shell, a debugger, and a packager. Don't worry, though: IDLE is simple and easy to use.

To open IDLE, go to the folder that Python is in and open the file "pythonw." Or, simply search your computer for it and open the first file that shows up.

Using the Shell



The first window you will see is the **shell**. A shell is a programming interface that allows code to be input line-by-line. The ">>>" indicates that the user can start entering code that the shell will run. It will also maintain any data you put in for the lifetime of the shell.

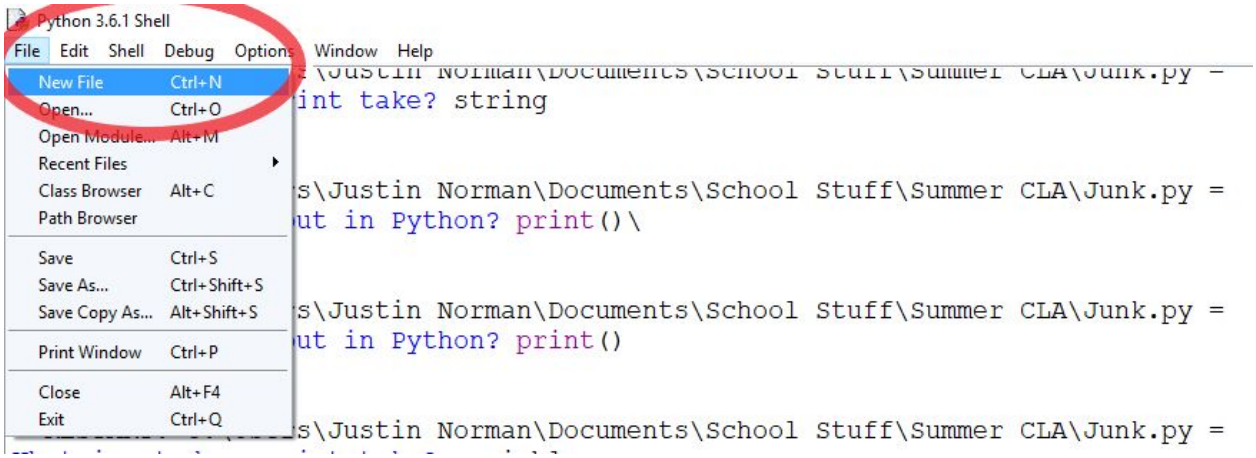


For example, the above code created a variable and printed it out. It's not important that you understand what this is yet. Just remember that the shell can run code line by line. If you are tasked with doing some simple coding, you can just use the shell.

`Try it yourself:` type **`print("Hello, World!")`** into the shell and press enter.

Creating a Program

The shell is useful, but you will want to create entire programs that can be ran again and again. To create a new program, go to the headers and select File -> New File. You can also do this by pressing [Cmd] + [N] on macOS and [Ctrl] + [N] on Windows.



This will open a blank text editor. This is where you will type the code for your programs. You can treat this like a regular file. Anything you would normally do, like open, close, save, save as, etc., can all be done under File. To save the file, you can also press [Cmd] + [S] on macOS and [Ctrl] + [S] on Windows. The standard file extension for Python programs is .py. All Python programs will be named "[name].py".

`Try it yourself:` create a new program in IDLE and save it as "Hello World.py".

Writing a Program

To program, all that you have to do is write text. Of course, this text must be accurate, so it may be hard to read and write. To help you with this endeavor, many IDEs, including IDLE, allow you to put comments in the code. These are explanatory lines of text that tell the reader what certain parts of the code does or how it's intended to work. To insert a comment, place a "#" before the text you mean to comment.

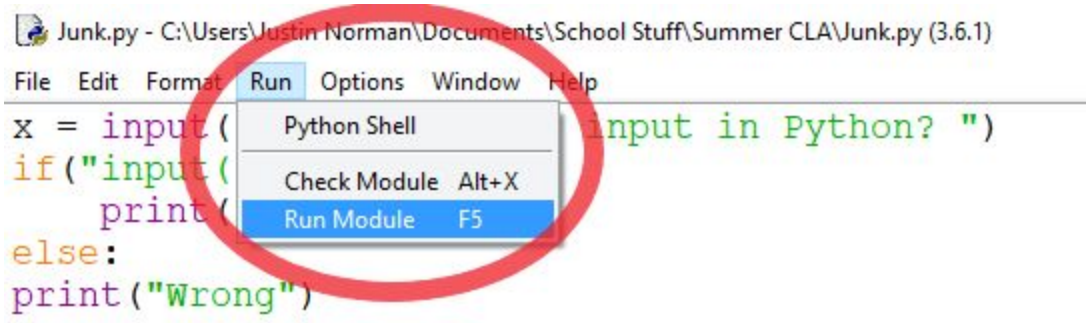
To help you program, IDLE also colors text to indicate its significant. The conventions for commenting are as follows:

- Red: comments
- Orange: keywords
- Blue: definitions
- Green: strings
- Purple: built-in functions, like the print function

`Try it yourself:` write the line **`print("Hello, World!")`** into the IDLE program, next we will run it. As a comment, write **`#print("this will not print!")`** under the line.

Running a Program

Let's say you've created a program, and you want to run it. To run a program, go to the top of the editor and select Run -> Run Module. You can also press [f5] on your keyboard. If your keyboard has a function key (fn, usually), press [fn] + [f5].



`Try it yourself:` run the program you just wrote.

Concepts

Programming is Problem Solving

Programming is the process of creating a solution to a problem. A programmer is a little like a TV chef. The computer, in this case, is represented by the viewer. As the TV chef, you provide an exact instruction set for the viewer to follow in order to create the dish. Instead of using ingredients, detailed descriptions, and cooking implements, programmers use program syntax, comments, and the documentation for the coding language. Every program ends up being a recipe for the computer to follow.

A **program** is a coded behaviour that a computer can execute. Programs are the machines devised by programmers to solve problems. Often, these programs are meant to be applied to specific tasks. In these cases, the program is referred to as an **application**. This course will teach the basics of programming and allow you to create simple programs and applications.

Algorithms

A program is executed by a computer in a step-by-step manner. When these individual steps are combined to carry out a specific task, they produce what is referred to as an **algorithm**. An algorithm is the step-by-step solution to a problem, but it is not necessarily the coded solution. For example, a baking recipe is an algorithm: the recipe is made up of a series of steps, and carrying out each step of the recipe results in a final product (hopefully the correct dish). There are a number of steps that you go through to bake cookies: laying out the dough on a pan, preheating the oven to 350 degrees, placing the pan in the oven, waiting 20 minutes, and pulling the pan out. Of course, this algorithm can be more or less specific. We might say “laying out the dough on a pan” as one of our steps, but the process might actually be more involved, such as laying down tinfoil, pulling apart the dough, and separating them evenly. The important part is that the directions are specific enough so that they can be followed exactly and the problem can always be solved.

Literalism

Before you start programming, you should understand **literalism**. Literalism is an adherence to the explicit meaning of a given text. Computers interpret programs *literally*. This means that computers will only do exactly what you tell them to.

If the computer falters in some way, it is a problem with the program rather than with the computer itself. This is known as a **bug** in the program. Much of programming is really a process of fixing the errors in the code. This process is known as **debugging**.

Often, the prevalence of bugs, and thereby literalism, frustrates new programmers. But, literalism is actually a benefit to programmers. Because of literalism, programmers can trust that the computer will always obtain the correct solution with the correct program; they can be sure that their programs will always function exactly as written. Once a program solves a problem, that program will solve the problem always and everywhere.

Syntax

Programs have to be written in a very specific way. The specific way code is written is known as the code's **syntax**. For example, in some other programming languages every line has to end with a semicolon and every function in programming needs to end with parentheses, Python on the other hand does not use either. If the programmer forgets this, the program will no longer run. These requirements are known as **syntactical requirements**. Different programming languages have different syntactical requirements, just like different spoken languages. For example, in English, the subject of the sentence must always come before the verb, but in Latin, the subject can be placed anywhere in the sentence.

A bug can be a **syntax error**. This type of error is when the programmer mistypes some piece of code. For example, a variable or function might be named **`helloWorld`**, but the programmer could accidentally type **`helloworld`**. The coding language won't recognize it because **`helloworld`** is not actually defined, even though **`helloWorld`** is defined (programming languages are often case-sensitive). Syntax errors are the most common for programmers.

Semantics

Programs are interpreted in a literal way. The interpretation is known as the program's **semantics**. Different pieces of code that perform that same exact task are called **semantically equivalent**. These pieces of code can even be in different languages, but their meaning is identical. Again, this is similar to spoken languages. For example, the English phrase "I came; I saw; I conquered" and the Latin phrase "Veni, vidi, vici" are semantically equivalent because they mean the same thing.

A bug can be a **semantic error**. This type of error is when the programmer codes something with the intention of doing one thing while the code actually means to do another. For example, a programmer might make a piece of code that divides a number by 2, but instead, the program multiplies the number by 2. In this case, the code is syntactically correct (because it completed the task given), but is semantically incorrect

because it did not function as intended. If the code does not behave correctly, but still runs, look out for semantic errors.

Exercises

1. Write an algorithm for the instructor to make a PB&J[1]. Have the instructor carry out the algorithm literally. Debug the algorithm until a successful algorithm is made.

[1] Adapted from http://static.zerorobotics.mit.edu/docs/team-activities/ProgrammingPeanutButterAndJelly.pdf

Hello, World and Variables

“Hello, World!”

The most basic program in any language (Python included) is often considered to be the “Hello, world!” statement. As it’s name would suggest, the program returns the phrase “Hello, World!”. In Python, this is done with the following line of code:

```
print("Hello, World!")
```

Which will display the following on the screen:

```
Hello, World!
```

`print()` will display what is in the parentheses. In this case a **string**; the text encased in quotation marks in the parentheses. A **string**, is a piece of text (letters, numbers, spaces, and punctuation) that the computer stores exactly as you type it, but does not try to understand. The computer knows it is a string because it is enclosed in quotation marks. For example, “Hello” is a string of “H”, “e”, “l”, “l”, and “o”.

Computers can store information in many different ways, but we will only use 4 of them: string, int, float, and bool. We will talk about int, float, and bool more later.

- String: piece of text (letters, numbers, spaces, and punctuation
- Int (Integer): whole numbers
- Float (Floating point numbers): Decimal numbers
- Bool (Boolean): True or False

`Try it yourself`: print some strings.

`print()` can display numbers as well. unlike strings, the computer understands numbers, so we do not need the quotation marks. They can be printed as follows:

```
print(10)
10
```

We can also print out two strings “added” together:

```
print("spam"+"eggs")
spameggs
```

It may seem like it would be easier to put **`"spameggs"`** as the string to print, and in this example it would be, but once we start working with variables, it will become very useful.

`print()` is an example of a **function**, which is code that does some predetermined thing. These will be covered in depth later, but functions will come up early and often during your programming. For now, remember that they are always written as a name followed by parentheses, sometimes with items inside the parentheses. We shall refer to these items as **input**. So, "print" is the name of the function and it writes whatever it supplied to it as input. This is one of the primary examples of program output. It is incredibly valuable for the vast majority of programs.

`Try it yourself`: write a line of code that will print out your name by combining a string for your first name and a string for your last name.

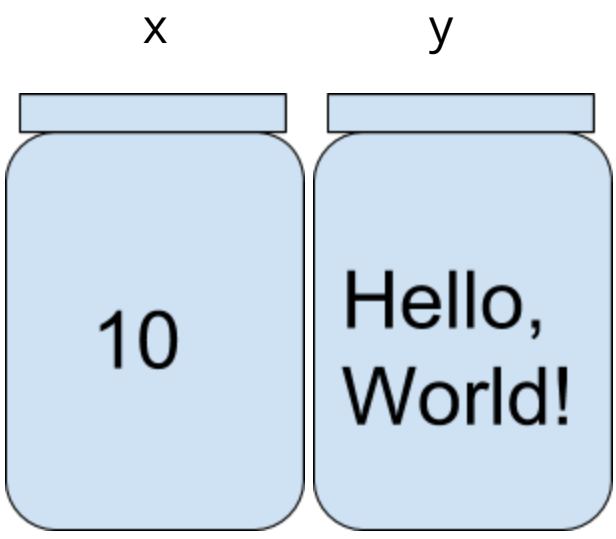
Variables

Another valuable tool for programming is the **variable**. A variable is a representation of a number, string, or other piece of data stored for future use. The representation is a letter or word that is **assigned** the given data with the **=** operator:

```
x = 10
y = "Hello, World!"
```

10 is now stored in x and the string "Hello, World!" is stored in y.

Think of a variable as a jar that you use to put something in:



`x = 10` can be thought of as "10 goes into the jar called **`x`**". In other words, jar **`x`** now contains 10. Similarly, **`y = "Hello, World!"`** is "Hello, World! goes into the jar called **`y`**".

Notice that when Hello, World! is stored in y, it does not contain quotation marks. Also notice that a SINGLE equals sign (=) means ASSIGNMENT: 10 is ASSIGNED to the jar called x, and Hello, World! is assigned to the jar called y. A DOUBLE equals sign (==) is used to denote EQUALITY: 10 == 10.

Variables can be used in place of the data for the same effect:

```
print(x)
10
print(y)
Hello, World!
```

In the program above, the program "prints the contents stored in jar x", which is the integer 10, and then "prints the contents stored in jar y", which is the string Hello, World! Notice that we do not put quotation marks around y when we print it because we used them when we stored it.

Some variables do not contain a string or a number, but instead contain either **`True`** or **`False`**. These are known as **boolean values**:

```
x = True
print(x)
True
```

Booleans are particularly useful as **Flags** (variables that are checked, like a flag on a mailbox) which can be used with **Conditionals** (question statements) and **Loops** (repeats), which will be covered in detail later.

There are also rules for variables in Python:

- The name must start with a letter or an underscore:
  - Both **`hello`** and **`_world`** are valid names for a variable
  - **`10hello`** is not, since it starts with the number 1
- After that, the name may contain letters, numbers and underscores, but only letters, numbers and underscores:
  - **`h3llo_w0rld`** is a valid name
  - **`Hello-world`** is not, since " - " is not a legal character
- Variable names are case sensitive:
  - **`helloworld`** and **`HelloWorld`** will be treated as two different variables
- Variables are assigned left to right:

    - `x = 10` works
    - `10 = x` will give you an error
- Variables can be reassigned at any time with the `=` operator:
    - If you say `x = 5` and then `x = 10`, `x` will be considered to be 10 until you reassign it
- Some words can't be used as variable names, since Python uses them for other purposes.
    - As a general rule, if the word changes color, such as `print`, don't use it as a variable name.

There are also some common conventions for variable names in most programming languages:

- **Constants**, which are variables that never change value while the program is running, are usually written in all caps: **GRAVITY = -9.8**.
- **Variables** that are not constants are usually written in **camelCase** or **with_underscores**. camelCase uses all lowercase letters except when there would be a space, we skip the space, and capitalize the next letter: Player one score would be stored in **playerOneScore** or **player_one_score**.

These conventions make it easier for programmers to read each other's code, but they are not rules of the language. You can break them any time you want, and your code will still run; it is just a good idea to follow them to make your code readable for yourself and others.

Variables can also be assigned as answers to equations:

```
x = 1+1
print(x)
2
```

And even as answers to equations that include themselves:

```
x = 3
x = x*2
print(x)
6
```

While addition, subtraction work the same way that you are likely already familiar with, there are some other different operands that it you should be familiar with:

- To divide, we use 18/6 rather than 18÷6 or 6⌐18

```
print(18/6)

3.0
```

- Multiplication works the same way in Python as you are familiar with, but Python uses the * symbol.

```
print(5*4)

20
```

- Exponents use ** ie. 2**3 = 2^3 = 8.

```
print(2**3)

8
```

- The Modulus operator, also known simply as mod, is represented as x % y. Modulus is basically a remainder operator. It gives the remainder of x / y, while dropping the quotient.
    - 10 % 5 = 0, since 10/5 = 2 R0
    - 11 % 5 = 1, since 11/4 = 2 R1

```
print(10%5)

0

print(11%5)

1
```

- Addition and subtraction work as you would expect: 2+3 = 5, 4-7 = -3

```
print(2+3)

5

print(4-7)

-3
```

- Order of operations works too (good old PEMDAS), but it is a good idea to use parentheses to make things explicit; 2+3*4 is 14, but it is easier to read if you write it as 2+(3*4)

```
print(2+3*4)

print(2+(3*4))

14
```

Try it yourself: use variables and math to multiply your age by your birth month, and divide by your shoe size. Then use the print function to display it on the screen.

Key Points

- Strings are denoted using quotations marks. **"Hello"**, **"Goodbye"**, **"Spam"** and **"4 eggs"** are all examples of strings.
- **print()** displays data on the screen; **print("Hello")** displays "Hello" on the screen.
- Variables store data. Data is assigned to a variable using a single equals sign (**=**) ; **x = 4** and **y = "Hello"** are examples of how data is assigned to and stored in variables.
- Constants are variables that never change their values. Constants are traditionally written in all uppercase letters.
- Boolean values are values assigned to a variable that are either **True** or **False**.

Exercises: Choose and complete one or more of the following exercises.

1. Consider the following code snippets and give the results for each:

```
x = 5 % 3
x = x + 5
x = x - 6
x = x**2
print(x)
```

```
x = 7
y = 9
x = y % x
y = y + x
z = y / x
print(z)
```

2. Use variables to store each of your names (first, middle, and last) and print them out in the following orders:
    a. First, Middle, Last
    b. Last, First, Middle
    c. Middle, First, Last

3. Find and print the slope for the lines created by each pair of coordinates .
   Remember that slope = $\frac{y_2-y_1}{x_2-x_1}$
    a. (1, 4), (2, 6)
    b. (5, 10), (3, 12)
    c. (34.5, 65.2), (81.6, 19.1)

Movement and Other Output

Setting up a Finch program

Before you begin programming the Finch, there are a couple of things you'll need to do before you can start. First, your program will always need the following line at the top of the program:

```
from finchAPI import *
```

This statement brings in the resources that you need. This statement looks up the file "finchAPI.py" and imports all of the code from the file into the current program. Don't worry about understanding the specifics of this: all that you need to know is that it allows you to access many of the resources that you'll need.

When you run a program that has this line at the top, you may get a message that says "Finch is not plugged in". If you see this, check that the Finch is securely connected to your computer and restart the program. You may also get a red error message. It probably means that your program can't find the file "finchAPI.py". Make sure that your program is in the same directory as the file "finchAPI.py". If you have any further questions, refer back to the Setting Up unit.

`Try it yourself`: place `from finchAPI import *` into a program and run it. If you get any error messages, ask your instructor for help. If nothing happens, you are free to move ahead..

Moving the Finch

The first thing you might want to do with any robot is to make it move. To move the Finch, use the functions **`forward()`**, **`backward()`**, **`turnLeft()`**, and **`turnRight()`**. **`forward()`** moves the Finch forward, **`backward()`** moves the Finch backward, **`turnLeft()`** turns the Finch counterclockwise, and **`turnRight()`** turns the Finch clockwise. Each of these will continuously move the Finch once the call is made. The code will continue to execute while the Finch is moving. To stop the movement, use **`stop()`**. **`stop()`** halts both wheels, stopping any movement that is currently happening.

```
forward()    #Moves the Finch forward
backward()   #Moves the Finch backward
turnLeft()   #Turns the Finch counterclockwise
turnRight()  #Turns the Finch clockwise
stop()       #Stops all movement
```

Each of the above functions can also take an argument to specify the amount of movement. **`forward()`** and **`backward()`** take a number of inches to move. **`turnLeft()`** and **`turnRight()`** take an angle in degrees to turn. Unlike before, supplying the inches or angle will make the function wait for the movement to complete. Code will not continue to execute until the movement is finished. This offers an advantage of chaining movement in a simple way. The following code will move the Finch forward 100 inches, then backward 100 inches.

```
forward(100)     #Moves forward 100 inches
backward(100)    #Moves backward 100 inches
```

The speed of the Finch's motors are initially set to ¼ of its full speed. This is to ensure the accuracy of its movement. You can change the speed of the Finch by using **`setSpeed()`**. **`setSpeed()`** takes a single number as an argument, from 0 to 1. At 0, the Finch will not move at all. At 1, the Finch will move at top speed. Note that the real physical issues of moving, like motor force, wheel slippage, friction, and air resistance, results in less actual gain in movement than is programmed. The difference between .75 and 1 is not actually a 25% increase in movement speed.

```
setSpeed(0)      #Not moving at all
setSpeed(.25)    #Moving at 1/4 speed
setSpeed(.5)     #Moving at 1/2 speed
setSpeed(.75)    #Moving at 3/4 speed
setSpeed(1)      #Moving at full speed
```

If you find yourself needing to control each wheel manually, you can use **`setWheels()`**. **`setWheels()`** takes two numbers as input. This is unlike the functions you've seen previously where you only input a single input. Any function that takes multiple inputs must have those arguments separated by commas to tell the function which input is which.The inputs must also be put in a specific order within parentheses. As illustrated by the **`setWheels()`** example functions below, the value for the left wheel is first argument (on the left) and the value for the right wheel is second input (on the right). These values range from -1 to 1. At -1, the wheel is moving backwards at full speed. At 1, the wheel is moving forward at full speed. This also overrides the set speed because it manually

controls the wheels.

```
setWheels(-1, -1)    #Moving backwards at full speed
setWheels(-.5, -.5)  #Moving backwards at half speed
setWheels(1, 1)      #Moving forwards at full speed
setWheels(.5, .5)    #Moving forwards at half speed
setWheels(1, -1)     #Turning right in place
setWheels(-1, 1)     #Turning left in place
setWheels(1, .5)     #Turning right gradually
setWheels(.5, 1)     #Turning left gradually
```

Notice how the first number and the second number have a comma between them. This separates the inputs. The parentheses surround both numbers, which indicates that they are both arguments to **`setWheels()`**. Further, the values are distinct. The combination (1, -1) causes the Finch to turn right while the combination (-1, 1) causes the Finch to turn left. This is because each input operates on a different wheel.

`Try it yourself`: move the Finch in a square. Then, move the Finch in a circle. (Hint: you should be able to use a single line of code to move in a circle). If you feel like you still need more practice, play around with the movement commands.

Lighting the Nose

Lights are useful indicators on any machine. Most modern computers have several lights to indicate if it has low battery, if the battery is currently charging, or if the computer is in sleep mode. The Finch has a single light in its nose that is capable of changing colors. This light may prove useful in providing a visual indicator of what the Finch is currently doing. To control the light, use **`light()`**. This can be used in one of two ways.

`light()` can be supplied a single string that tries to match the word you've put in with a color to display. Light can take any of the following as colors: red, blue, green, cyan, magenta, yellow, and white. You may also supply the string **`"off"`** to turn off the light.

```
light("red")     #turns the light red
light("blue")    #turns the light blue
light("green")   #turns the light green
light("cyan")    #turns the light cyan
light("purple")  #turns the light purple
light("yellow")  #turns the light yellow
light("white")   #turns the light white
light("off")     #turns the light off
```

`light()` can also take three numbers as the red, blue, and green color components of the light, separated by commas and in that order. If you are familiar with the RGB color model, this might be interesting, but the above colors should be sufficient for most applications.

```
light(255, 0, 0)     #turns the light red
light(0, 255, 0)     #turns the light blue
light(0, 0, 255)     #turns the light green
light(0, 255, 255)   #turns the light cyan
light(255, 0, 255)   #turns the light purple
light(255, 255, 0)   #turns the light yellow
light(255, 255, 255)#turns the light white
light(0, 0, 0)       #turns the light off
```

`Try it yourself`: create a program that turns the light blue and moves the Finch forward a foot. When the movement is complete, turn the light red.

Making Some Noise

Like lights, sounds are also useful indicators on machines. While visual cues are excellent, non-invasive ways of alerting people to some event, sounds draw attention to said event while alerting people to the event. Fire alarms operate on this principle (imagine if fire alarms just lit up red). In the context of programming, sounds can be used to signal that some action has taken place within a program. For this purpose, the Finch has a buzzer that can generate sounds with frequencies between 20 to 20000 Hz, which is the audible frequency range for humans (but, you should avoid using sounds above 5000, as those can hurt your ears). To make a sound with the buzzer, use **`buzz()`**.

`buzz()` takes two numbers as arguments. The first number is the duration in seconds of the sound you want to play and the second number is the frequency of the sound in Hertz. These two arguments are separated by a comma syntactically. The following code plays a sound at the frequency of 1000 Hz for 1 second:

```
buzz(1, 1000)
```

`buzz()` does not wait for the sound to finish. Because of this, the Finch may continue moving and otherwise acting as it would. If you want to wait for the sound to finish before continuing, use **`delayedBuzz()`** instead. Otherwise, this works exactly as **`buzz()`**.

```
delayedBuzz(1, 1000)
#Will wait for 1 second
print("Sound is over")
```

The Finch can also play music using it's buzzer. To do so, compose a string of notes separated by spaces. For example, **`"A B C A B C"`**. You can make notes sharp by adding a "#" and make notes flat by adding a "b", like **`"A#"`** and **`"Eb"`**. Then, use **`sing()`**, placing that string and the speed you'd like to play it at as the input, separated by a comma. The speed is how long a note is held for. The Finch will sing the song you've put in at the desired speed.

```
sing("A B C A B C", .25)
sing("A C D C", .5)
sing("C F# Eb", .1)
```

Try it yourself: create a program that plays a 440 frequency sound for a second before moving forward a foot. Once the Finch completes the movement, play a 880 frequency sound for a second.

Simultaneous Behaviour

Most robots will do things simultaneously. They'll be moving, checking sensors, playing sounds, lighting up, and much more, all at the same time. The Finch is also capable of doing this. Already, you've seen that the movement functions keep running code if no distance or angle is supplied. **buzz()** and **light()** also keep running the code while they're in effect. You can combine these to make complicated behaviour, like moving while playing a song.

You will also want to move a set distance or angle while doing other things. For example, you may want to move a certain distance while checking the sensors. To do so, you can add **False** as an input to any movement function that has a set distance, separated by a comma. This simply tells the program not to wait for the movement to finish. It is set to **True** by default, which is why the program will wait by default. Note that if you call another movement after a movement function with **False**, it will override the previous one. You can check if the Finch is currently moving by using the flag **isMoving**. **isMoving** will be **False** whenever the Finch is not executing a movement function and will be **True** whenever the Finch is executing a movement command.

```
print(isMoving)      #Will be False
forward(100, True)   #Will wait for the movement to finish
print(isMoving)      #Will be False BECAUSE THE LAST MOVEMENT WAITED
forward(100, False)  #Will not wait for movement to finish
print(isMoving)      #Will be True because the Finch is still moving forward
```

If you're doing simultaneous behaviour and need to stop everything, use **halt()**. **halt()** will stop everything the Finch is currently doing, stopping all movement, turning off the light, silencing any noise, etc. This can be useful for an emergency measure.

The use of this will become much more apparent when we start getting data from sensors continuously. Then, you'll be able to make the Finch respond to its surroundings while doing other things. For now, know that this is possible and will be revisited later.

Try it yourself: make the Finch move three feet while playing a song.

Key Points

- To move the Finch, use **`forward()`**, **`backward()`**, **`turnLeft()`**, and **`turnRight()`**. Each can take a number for a distance in inches or an angle to turn. When using a distance or angle, they an also take a boolean value which determines if the program should wait for the movement to finish. This is **`True`** by default, but adding **`False`** makes the program continue during movement. You can stop any movement currently happening using **`stop()`**.
- To light up the Finch's nose, use **`light()`**.
- To use the buzzer, use **`buzz()`**. If you want to wait for the sound to finish, use **`delayedBuzz()`**. If you want to play a song, use **`sing()`**.
- To stop everything that the Finch is doing, use **`halt()`**.

Exercises

1. Move the Finch in a triangle. Before a turn, slow down the Finch's movement, make the Finch play a short low noise and make the nose red. Before moving forward, speed up the Finch's movement, make the Finch play a short high noise and make the nose green. It is best to do this in small parts, tackling the movement first, then putting in the sounds and lights.

Text Input and Conditionals

Text Input

Many programs allow the user to enter information, like a username and password. Python makes taking input from the user seamless with a single line of code:

```
input("How do you get input in Python? ")
```

`input()` will print the string you put in the parentheses and wait for the user to respond. Often, you will use a question as the input string, followed by a space to separate the user's text from the output text. In this example, the program will print "How do you get input in Python? " and user can then enter text to respond.

```
How do you get input in Python? By using input()
>>>
```

The user responded with "By using input()," but right now, that response isn't doing anything. To catch the response and do something with it, we need to set that to a variable.

```
x = input("How do you get input in Python? ")
print(x)
```

Now, the variable **`x`** stores the response from the user.

```
How do you get input in Python? By using input()
By using input()
```

`Try it yourself`: ask for your name and store it in a variable, then print the variable.

Strings and Numbers

Suppose you ask the user for a number, such as a pin number. **`input()`** gives you the user's response. However, it doesn't determine whether or not the user entered a number. Instead, it just gives you a string. So, you can sometimes get errors if you try to treat the input as a number immediately. If you run the following program, Python will give you an error.

```
x = input("Enter a number: ")
print(x + 1)
```

```
Enter a number: 1
Traceback (most recent call last):
  File "C:\Users\Justin Norman\Doc
5, in <module>
    print(x + 1)
TypeError: must be str, not int
```

This is because Python can't add a string and a number together, and **`input("Enter a number: ")`** is a string. Use the **`int()`** function if you want to treat the input as an integer or the **`float()`** function if you want to treat the input as a number with a decimal point. If you are in doubt, just use the **`float()`** function.

```
x = input("Enter a number: ")
x = int(x)
print(x + 1)
```

```
Enter a number: 1
2
```

To convert a number to a string, use **`str()`**.

```
x = 1
x = str(x)
print("This is now a string: " + x)
```

```
This is now a string: 1
```

`Try it yourself:` ask for your age, then print what your age will be in one year.

Truth Statements

Once you have a response from the user, you'll want to tell if that response is correct or not, such as a password that was entered or a button that was pressed. To do so, you'll need to check a property of the response. In the above example, you'll want to check if the user enters the correct answer by making sure "input()" is somewhere in the response. To do this, you'll need to use a **truth statement** or **boolean expression**. A truth statement is a statement that evaluates to, or becomes, a boolean value. Both truth statements and boolean values use the program to check if a statement is either **`True`** or **`False`**. Here is an example of a truth statement:

```
x = 1
y = 1
print(x == y)
```

Since **x** is equal to **y**, the program displays **True**.

x == y is a truth statement. The code **==** checks whether or not **x** is equal to **y**. This is distinct from **x = y**: **x == y** checks if **x** and **y** have the same value, where **x = y** sets the value of **x** to the value of **y**. If **x** and **y** have the same value, **x == y** will have the value of **True**. If they are not, it will have a value of **False**. In the example above, **x** has the same value as **y**, so the output is:

```
True
```

There are several other types of truth statements, like:

- **x > y** - is true if x is greater than y
- **x >= y** - is true if x is greater than or equal to y
- **x < y** - is true if x is less than y
- **x <= y** - is true if x is less than or equal to y
- **x != y** - is true if x is not equal to y
- **x in y** - is true if x is somewhere in y

These work for more than just numbers. Often, they'll work for any data type they make sense for. Just use your intuition: **==** can check if two strings are the same, **<** can check if one string comes before another alphabetically, **>** can check if one string comes after another alphabetically, and so on. When in doubt, experiment using the shell.

```
>>> "Hello" < "Pizza"
True
>>> "Fries" > "French"
True
>>> "I am" == "Am I"
False
>>> "Pizza" in "Pizza Pie"
True
```

You can also combine boolean values, which can be truth statements, using the following keywords:

- **x or y** - is true if **x** is **True**, **y** is **True**, or both **x** and **y** are **True**
- **x and y** - is true if both **x** and **y** is **True**
- **not x** - is true if x is not true

```
>>> True and False
False
>>> True or False
True
>>> not True
False
```

For example:

```
x = input("Pick a number: ")
x = int(x)
print("It is " + str(x > 3 and x < 7) + " that the number is between 3 and 7.")
```

```
Pick a number: 4
It is True that the number is between 3 and 7.
```

Using a truth statement, we can check if the user's response is correct:

```
x = input("How do you get input in Python? ")
print("input()" in x)
```

```
How do you get input in Python? By using input()
True
```

Beware! Keep track of your syntax. A common mistake for programmers is to use **=** instead of **==**. **`x = y`** assigns the value of **`y`** to **`x`**, but **`x == y`** checks if **`x`** and **`y`** have the same value. Using the wrong one can break your program. If you use **`x == y`** when you mean **`x = y`**, the variable will not be assigned properly.

```
x = 1
y == x
print(y)
```

```
Traceback (most recent call last):
  File "C:\Users\Justin Norman\Docu
Junk.py", line 17, in <module>
    y == x
NameError: name 'y' is not defined
```

If you use **`x = y`** when you mean **`x == y`**, the code may break or flags will be assigned improperly.

```
x = 1
flag = x = 1
print(flag)
```

```
1
```

Be sure you're using the right equals sign!

`Try it yourself:` write a program that asks for your age, and prints **`True`** if you are at least 16 years old.

Conditionals

We can now determine whether the user's answer is correct, but now we want to do something that depends on it. For example, you only want the user to log in if the password is correct. You can do this using a **conditional**. A **conditional** is a command that takes a **`True`** or **`False`** value, which can come from a truth statement, and does something depending on that. This is also known as an **if statement**.

```
x = input("How do you get input in Python? ")
if "input()" in x:
    print("Correct!")
```

The syntax for an if statement is an **`if`** followed by any boolean value, which may be a truth statement, and a colon. The colon after the if statement indicates that a **block** follows the statement. A block is a piece of code that is organized together and offset from the rest of the code. In Python, blocks are indicated by indentation, so all the code in block after an if statement is indented once. A block is usually executed contingently, based on some other factor in the code. For if statements, the block of code is executed only if the boolean value in the if statement is **`True`**. In this case, **`"input()" in x`** is **`True`** only if **`"input()"`** is somewhere in **`x`**. The line **`if "input()" in x:`** checks the truth statement. If it is **`True`**, then it executes the block of code beneath it, which contains only the line **`print("Correct!")`**.

```
How do you get input in Python? By using input()
Correct!
```

Although this example only uses a single line for a block, blocks can contain any number of lines, so long as they are all indented. For example:

```
if True:
    print("Block line 1")
    print("Block line 2")
    print("Block line 3")

Block line 1
Block line 2
Block line 3
```

You will also want to tell the user if his answer was incorrect. You can do this by placing an **`else`** directly after the indented code of the **`if`**. The syntax for the **`else`** also needs a colon after it followed by a block of code, like an **`if`**. As with all blocks, it must be indented. The block beneath **`else`** will run if the condition of the **`if`** it is placed after is **`False`**.

```
x = input("How do you get input in Python? ")
if "input()" in x:
    print("Correct!")
else:
    print("Wrong!")

How do you get input in Python? By using print()
Wrong!
```

We now have a working program that asks the user a question and determines whether they got the right answer! But before you get too carried away, make sure you remember that indentation is important. If you don't indent correctly, the whole program may crash or run improperly. For example:

```
x = input("How do you get input in Python? ")
if "input()" in x:
    print("Correct!")
else:
print("Wrong!")
```

This code won't run because Python needs an indented statement after an **`if`** or an **`else`**. If the code did run, "Wrong!" would be printed regardless of the boolean value of the truth statement. Consider another example:

```
x = input("How do you get input in Python? ")
if "input()" in x:
    print("Correct!")
else:
        print("Wrong!")
```

This code won't run because the code run by the **`if`** statement must only be indented one more than it. In this case, the **`else`** is indented 0 times and the **`print("Wrong!")`** is indented 2 times, so Python will give an error. Consider one more example:

```
x = input("How do you get input in Python? ")
if "input()" in x:
    print("Correct!")
else:
    print("Wrong!")
print("Try again!")
```

You may want "Try Again!" to print only if the user entered the wrong answer. However, it will always print because it isn't indented.

```
How do you get input in Python? By using input()
Correct!
Try again!
```

So, always be careful with indentation!

`Try it yourself`: write a program that asks for a color and tells you if that color is the same as the your favorite color.

Multiple Conditionals

You may want to check the input against different conditions. With a security program, there may be several correct passwords, all of which are different from each other and take you to different places. You can check this using multiple conditionals. Here is an example:

```
x = input("What is the password? ")
if "Hello" in x:
    print("Hi")
elif "Secret" in x:
    print("Shhhh!")
elif "Flim" in x:
    print("Flam!")
elif "Zim" in x:
    print("Zam!")
else:
    print("That's not the password...")
```

After a password is entered as input, a unique message is displayed on the screen. To have these unique messages display, we use an **`elif`**. **`elif`** is a condensed way of writing "else if" and runs a conditional **`if`** statement if the previous conditional statement was **`False`**.

In the program above, let's say the user enters "Secret" as the input. After the input is entered, the program will check the first if statement. A good technique to guide yourself through programming conditionals is to think about it as though you were speaking aloud:

> `if` "Hello" is the input:
>
> Print "Hi"
>
> Otherwise (`else`), if "Secret" is the input:
>
> Print "Shhhh!"

Since “Secret” is the input, the program prints “Shhhh!”

elif must be placed after an **if**, or an **elif**, and you can use as many as you want within a block of code. Just like an **if**, an **elif** does not need to have an **else** after it (although it can). After the **elif** comes a boolean value (truth statement) placed inside of parentheses, with a colon placed after the parentheses. The block of code placed after the **elif** statement is then executed so long as the truth statement holds true. The structure of an **elif** statement is similar to an **if** statement:

```
elif (truth statement):
     …..
     Code to be executed
     …..
```

```
What is the password? Hello
Hi

What is the password? Secret
Shhhh!

What is the password? Flim
Flam!

What is the password? Zim
Zam!

What is the password? Foo
That's not the password...
```

Else if is different from having a second **if** statement, specifically because if the first conditional **if** statement is true, the **elif** statement will not run. Take for example the program below.

```
x = input("What is the password? ")
if "Hello" in x:
    print("Hi")
if "Secret" in x:
    print("Shhhh!")
if "Flim" in x:
    print("Flam")
if "DeepThought" in x:
    print("Fourty Two")
else:
    print("That's not the password...")
```

Using the input “Hello Flim” would produce a different output than the first program:

```
What is the password? Hello Flim
Hi
Flam
That's not the password...
>>>
```

Because **`elif`** statements were not used, the **`print`** statements for both “Hello” and “Flim” were output. Also, because **`else`** only looks at the previous **`if`** statement, if “DeepThought” is not entered, it will always print “That’s not the password…”.

`Try it yourself:` ask for a name and tell the user if that’s one of the programmer’s first, middle, or last names.

Nested Conditionals

You can put conditionals inside each other to allow for more branching behaviour. Many programs will layer security by requesting a password, then a pin, then a security question. This is known as **nesting**. To nest a conditional, simply place an if statement one more indentation under another if statement.

```
x = input("Pick a number between 0 and 10: ")
x = int(x)
y = input("Pick a number between 0 and 10: ")
y = int(y)
if x < y:
    if y - x > 5:
        print("The difference is greater than 5")
    else:
        print("The difference is less than 5")
else:
    if x - y > 5:
        print("The difference is greater than 5")
    else:
        print("The difference is less than 5")
```

This program determines whether or not the difference between two numbers is greater than five. To do so, you need to determine which number is larger before performing the subtraction to avoid negative numbers. So, we first check whether or not **`x`** is less than **`y`**. Then, we perform our subtraction based on which number is larger.

```
Pick a number between 0 and 10: 3
Pick a number between 0 and 10: 7
The difference is less than 5

Pick a number between 0 and 10: 10
Pick a number between 0 and 10: 0
The difference is greater than 5
```

Again, be careful with indentation. Code run by a statement must be indented one more than the statement is, regardless of where it is. So, the code inside an **`if`** statement needs to be indented once, and the code inside an **`if`** statement nested in another **`if`** statement must be indented twice.

`Try it yourself:` write a program that asks the user for a password, then a pin number. If the user gets either wrong, tell the user and quit the program.

Flags in Conditionals

It can be annoying to have to write out a long truth statement every time you want to check something. You wouldn't want the user to enter their password each and every time they try to access a website. You can simplify the process by using a **flag**. A flag is a variable that holds a boolean value, either true or false, based on something in the code. With a flag, we can simplify our program one further step:

```
x = input("How do you get input in Python? ")
correct = "input()" in x
if correct:
    print("Correct!")
else:
    print("Wrong!")
```

This will perform the same task as before, except now, we don't need to type **"input()" in x** every time we want to check if the answer is correct. Instead, we can simply check the value of **correct**.

```
How do you get input in Python? By using input()
Correct!
```

Try it yourself: change any of the programs you've written to use flags.

Key Points

- **input()** allows users to enter data from the keyboard into a variable.
- Truth statements (which become boolean values) are used to determine whether something is true or false. For example, **(10 == 10)** is **True**, and **(8 == 10)** is **False**.
- A flag is a variable that is assigned a boolean value. For example, in **x = True**, **x** is considered a flag.
- Conditionals are statements that use **if**, **elif**, and **else** that perform some behavior so long as a certain condition is **True** or **False**.

Exercises: Choose and complete one or more of the following exercises.

1. Create a program that makes a user go through some fun security checks (i.e. what is your favorite movie, etc.) before telling him your name.
2. Create a program that asks the user for their birthday and tells them their zodiac sign.
3. Create a small game of twenty questions (it doesn't actually need to be twenty questions).
4. Create a small text adventure, where the program presents the user with situations, asks the user what he or she wants to do, and changes the story accordingly. You may either provide them with a list of choices to choose from, or you may allow them to type their choices.

Responding to the Environment

Checking for Obstacles

So far, the Finch has been moving about wildly, relying on the user to provide a clear path for the unit to navigate. However, the Finch can check for obstacles on its own. The Finch uses two infrared obstacle sensors on the front of the unit and an infrared emitter on the center of the unit towards the front. The obstacle sensors are surrounded by black plastic and protrude from the 'face' of the Finch, while the emitter lies in an indentation above the 'nose' of the Finch. These sensors determine if an obstacle is present within 3 to 12 inches in front of the FInch.

To check these sensors, use **`detectObstacle()`**. Unlike other functions you've seen, **`detectObstacle() returns`** a value instead of doing something. **`detectObstacle()`** returns **`"left"`** if there is an obstacle that the Finch sees only on the left side, and returns **`"right"`** if there is an obstacle that the Finch sees only on the right side. The Finch returns **`"both"`** if the Finch detects obstacles on both sides and returns **`"none"`** if there are no obstacles that the Finch can see. You can use this in combination with a conditional to prevent the Finch from moving. Here's an example of using a conditional:

```
if (detectObstacle() == "none"):
    light("green")
else:
    light("red")
```

If there are no obstacles present, the above code will turn the nose green. If there is an obstacle present, the nose will turn red. Remember that the double equals sign in the above code denotes equality. So, the code above is using the double equals to detect if **`detectObstacle()`** is equal to, or the same as, none.

In addition to **`detectObstacle()`**, you can check each obstacle sensor individually, should you need to determine if the obstacle is on the left or the right. **`detectObstacleLeft()`** checks the left sensor and **`detectObstacleRight()`** checks the right sensor. These two functions will return **`True`** if it detects an obstacle or **`False`** if it does not detect an obstacle.

`Try it yourself`: check whether or not there is an obstacle. If there isn't, turn the nose green and move the Finch forward some distance. If there is, turn the nose red and make a short noise.

Checking Light Levels

The Finch is equipped with light sensors that can determine the brightness of the light at its location. The two light sensors are above the obstacle sensors. They are hard to see because they are hidden in small indentations on the top of the Finch.

To read the value from these sensors, use **`detectLight()`**. **`detectLight()`** returns a float (a decimal value) between 0 and 1. At 0, the Finch is in absolute darkness and at 1, the Finch is in blindly bright light. Here is an example of code that represents this:

```
if detectLight() < .2:
    light("white")
else:
    light("off")
```

The above code will turn the light white when the Finch is in relative darkness. If the Finch is in bright enough light, the light will turn off. Again, you can use **`detectLight()`** as though it is a number.

The finch also allows you to check the light levels on each side individually. To do this you can use **`detectLightLeft()`** and **`detectLightRight()`**.These functions return the same range of values as **`detectLight()`** except they only use the light level from one side of the Finch.

`Try it yourself`: check the level of light. If there is a large amount of light, make a high noise and turn the nose green. If there is a low amount of light, make a low noise and turn the nose red.

Checking the Temperature

The Finch has a temperature sensor that tellsyou how hot the environment is within 2 degrees Fahrenheit. The temperature sensor is in the upper middle portion of the Finch's face. It looks like a small black dot.

To check this sensor, use **`detectTemperature()`**. **`detectTemperature()`** returns the temperature in Fahrenheit.

```
>>> detectTemperature()
74.75
```

`Try it yourself`: check the temperature. If it is above room temperature, turn the nose red. If it is below room temperature, turn the nose blue. Average room temperature is about 70°F.

Checking The Orientation

The Finch has an accelerometer in the center of its shell. This accelerometer determines the forces applied to the Finch ranging from -1.5G to 1.5G. This allows you to test the orientation of the Finch. It also allows you to check whether or not the Finch is tapped or shaken.

The following functions allow you to check the Finch's orientation.

- **`isOnTail()`** - returns **`True`** if the Finch is standing on its tail. Returns **`False`** otherwise.
- **`isOnWheels()`** - returns **`True`** if the Finch is sitting on its wheels. Returns **`False`** otherwise.
- **`isFlippedOver()`** - returns **`True`** if the Finch is on its back. Returns **`False`** otherwise.

The following functions allow you to determine events that happen to the Finch. These will be more useful when we cover continuous behaviour. For now, it is okay to know that these exist and use to use the accelerometer in the meantime.

- **`isTapped()`** - returns **`True`** if the Finch was tapped since the last time you checked acceleration data. Returns **`False`** otherwise.
- **`isShaken()`** - returns **`True`** if the Finch was shaken since the last time you checked acceleration data. Returns **`False`** otherwise.

`Try it yourself`: check the Finch's orientation. If it is on its back, turn the nose red. If it is on its wheels, turn the nose green. If it is on its tail, turn the nose blue.

Key Points

- Use **`detectObstacle()`** to check for obstacles. Use **`detectObstacleLeft()`** and **`detectObstacleRight()`** to check each obstacle sensor individually.
- Use **`detectLight()`** to check for light.
- Use **`detectTemperature()`** to check the temperature in Fahrenheit.
- Use **`isOnTail()`**, **`isOnWheels()`**, and **`isFlippedOver()`** to check the Finch's orientation.
- Use **`isTapped()`** and **`isShaken()`** to check if the Finch was tapped or shaken.

Exercises

1. Turn the nose green and move the Finch forward a foot. If there is an obstacle at the end of the movement, turn the nose red, move the Finch backward 6 inches, and turn the Finch left 90 degrees. Repeat this process 3 times.

2. Write a program that changes the led color based on the light level. Make the light red at 0, yellow at .2, green at .4, blue at .6, violet at .8 and white at 1.

Loops and Flags

While Loops

A common task for programmers is repeating the same code over and over, also known as **iteration**. There are a couple different ways to do this in Python, but the simplest way is to use a **while loop**.

```
while True:
    print("So many loops!")
```

While loops will continue running as long as their condition stays **`True`**. For example, the loop above will never end, since its condition never changes to **`False`** This type of behavior is called an **infinite loop**, and should be avoided to prevent programs from getting stuck or running forever. If a loop doesn't have an exit condition or the exit condition can never be reached, the code inside the loop will run forever (or at least until you force-exit the program — in IDLE you can use [Ctrl] + [C] to exit whatever's running). To get the loop to exit, the condition needs to be a variable that changes throughout the flow of the loop.

```
x = 3
while x > 0:
    print("The value of x is", x)
    x = x - 1
print("The final value of x is", x)
```

The output of the loop above looks like

```
The value of x is 3
The value of x is 2
The value of x is 1
The final value of x is 0
```

Python checks the value of **`x`** each time the loop runs, and exits when the condition fails.

If the condition is **`False`** the first time through, Python will skip over the loop entirely. If **`x`** was set to 0 instead of 3 in the last example, the output would look like

```
The final value of x is 0
```

This type of strategy is called **decrementing**, where a variable is decreased every time the loop runs. Another common technique is **incrementing**, where the variable is increased every time.

```
x = 0
while x < 3:
    print("The value of x is", x)
    x = x + 1
print("The final value of x is", x)
```

```
The value of x is 0
The value of x is 1
The value of x is 2
The final value of x is 3
```

While loops are also useful for user input, such as waiting for a user to enter a command during a program. The follow snippet of code asks the user to enter the word “red”, then keeps running until they enter that word.

```
userInput = ""
while userInput != "red":
    userInput = input("Enter the word red! ")

print("The user entered red!")
```

This then outputs:

```
Enter the word red! blue
Enter the word red! orange
Enter the word red! red
The user entered red!
```

`Try it yourself:` print out your name 5 times, then change it to print 100 times.

Using Flags With Loops

One problem with while loops is that they only have one conditional. Conditionals can be combined with operators, but that can create messy code that’s hard to read and write. Especially as code gets more complicated, it can become very handy to use **flags** to check conditions throughout the loop.

If a programmer is writing a loop to run diagnostics on a car, the loop might need to run until it finds a broken component. Writing the loop so that it checks every single component inside its conditional would be cumbersome and impossible to read through.

To avoid this, the programmer can add a flag to signal that a broken component was found.

```
maintenance = False
while maintenance != True:
    if tirePressure < 25:
        print("Tire pressure too low!")
        maintenance = True
    if idleRpm < 600:
        print("Idle RPM too low!")
        maintenance = True

    # Run checks on the rest of the car

print("This car needs maintenance!")
```

If the tire pressure was 35 but the idle RPM was only 500, the maintenance check would fail, and the program would output:

```
Idle RPM too low!
This car needs maintenance!
```

`Try it yourself:` write a program that continuously takes user input until the input either has a “r”, “b”, or “g” in it. It may be easier to do this using a flag.

For Loops

An easier way to decrement or increment in a loop is to use the second type of loops, the **for loop**. To write a while loop that executes ten times, we would write

```
counter = 0
while(counter < 10):
    print("The Loop is running")
    counter = counter + 1
print("The loop has finished")
```

Using a for loop instead, we can do the same thing using the following code

```
for counter in range(10):
    print("The loop is running")
print("The loop has finished")
```

Both of these programs will do exactly the same thing; they are semantically equivalent. The only difference is the way the code runs in the background. The variable **`counter`** is accessible from inside the loop on either one, so we could have them print out the counter

with a statement such as **`print(counter)`**, and both loops would still have the same output.

For loops use a different kind of approach to incrementing. The loop variable is set to an element in a list of numbers, which we create using **`range()`**, which provides a list of numbers from the start point to the end point minus 1. You can set the endpoint or the start and end points using **`range(max)`** or **`range(min, max)`**, where *min* and *max* can be integers or a variable. In the code above, the variable **`counter`** is set to the numbers 0 to 9, increasing by one on each pass of the loop. It's fine to know that for loops have a counter variable, a start point (0 by default), and an end point. This list reading allows some more advanced functionality that you may want to explore on your own later.

A common use of **for** loops is repeating a chunk of code with different values each time.

```
for i in range(5, 11):
    print(i, "to the second power is", i**2)
```

The loop above will run through six times, using the values 5, 6, 7, 8, 9, and 10 for **`i`** on each pass. The output looks like

```
5 to the second power is 25
6 to the second power is 36
7 to the second power is 49
8 to the second power is 64
9 to the second power is 81
10 to the second power is 100
```

Again, notice that 10 is one less than the maximum value, 11, in the range.

While loops are best for code that needs to wait for something to happen. **For** loops are best for code that needs to repeat a certain number of times.

`Try it yourself:` write a program that counts from 0 to 100.

Nested Loops

Sometimes a task needs to include a loop, but that task needs to be in a loop too. Just as **if** statements can be nested inside each other, loops can contain other loops. **For** loops can be put inside of **while** loops, and **while** loops can be put inside of **for** loops, but most of the time it's more common a **for** loop inside another **for** loop.

```
for i in range(2):
    print("<", end='')
    for n in range(5):
        print("-", end='')
    print(">")
```

Python runs through the outer loop, prints out an arrow, and starts the inner loop. The inner loop runs five times and prints a line of dashes. Once it ends, it dumps back to the outer loop, and moves on to the second pass. The output looks like

```
<----->
<----->
```

(Hint: you can stop Python from starting a new line after printing by adding an extra tag, ***end=”****)*

`Try it yourself`: write a program that counts from 1 to 10, 3 times.

Key points

- While loops carry out a specified action so long as a condition remains **`True`**. Typically, if you aren’t sure how many times an action will be iterated, you will want to use a while loop.
- For loops carry out a specified action a predetermined number of times.
- Nested loops are loops within loops.
- Incrementing a variable consists of adding 1 to its value every time some event occurs
- Decrementing a variable consists of subtracting 1 from its value every time some event occurs

Exercises: Choose and complete one or more of the following exercises.

1. Write a program that prints out the lyrics to *99 Bottles of Beer on the Wall*.
   (Hint: you can count down by subtracting the counter from 99)
2. Write a program that asks for your name until you get it right.

Continuous Behaviour

Continuous Behaviour

Using loops, you can make the Finch do things continuously. Now, you can make it drive in a square as many times as you want by simply putting it in a loop. But, this is most powerful when you make the Finch run until it reaches some exit condition. Here is an example.

```
while not isTapped():
    forward(10)
    turnLeft(90)
```

This simple program will cause the Finch to travel in a square until you tap it. In this program, the exit condition is being tapped. Until it is tapped, it will just keep driving forward 10 inches and turning left 90 degrees.

This method of coding allows you to complex behaviours that occur constantly until the user decides to quit. Many modern programs operate this way: operating systems require the user to shut down, browsers say open until they are closed, and your router keeps the internet on until you shut it off.

`Try it yourself`: move the Finch in a triangle. At each turn, print out the temperature and light level. If the Finch is ever on its tail, exit the program.

Combining Conditionals and Iteration

Now that you can make the Finch behave continuously and conditionally, you can create very complex behaviours by combining them. Putting conditionals in loops allows you to create a branching behavioural scheme that operates until some condition is met. This strategy will be incredibly important moving forward. Many programs will use this strategy to leverage the full power of the Finch. This example demonstrates how you can use this to continue checking the light until you've moved into a spot where the light is greatest.

```
checks = 0  #How many times we've checked
lightLevel = detectLight()  #Our initial light level
while checks < 4:   #Until we've checked 4 times
    forward(12)
    if detectLight() > lightLevel:
        #If light is greater here, reset
        lightLevel = detectLight()
        checks = 0
    else:
        #If light is not greater here, go back
        backward(12)
        #Preparing for next check
        turnLeft(90)
        checks = checks + 1
```

The code uses two variables: **`checks`** and **`lightLevel`**. **`checks`** is the number of times you've moved away from our best point. The best point is our initial point at first, so we initialize **`lightLevel`** to our current light. We then move from the center, checking the light. If the light is greater, we make that our new center, **`lightLevel`** to the light there and resetting setting **`checks`** to 0. If it is not, we move back to our starting point, turn 90 degrees for our next check, and mark that we've checked one more time. We stop the loop when we've done so 4 times with no improvement.

The result is that the Finch will look for the place with the greatest amount of light in the room. The Finch will do so until it's convinced, to an accuracy of about of foot, that it has found the brightest spot in the room. The conditional creates the checking behaviour, which is how the Finch comes to know the brightest spot. The loop allows the Finch to perform this check continuously, so that it may check more than just a single spot. By combining loops and conditionals then, we've created this complex behaviour.

`Try it yourself`: create a program that finds the place with the greatest temperature in the room.

Simultaneous Behaviour: Revisited

Recall that you can move the Finch, use the buzzer, and light the nose all at the same time. The movement functions, **`forward()`**, **`backward()`**, **`turnLeft()`**, and **`turnRight()`**, do not stop the code from executing when they are not supplied with a distance or angle. If they are, you can continue the code by adding an extra boolean value input to the function like so:

```
forward(10, False)
backward(15, False)
turnLeft(90, False)
turnRight(180, False)
```

Note that calling movement functions that do not wait, one after another, will override each other, defaulting to the most recent function call. In this case, the Finch will only turn right 180 degrees. To prevent overriding, you can check **`isMoving`**, a flag that is **`True`** when the Finch is moving and **`False`** when the Finch is not moving. Particularly, you'll want to use a loop with a conditional like so:

```
emergencyBrake = False
while isMoving and not emergencyBrake:
    if detectObstacle() != "none":
        emergencyBrake = True
```

This technique will also be very valuable. While the Finch is moving and doing other things, you can check the sensors and make the Finch respond to its environment. Just set the movement and execute a loop that checks the sensors until **`isMoving`** is **`False`**.

`Try it yourself`: turn the nose green. Then, move in a square. Stop if the Finch detects an obstacle and turn the nose red.

Key Points

- Use loops to build a continuous behavior.
- Combining loops and conditionals allows you to build complex behaviours.
- By using movement functions that continue executing code, you can make the Finch operate while making it respond to its environment.

Exercises

1. Make the finch move along a path with n sides each of length d where the user can enter both n and d.

   Hint: for a square with 4 sides, each turn needs to be 90° = 360° / 4, and a triangle with three sides, the finch will need to turn 120° = 360° / 3.

2. Make the finch patrol back and forth in a straight line between two points, stopping, changing the color of the nose light and emit a noise if it detects something blocking it.
3. Have the finch follow a path, but only in the dark. The finch should wait for darkness before it tries to move.

Functions

Functions

In programming there are times when you may want to repeat blocks of code in different parts of your program in a way that loops or if-else statements can't help. In these cases, it is often useful to make **functions.** In fact, you have been using functions like **`print()`** and **`range()`** all week.

Think of a function like a recipe. A recipe gives you a list of instructions to follow in order to make a meal (think back to the PB&J exercise). Afterwards, when you want to make the same dish, you just need to recall the recipe and do so. Like a recipe, a function contains a list of instructions for a program to follow, and allows the program to access them over and over again when the function is called.

```
def spam():
    print("eggs")
    print("spam")
    print("eggs and spam")
```

Functions begin with the **`def`** keyword (since you need to **def**ine what the function does), followed by the function name, parentheses, and a colon. The block code indented underneath it will execute normally when the function is called.

```
>>> spam()
eggs
spam
eggs and spam
```

Like with loops and if/else statements, block indentation is very important. The code in the function after the **`def`**, name, parenthesis, and colon must be indented, and any code not indented will be presumed to be the end of the function.

`Try it yourself:` Write a function that prints your name without a new line at the end. Use your function to print out "Hello, my name is <your name>, and I wrote this." Change your name function to print out a different version of your name (add middle initial, use nickname, etc.), and run the same "Hello…" code as before. (Hint: remember that the print(*output*, end=*endCharacter)* lets you change how the print function ends.)

Parameters and Arguments

Functions can also have their abilities expanded greatly with **parameters**. Parameters are variables that the function accepts as input to be used within the function, introduced by putting variable names within the parenthesis after the function name, like so:

```
def spam(x):
    z = x+9
    print(z)
```

When the above function is called, a number has to be placed inside the parentheses following it, unlike the previous function. This is because **`spam(x)`** has the parameter, **`x`**.

```
>>> spam(5)
14
```

The inputs of a function are formally called its **arguments**. In the code above, 5 is the argument of **`spam(5)`**. These seem like similar concepts, but they are distinct: a parameter is placed within the definition of the function as required input, while an argument is the actual data input into a function's call. When talking about a function call, the inputs are arguments. When speaking of the function itself, the inputs are parameters. In other words, a function *has* parameters and a function *takes* arguments.

If you think you've seen something similar to this before, you have **`print()`** is a function that has a string parameter, and the string you pass to be printed is its argument.

`Try it yourself:` Write a function that has a string as a parameter and tells you if it is your name.

The Return Statement

Functions can also be ended with a return statement. The return statement sends data out to be used as the function caller sees fit, such as a string to be printed, or a number to be assigned to a variable:

```
def spam():
    x = 7
    return x
y = spam()
```

`Try it yourself:` write a function that returns your name.

Variable Scoping

Another important part of functions is the idea of **variable scope.** The scope of a variable is the area of the code in which it can be used. For example, in the above code for **`spam(x)`**, the scope of variable **`z`** is only within the function. If you were to attempt to use **`z`** in any other part of the code, you would get an error message, since for all intents and purposes, **`z`** only exists inside of **`spam(x)`**. Similarly, no variables created outside the function can be accessed directly by the function without being inserted as a parameter.

Key Points:

- Functions serve as reusable blocks of code that can be placed anywhere within a program.
- The heading of a function is: **`def functionName():`**
- Each line of code after the "def" line of the function is indented.
- To run a function within your program, you must call the function: **`functionName()`**
- Functions can take parameters. Parameters are values within the parentheses of functions that are taken into account when running a function: **`functionName(parameterGoesHere)`**.
- When you call a function with a parameter within your program, an argument replaces the parameter. In functionName(10), 10 is the argument.

`Try it yourself:` write two functions that both take numbers as input, but return different values (x * y and x / y, for example). Use the same variable names for both functions.

Exercises: Choose and complete one or more of the following exercises.

1. Write a function that uses the Pythagorean Theorem ($a^2 + b^2 = c^2$) to find the length of the hypotenuse when given the lengths of the other two sides of the triangle. *(Hint: the square root of a number can also be found by raising the number to the 1/2 power)*
2. Write a function that when given a name and an age will return a string that says ("[Name] is [age] years old")
3. Write a function that computes the total cost of an order of food. There are three food items: pizza which costs $9, sandwiches which cost $6, and macaroni and cheese which costs $4. For each food item, the user enters how many orders of that food item he or she would like. The function should display the total cost of the food order.

Using Functions: Navigation

Using Functions

By using functions, you can eliminate a lot of repetitive code. By eliminating repetitive code, you can create complex behaviours with only a few lines of code and reuse that code wherever applicable. One of your earlier exercises probably looked something like this:

```
forward(12, False)
emergencyBrake = False
while isMoving and not emergencyBrake:
    if detectObstacle() != "none":
        emergencyBrake = True
return emergencyBrake
turnLeft(90)
forward(12, False)
emergencyBrake = False
while isMoving and not emergencyBrake:
    if detectObstacle() != "none":
        emergencyBrake = True
return emergencyBrake
turnLeft(90)forward(12, False)
emergencyBrake = False
while isMoving and not emergencyBrake:
    if detectObstacle() != "none":
        emergencyBrake = True
return emergencyBrake
turnLeft(90)forward(12, False)
emergencyBrake = False
while isMoving and not emergencyBrake:
    if detectObstacle() != "none":
        emergencyBrake = True
return emergencyBrake
turnLeft(90)
```

This can be painful to look at as a whole. It can also be annoying to repeatedly type that while loop when it does the exact same thing every time. Instead, you can make a function that does it for you, giving the code a nice, crisp look.Now, we can just check if the emergency brake was triggered.

```
def checkObstacleWhileMoving():
    emergencyBrake = False
    while isMoving and not emergencyBrake:
        if detectObstacle() != "none":
            emergencyBrake = True

forward(12, False)
if checkObstacleWhileMoving():
    stop()
turnLeft(90)
forward(12, False)
if checkObstacleWhileMoving():
    stop()
turnLeft(90)
forward(12, False)
if checkObstacleWhileMoving():
    stop()
turnLeft(90)
forward(12, False)
if checkObstacleWhileMoving():
    stop()
```

Even after these changes, we have a lot of repeated code. If you look closely, you'll notice that the above code the same four chunks repeated. You can make yet another function that will simplify it even further.

```
def moveForwardAndTurn():
    forward(12, False)
    if checkObstacleWhileMoving():
        stop()
    turnLeft(90)
```

Our code for that behavior is now four lines of code.

```
moveForwardAndTurn()
moveForwardAndTurn()
moveForwardAndTurn()
moveForwardAndTurn()
```

To even further simplify, we can put this in a for loop that automates the repetition for us!

```
for i in range(0, 4):
    moveForwardAndTurn()
```

This is much better. The code performs the intended behavior, is easy to look at and understand, and is far easier to type out. Functions exist purely for the programmer's ease. They are tools that simplify otherwise complicated and messy code. Use functions whenever you want to make your code easier on the eyes and on your fingers.

`Try it yourself`: take any exercise or try it yourself you've completed thus far that uses repeated code or was hard to read at the end. Translate that code to use functions instead.

Responding to the Environment with Functions

The above code was a way to check for obstacles. This was best implemented using functions because of the complexity involved in avoiding obstacles. The function implemented above, **`moveForwardAndTurn()`**, handles obstacles by truncating the forward movement and turning in that spot instead of at the end. However, you may want to handle this differently and not have to retype the code for handling obstacles each and every time. The same is true of anything else in your environment, like temperature, light, and position. In fact, you may want to have a whole function dedicated to just responding to the environment.

```
def checkEnvironment():
    if detectLight() > .5:
        #do something
    else:
        #do somehing
    if detectTemperature() > 70:
        #do something
    else:
        #do something
    #etc
```

This allows you to compact your code that responds to the environment in a concise manner so you only have to write a single line when you want to do so.

Function Application: Navigation

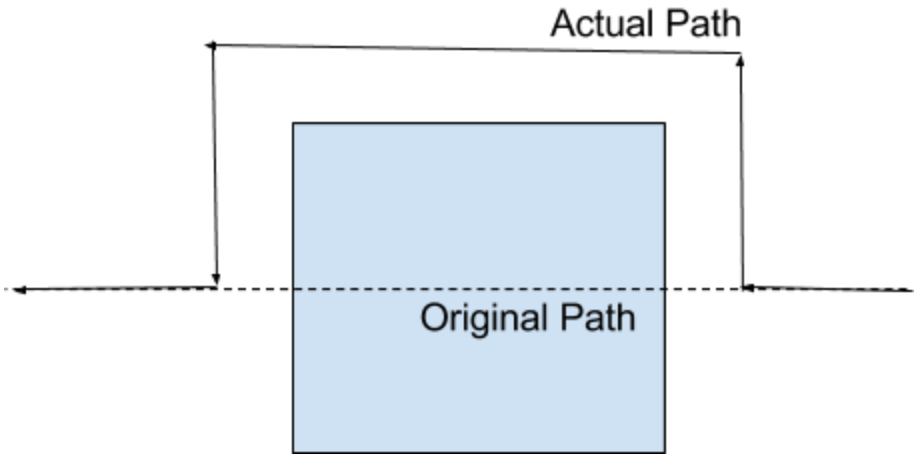
A navigation system is an example of a real-world application that uses functions. The Finch has several useful functions that control basic movement, but that alone is hardly enough to get the Finch to move from point A to point B while avoiding obstacles in between. Suppose I want to move forward in a line. This seems like it would only involve one step:.

```
forward()
```

However , that only works if you don't have an obstacle in the way. If you do, you will have to find a way to move around it. To deal with this in an orderly fashion, here is a function that handles an obstacle in an orderly fashion.

```
def goAroundObstacle():
    obstacle = True
    distanceLeft = 0        #Tracking distance
    #Going left
    while obstacle:
        if not detectObstacle():
            obstacle = False
        else:
            turnLeft(90)
            forward(6)
            distanceLeft = distanceLeft + 6
            turnRight(90)
    #At this point, obstacle is no longer in front of us
    obstacle = True
    #Going forward
    while obstacle:
        forward(6)
        distanceForward = distanceForward + 6
        turnLeft(90)
        if not detectObstacle():
            obstacle = False
        else:
            turnRight(90)
    #Returning to line of travel
    forward(distanceLeft)
    turnRight(90)
```

Calling this function will cause the Finch to go around a square obstacle. The function first assumes that there is an obstacle present and enters the first while loop. So long as the obstacle is still in front of the Finch, the Finch will move left 6 inches and check for an obstacle. Once it has done that, the Finch will move along the length of the obstacle, checking if the obstacle is still present to its left. If it's not, the Finch will return to the original path and return how far it has moved forward to get around the obstacle.



Here is an example use of this:

```
while not isTapped():
    forward()
    if detectObstacle():
        stop()
        goAroundObstacle()
```

This code will continue moving the Finch in a line until it is tapped. If the Finch encounters an obstacle, it will execute our **`goAroundObstacle()`** function to navigate around the obstacle. Note that **`goAroundObstacle()`** assumes that the obstacle is a square. For irregular shaped obstacles or an unknown environment, more complicated code would be needed. For now, recognize that this function allows one to navigate around a square obstacle easily.

`Try it yourself`: move the Finch in a circle. If it encounters an obstacle, turn in the opposite direction until there is no longer an obstacle. Then, go in a circle in the opposite direction.

Key Points

- Functions allow you to make code that is cleaner and easier to use.
- Functions allow you to have systematic ways of responding to your environment.
- Navigation is a complex system that is best implemented using functions.

Exercises: Choose and complete one or more of the following exercises.

1. Write a function that has the Finch write out a letter of the alphabet. Then, write a function to have the Finch write out the initials of each team member.
2. Move the Finch forward until it encounters a wall. The, move the Finch along the wall until you tap it.

Student Projects

Stay in the Light

Setup: Turn out the lights in the room. Using flashlights, computer screens, and other lights sources, create patches of light throughout the room.
Task: Create a program that does the following:

- When the Finch is in the light, turns the LED nose green,
- When the Finch is in the darkness, turns the LED nose red,
- If the Finch is in darkness, finds another source of light.

Stay out of the Light

Setup: Turn the lights out in the room. Give students flashlights.
Task: Create a program that does the following:

- When the Finch is in the darkness, turn the LED nose green,
- When the Finch is in the light, turn the LED nose red,
- If the Finch is in the light, the Finch runs away from the light.

Race to the Finish

Setup: Create a small race track using books and other obstacles.
Task: Create a program that does the following:

- Plays three sounds in ascending frequency,
- At the end of the last sound, takes off on the race course,
- When the Finch detects an obstacle, backtracks and tries another route,
- When the Finch is tapped, plays a victory jingle and stops.

Escape the Maze

Setup: Create a maze using books and other obstacles.
Task: Create a program that makes the Finch navigate the maze.