Programming in Python

by Michael Parkinson and Timothy Johnson

Python is a high-level, object-oriented programming language that focuses on simplicity and readability. It is one of the top 5 most popular programming languages, and it has many applications including science, robotics, the web, security, and artificial intelligence. Today, you are going to learn the basics of how to code in Python 3, and you will eventually use it to program your semester engineering project.

Setting Things Up

1. Go to https://repl.it/languages/python3
2. Create an account if desired, otherwise click "continue as Anonymous" (it is optional)
3. Click Settings (the gear on the top left), and choose "4" for "indent size."
4. You're all set!

Strings: Output and Input

It is an old tradition for new programmers to display the phrase **`Hello, world!`** to the console. Type this line exactly as it is displayed into the code editor on the left, and click "run."

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

If you have never programmed before, now you have! You just wrote your first program, which prints something to the ***console***. It's pretty simple, but everything starts this way.

You might be wondering how your program works. **`print`** is the name of a ***function*** that takes an input and prints it to the console. All functions end with a pair parentheses **`()`**, inside which any ***arguments*** are inserted. Arguments are just information given to the function so that it knows exactly what to do. In this example, **`"Hello, world!"`** is the argument. The quotation marks **`""`** indicate that the information is a ***string***, which is a set of letters, numbers, and punctuation. For example, a string could be a name, a street address, or a paragraph from a book. There are other ***data types*** as well, which will be introduced later.

Rather than putting the string directly into a function as an argument, we can ***assign*** it to a ***variable*** and use the variable as the argument, instead. Variables store information for later use, and their values can vary. The first line of this program assigns your name to the variable **`name`**; notice how the name of the variable and the string are separated by an equals sign **`=`** with spaces on each side of it. The second line prints the variable `name`; notice that it does not have quotation marks anymore, since it has already been assigned as a string. Programs always ***execute*** from top to bottom, line-by-line.

```
name = "Your_name_here"
print(name)
```

Strings can also be inserted into other strings. Put a pair of curly brackets **`{}`** in the first string at the position that you want the second string to be inserted. Right after the second quotation mark, put the sub-function **`.format()`** with the argument **`name`**. A sub-function, denoted with a period **`.`** after a variable, is like normal function except that it uses the variable that comes before it. This will replace the curly brackets with your name.

This is a good time to introduce ***comments***, which begin with a pound sign **`#`** and can be placed at the end of a line of code or on their own line. Comments are not executed; they just describe how a program works to anyone who is reading it, and are important for large or complex programs.

```
name = "Your_name_here"
print("Nice to meet you, {}.".format(name))  # Print a personalized greeting
# Here is a full-line comment.
```

The **`print`** function is used for string ***output***, and similarly, the **`input()`** function is used for string ***input***. **`input`** does not need an argument, but if there is one, it will display it to the console just like **`print()`** does. Additionally, it will pause the program and wait until the user types something into the console and presses Enter. After Enter is pressed, the **`input()`** function will ***return*** what was entered as a string. Run this program, click on the console (the dark window on the right side), type your name, and press Enter. Almost every program that you use has some type of input, whether it is a toggle button, a text box, or a keypress.

```
name = input("What is your name? ")  # The extra space makes input in some consoles look nicer
print("Nice to meet you, {}.".format(name))
```

Before the next step, a few things need to be set up in repl.it. Click the "Add new file" button under the "settings" button in repl.it. Click "Continue" in the popup. Notice how it added a main.py file tab above the code editor. Click the "Add new file" button again, click the three dots on the new tab, and rename the file to "spaceship.py". Type the line **`import spaceship`** into the main.py tab (and remove any other code). This will tell the program to run the code that is in the spaceship.py tab.

If you haven't guessed yet, you are going to code a space ship program. It will keep growing throughout this tutorial, step-by-step, and at the end you will have a program which implements everything you will have learned. Type the following two lines into the new spaceship.py tab, and click "run". The program will ask you what your ship's name is. Every ship needs a captain, so it also asks for your name.

[Switch to the spaceship.py tab]

```
ship_name = input("What is your space ship's name? ")
captain = input("Welcome aboard {}, Captain. What is your name? ".format(ship_name))
```

Math

So far you have used one data type, the string. Two more data types are ***integers*** and ***floats***. Integers are whole numbers, and floats are numbers with decimal places. These are different from strings, because they don't have quotation marks and they can be used in mathematical operations.

Basic Mathematical Operators

Add	Subtract	Multiply	Divide	Power	Root	Absolute
`a + b`	`a - b`	`a * b`	`a / b`	`a ** b`	`a ** 0.5`	`abs(a)`

Parentheses can be used for controlling the order of operations on a calculation. Additionally, you can convert a string **`str`** of numerals into an integer with the function **`int(str)`**. This is called ***casting***, and is very useful if you need to convert a string returned by an **`input()`** into an integer or some other data type. Here is an example that uses the basic operators, parentheses, and casting. Make predictions of what the outputs will be and run it in main.py, removing **`import spaceship`**.

[Switch to the main.py tab]

```
a = 5 - (1 + 2)
b = 3.14 / a * 4
c = (5 ** 2 - 1) ** 0.5
d = abs(-42)
e = 7 + int("7")
print(a, b, c, d, e)  # Multiple arguments in print() will be separated by spaces
```

Useful Formatting Codes

String	Integer	Float, 6 decimals	Float, X decimals	Nth arg of format()
`{}`	`{} or {:d}`	`{:f}`	`{:.Xf}`	`{N}`

```
...
print("a: {}, b: {:f}, c: {:.3f}, d: {:d}, e: {:d}".format(a, b, c, d, e))
```

Lists

Another data type is the ***list***. A list is an ordered group of items, such as strings or numbers. Just like strings are denoted with quotation marks, lists are denoted with square brackets **`[]`**. Inside the brackets, items are separated by commas. In this case, the values in the list are strings. **`fruits[0]`** takes the first item in the list **`fruits`**, which is **`"apple"`**. You might be confused why the ***index*** is the integer **`0`** rather than **`1`**, since it is the first item. The reason is that computers start counting at the number 0, so if you want the Nth item in a list, the index should be equal to N - 1. To add a new item to a list, use the sub-function **`.append(item)`**. Lists can be edited by using the index of the item you want to change and assigning a new value to it (with an equals sign), just like a variable. A sub-list, or ***slice***, of **`list`** uses the format **`list[A:Z]`** which creates a new list consisting of **`list[A]`** and all items up to but not including **`list[Z]`**. If A is empty the slice will begin at **`list[0]`**, and if Z is empty the slice will end with the last item in **`list`**.

```
fruits = ["apple", "orange", "banana"]
print("The {} is my favorite fruit.".format(fruits[0]))
fruits.append("pineapple")  # Add "pineapple" to the end of the list
fruits[1] = "lemon"  # Replace "orange" with "lemon"
print(fruits)  # Printing a list will display it in a special format
print(fruits[1:3])  # Print list slice starting at index 1 and ending with index 3-1=2
print(fruits[:2])  # Print list slice starting at beginning of list and ending with index 2-1=1
print(fruits[2:])  # Print list slice starting at index 2 and ending with last item (index 3)
```

The **`sum()`** function can be used on a list of integers and/or floats to add them together, and the result can be stored in a variable as an integer or float. Similarly, the **`len()`** function returns the number of items that are in the list. An average of multiple numbers can be easily calculated by using these functions together.

```
numbers = [8, 3.14, 1969]
average = sum(numbers) / len(numbers)
print("The average is {:.2f}".format(average))
```

Time for another addition to the spaceship.py program! This code creates a list of strings which are planet names and assigns your location to the index of Earth. This is called ***initializing***, or giving variables initial values. The program then prints both variables, gets a destination input and converts it to an integer, calculates the estimated time of arrival, and indicates that travel has begun. Notice how the last line is a continuation of the previous line, which would otherwise be very long. Python allows this as long as the second line is indented and no values, functions, or variable names are split. Remember to add the **`import spaceship`** line to main.py before you run it.

[Switch to the spaceship.py tab]

```
ship_name = input("What is your space ship's name? ")
captain = input("Welcome aboard {}, Captain. What is your name? ".format(ship_name))
planets = ["Sun", "Mercury", "Venus", "Earth", "Mars", "Jupiter",
           "Saturn", "Uranus", "Neptune", "Pluto"]
location = 3
print("--------Captain's Log for {}--------".format(ship_name))
print("LOCATION: {}".format(planets[location]))
print("PLANETS: {}".format(planets))
destination = int(input("Where would you like to go, {}? ".format(captain)))
eta = abs(destination - location) ** 2 / 2
print("Traveling from {} to {}. It will take {:.1f} years.".format(
      planets[location], planets[destination], eta))
```

Boolean and If Statements

All of the data types up to this point have had almost infinite possible values, but ***Booleans*** just have two values: **`True`** or **`False`** (notice the capitalization). Booleans are fundamental to any program, even if it isn't immediately apparent. Computers store and process data with Boolean values and ***Boolean algebra*** at the fundamental level, in the form of 1s for **`True`** and 0s for **`False`**. Whenever you run your program, it is converted into ***binary***, which is a representation of your program using only 0s and 1s. Boolean algebra can be

used in a program to make decisions; if Boolean algebra didn't exist, every program would run exactly the same way every time, which would be boring.

The three basic Boolean operators are **`and`**, **`or`**, and **`not`**. **`and`** will be **`True`** if both inputs are **`True`**, **`or`** will be **`True`** if at least one input is **`True`**, and **`not`** returns the opposite of the input. These properties can be shown with what are called ***truth tables***, which show the output for each set of inputs.

Boolean Algebra Truth Tables (1 = True, 0 = False)

A	B	A and B
0	0	0
0	1	0
1	0	0
1	1	1

A	B	A or B
0	0	0
0	1	1
1	0	1
1	1	1

A	not A
0	1
1	0

There are also other Boolean operators which compare other data types, such as numbers and strings. They always return a Boolean regardless of the values being compared, meaning they can be used for decision making. Parentheses can be used to control order of operations, like in math.

Boolean Operators

Equal To?	Not Equal To?	Greater Than?	Less Than?	Greater Than or Equal To?	Less Than or Equal To?
`a == b`	`a != b`	`a > b`	`a < b`	`a >= b`	`a <= b`

[Switch to the main.py tab]

```
print(True and False, True or False, True and not False, False or (True and False))
print(2 == 1, 2 != 1, 100 > 99, 3.14 >= 3.14)
```

Now that you know how to generate Boolean values from data, it is time to make some decisions based on them. This can be done with an ***if statement***. The most basic if statement begins with the keyword **`if`**, followed by a Boolean calculation and a colon **`:`**. The next line begins with an indent (this is important!), followed by any lines that you want to be executed if the Boolean calculation returns **`True`**. Optionally, one or more secondary **`elif`** (else if) statements can be added to the primary one, and they will only run if all of the previous statements return **`False`**. Also optionally, the keyword **`else`**, with no Boolean value or calculation, can be used. If all of the previous **`if`** and **`elif`** statements return **`False`**, the lines in the **`else`** statement will be executed.

```
apple_cost = 5
money = 4.99
if money > apple_cost:
    print("Congratulations, you have purchased an apple. You now have ${:.2f}.".format(
        money - apple_cost))
elif money == apple_cost:
    print("Congratulations, you are now broke.")
else:
    print("You don't have enough money to buy an apple.")
```

In the previous spaceship.py program, you could select Earth as the destination, and the program would say "Traveling from Earth to Earth." That doesn't make any sense. An if statement will fix the problem by checking if the destination and the location are the same. Notice how some of the old lines of code are kept, but are indented because they have been moved to inside the if statement.

[Switch to the spaceship.py tab]

```
...
destination = int(input("Where would you like to go, {}? ".format(captain)))
if destination == location:
    print("We are already on {}, so we can't travel there.".format(planets[location]))
else:
    eta = abs(destination - location) ** 2 / 2
    print("Traveling from {} to {}. It will take {:.1f} years.".format(
          planets[location], planets[destination], eta))
    location = destination
    print("You have arrived at {}!".format(planets[location]))
```

Loops and Timing

Programming is so powerful because it can do many calculations with very little human time required. Loops are a good example of this. Loops can do a certain operation a few times, or millions of times. A ***for loop*** executes a set of code for a certain amount of times, or ***iterations***. The format starts with the keyword **`for`**, followed the iteration variable name, the keyword **`in`**, the object through which you are iterating, and a colon. The number of iterations depends on the length of the object through which the for loop is looping. The function **`range(x)`** generates a list of length x, from 0 to x - 1. If you want to ***break*** out of a loop when a certain condition is met, you can use the **`break`** keyword to jump outside of the loop. Notice that in the example below, the second for loop stops after the fifth iteration because of the **`break`**.

[Switch to the main.py tab]

```
countries = ["USA", "Japan", "Switzerland", "Russia"]
for country in countries:  # Loop through the countries, "country" is the current country
    print(country)  # Print the current country
for i in range(10):  # Loop through a list of 0-9, "i" is the current number and is unused
    if i == 5:
        break  # Jump out of loop
    print("Python")  # This will run 5 times (not 10)
```

A ***while loop*** is very similar to an if statement, except that it will continue to run the embedded code while the result of the Boolean calculation is **`True`**. If it never returns **`False`**, the loop will continue infinitely, so it is important to ensure that it will eventually ***terminate*** under some condition.

```
while input("Say 'cheese'") != "cheese":  # Keep looping if you don't say "cheese"
    print("I need you to smile.")
print("Click!")  # Runs after you say "cheese"
```

Another useful function is **`time.sleep(x)`**, which will ***halt*** the program for **`x`** seconds. Computers compute so fast that sometimes they need to be slowed down for their outputs to be seen. There is a problem, though; **`sleep()`** requires the ***module*** **`time`**, which is not loaded by default in Python. The module **`time`** needs to be ***imported*** in order to use the function **`sleep()`**. Does this look familiar? This is the same way that the spaceship.py program is in repl.it; **`spaceship`** is a custom module which is imported into main.py, and it is executed by doing so.

```
import time  # Modules are imported at the top of the program, before any other code
for i in range(5):
    print(i)
    time.sleep(1)
```

The **`time`** module also has a **`time`** function which can be used to measure elapsed time. The **`time.time()`** function returns a float representing the number of seconds since January 1, 1970 UTC (called the ***epoch***), which is a very large number. To measure time elapsed between two events, the result of **`time.time()`** can be saved to a variable T1 at the time of the first event, and the result can again be saved to variable T2 at the time of the second event. The difference T2 – T1 is the elapsed time in seconds between the two events.

```
import time
t1 = time.time()
print(t1, "sec")  # Print seconds since Jan 1, 1970 with labeled units
print(t1 / 3600 / 24 / 365, "yrs")  # Print years since Jan 1, 1970 with labeled units
while input() == "":  # Continue timing until non-empty console input
    t2 = time.time()
    print(round(t2 - t1, 3))  # Print time since last console input
    t1 = t2  # Reset timer to start at t2 which is now
```

Speaking of your spaceship, it is advanced and can travel to multiple planets in one journey. Because of this, add a while loop. Have it terminate when your location is the Sun, because anyone who flies to the Sun isn't coming back. Also remove the current, ugly display of planets, and replace it with a for loop. Lastly, traveling from planet to planet is not instant. Add a delay with an argument of **`eta`** (don't worry, it will only wait a few seconds, not years).

[Switch to the spaceship.py tab]

```
import time
ship_name = input("What is your space ship's name? ")
...
location = 3
while location != 0:
    print("--------Captain's Log for {}--------".format(ship_name))
    print("LOCATION: {}".format(planets[location]))
    print("PLANETS:")
    for i in range(len(planets)):
        print("{}: {}".format(i, planets[i]))
    ...
        print("Traveling from {} to {}. It will take {:.1f} years.".format(
              planets[location], planets[destination], eta))
        time.sleep(eta)
        ...
```

Functions

You have learned how to use lots of built-in functions. They are useful, right? Now you will learn how to make your own functions, which are very useful when you want to reuse a certain part of code multiple times. First of all, functions must be ***defined*** before they are ***called***. Because of this, function definitions are usually placed near the top of the program. Their format begins with **`def`** (define), followed by the function name, the required arguments in parentheses, and a colon. Inside the function is a group of actions or calculations and an optional **`return`** statement. Some functions do not return a value, e.g. **`print()`** and **`sleep()`**, but many do, e.g. **`abs()`** and **`format()`**. When a function with arguments is called, the names of the variables in the call and the definition do not need to be the same, but they need to be in the same order.

[Switch to the main.py tab]

```
x = 3
y = 4
def hypotenuse(leg_1, leg_2):  # Function name: "hypotenuse"; Arguments: "leg_1" and "leg_2"
    hyp = (leg_1 ** 2 + leg_2 ** 2) ** 0.5  # Pythagorean theorem
    return hyp  # Return the result
print(hypotenuse(x, y))  # "x" will become "leg_1", and "y" will become "leg_2"
```

The variables inside a function definition are called ***local*** variables, and the variables outside of it are called ***global*** variables. In the previous example, **`hypotenuse`**, **`leg_1`**, and **`leg_2`** are local variables, and **`x`** and **`y`** are global variables. The group of variables to which a section of code has access is called that section's ***scope***. The scopes of everything so far (if statements, for loops, and while loops) have not been limited. Conversely, the scope of a function definition is its own little world; it normally only has access to local variables. If you desire to change the value of a global variable **`V`** from within a function, use the command **`global V`** at the beginning of the function. Try running the following code first with line 3 commented out (the line with the **`global`** command) and then again after uncommenting it, and compare the results.

```
var = 1  # Global variable
def function():
    global var  # Indicates that "var" should reference the global variable "var"
    var = 2  # Local variable unless "global var" has been run
    print(var, "local")
function()
print(var, "global")
```

In spaceship.py, remove **`location`** as a global variable (leave the rest), since it will be used as a local variable. Encase the while loop in a function **`travel()`**, with the arguments of **`ship_name`**, **`captain`**, and **`location`**. After that, call **`travel()`** with the arguments of **`ship_name`**, **`captain`**, and **`3`** (which will become **`location`** locally).

[Switch to the spaceship.py tab]

```
...
          "Saturn", "Uranus", "Neptune", "Pluto"]
def travel(ship_name, captain, location):
    while location != 0:
        ...
travel(ship_name, captain, 3)
```

Threading

Everything until now has been linear; one line of code runs, then the next, then the next. Sometimes multiple things need to happen at the same time, and that is what ***threading*** is for. When you are listening to music and doing homework in Word on your laptop at the same time, your computer is running your audio player and Word in two different threads. Without threading, you would need to write a few words, then listen for a few seconds, then write a few words… and that would be annoying.

A Thread needs to be created before it can be started, and this is done by calling the constructor of the **`Thread`** class, which is in the **`threading`** module. The **`threading`** module is not loaded by default, similar to **`time`**, so it must be imported. The Thread constructor has many arguments, but only a few of them are necessary; the others have default values that are sufficient. Normally constructors and functions must be given an exact number of arguments in the right order, but if they have default values, specific arguments can be changed by assigning their variable names to the desired values inside the constructor or function call. The noteworthy argument names in the **`Thread`** class are **`target`**, which is assigned to the name of the function that you want to run in a different Thread, and **`args`**, which is a list of the arguments to be passed to that function. Call the **`.start()`** method on the Thread to start it.

[Switch to the main.py tab]

```
import threading
import time
def count(amount):
    for i in range(amount):
        print(i)
        time.sleep(1)
threading.Thread(target=count, args=[10]).start()  # Run count(10) in a new Thread
print("Input: {}".format(input()))  # Waits for input at the same time that count() counts
```

This is the last addition to spaceship.py in this tutorial! For long trips, it would be good to run some checks on the spaceship to make sure it is healthy, at the same time that it is traveling. Start by importing the **`threading`** module. Create a function called **`systems_check()`**, which does a health report halfway through a flight if **`eta`** is 8 years (seconds) or longer. Before the travel (sleep) begins, create a new Thread that calls this method.

[Switch to the spaceship.py tab]

```
import threading
def systems_check(eta, captain):
    if eta >= 8:
        time.sleep(eta / 2)
        print("All systems are nominal, {}.".format(captain))
...
def lift_off(ship_name, captain, location):
    ...
            print("Traveling from {} to {}. It will take {:.1f} years.".format(
                    planets[self.location], planets[destination], eta))
            threading.Thread(target=systems_check, args=[eta, captain]).start()
            time.sleep(eta)
            ...
lift_off(ship_name, captain, 3)
```