Programming in Python

As you learn Python throughout this course, there are a few things you should keep in mind.

1. Python is case sensitive.
2. Spacing is important.
3. Use error messages to help you learn.

Data Types and Operators

Welcome to this lesson on Data Types and Operators! You'll learn about:

* Data Types: Integers, Floats, Booleans, Strings
* Operators: Arithmetic, Assignment, Comparison, Logical
* Built-In Functions, Type Conversion
* Whitespace and Style Guidelines

Arithmetic Operators

Arithmetic operators

* + Addition
* - Subtraction
* \* Multiplication
* / Division
* % Mod (the remainder after dividing)
* \*\* Exponentiation (note that ^ does not do this operation, as you might have seen in other languages)
* // Divides and rounds down to the nearest integer

The usual order of mathematical operations holds in Python, which you can review in this Math Forum [**page(opens in a new tab)**](http://mathforum.org/dr.math/faq/faq.order.operations.html) if needed.

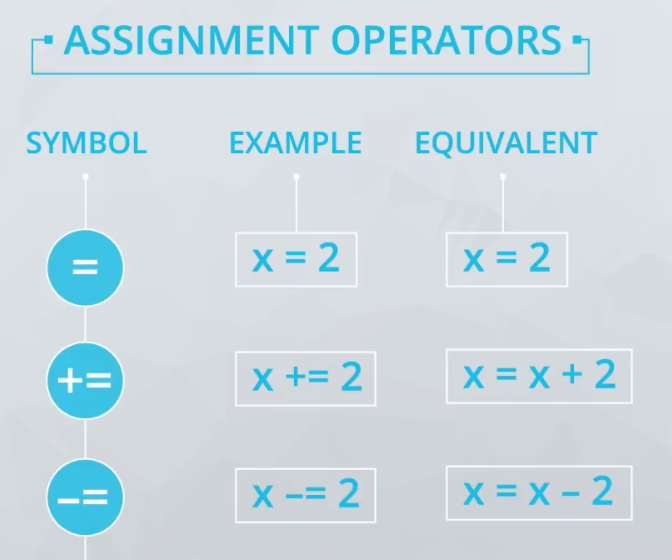
**Bitwise operators** are special operators in Python that you can learn more about [**here(opens in a new tab)**](https://wiki.python.org/moin/BitwiseOperators) if you'd like.

**Examples**

1. print(3 + 5) # 8
2. print(1 + 2 + 3 \* 3) # 12
3. print(3 \*\* 2) # 9
4. print(9 % 2) # 1

Assignment Operators

Below are the assignment operators from the video. You can also use \*= in a similar way, but this is less common than the operations shown below. You can find some practice with much of what we have already covered [**here(opens in a new tab)**](https://www.programiz.com/python-programming/operators).



**Integers and Floats**

There are two videos on this page to discuss integers and floats, as well as some additional helpful notes!

Integers and Floats

There are two Python data types that could be used for numeric values:

* **int** - for integer values
* **float** - for decimal or floating point values

You can create a value that follows the data type by using the following syntax:

x = int(4.7) # x is now an integer 4

y = float(4) # y is now a float of 4.0

You can check the type by using the type function:

>>> **print**(type(x))

int

>>> **print**(type(y))

float

Because the float, or approximation, for 0.1 is actually slightly more than 0.1, when we add several of them together we can see the difference between the mathematically correct answer and the one that Python creates.

>>> **print**(.1 + .1 + .1 == .3)

False

**Booleans, Comparison Operators, and Logical Operators**

The bool data type holds one of the values True or False, which are often encoded as 1 or 0, respectively.

There are 6 comparison operators that are common to see in order to obtain a bool value:

Comparison Operators

| **Symbol Use Case** | **Bool** | **Operation** |
| --- | --- | --- |
| 5 < 3 | False | Less Than |
| 5 > 3 | True | Greater Than |
| 3 <= 3 | True | Less Than or Equal To |
| 3 >= 5 | False | Greater Than or Equal To |
| 3 == 5 | False | Equal To |
| 3 != 5 | True | Not Equal To |

And there are three logical operators you need to be familiar with:

| **Logical Use** | **Bool** | **Operation** |
| --- | --- | --- |
| 5 < 3 and 5 == 5 | False | and - Evaluates if all provided statements are True |
| 5 < 3 or 5 == 5 | True | or - Evaluates if at least one of many statements is True |
| not 5 < 3 | True | not - Flips the Bool Value |

**String Methods**

In this video you were introduced to **methods**. **Methods** are like some of the **functions** you have already seen:

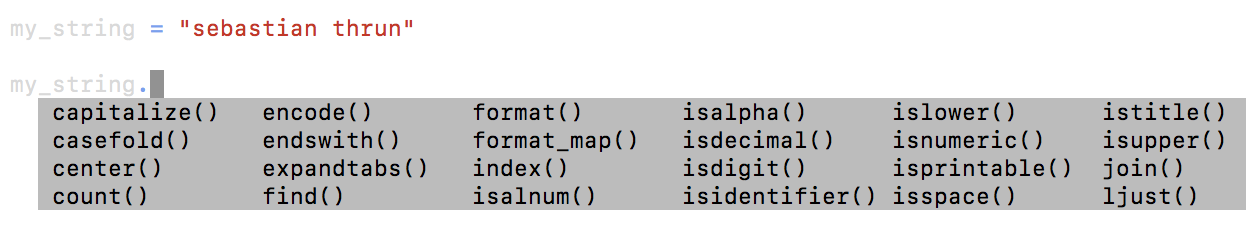
1. len("this")
2. type(12)
3. print("Hello world")

These three above are **functions** - notice they use parentheses, and accept one or more **arguments**. Functions will be studied in much more detail in a later lesson!

A **method** in Python behaves similarly to a function. Methods actually are functions that are called using dot notation. For example, lower() is a string method that can be used like this, on a string called "sample string": sample\_string.lower().

Methods are specific to the data type for a particular variable. So there are some built-in methods that are available for all strings, different methods that are available for all integers, etc.

Below is an image that shows some methods that are possible with any string.



Each of these methods accepts the string itself as the first argument of the method. However, they also could receive additional arguments, that are passed inside the parentheses. Let's look at the output for a few examples.

>>> my\_string.islower()

True

>>> my\_string.count('a')

2

>>> my\_string.find('a')

3

You can see that the count and find methods both take another argument. However, the .islower() method does not accept another argument.

No professional has all the methods memorized, which is why understanding how to use documentation and find answers is so important. Gaining a strong grasp of the foundations of programming will allow you to use those foundations to use documentation to build so much more than someone who tries to memorize all the built-in methods in Python.

**One important string method: format()**

We will be using the format() string method a good bit in our future work in Python, and you will find it very valuable in your coding, especially with your print statements.

We can best illustrate how to use format() by looking at some examples:

Another important string method: split()

A helpful string method when working with strings is the .split method. This function or method returns a data container called a **list** that contains the words from the input string. We will be introducing you to the concept of lists in the next video.

The split method has two additional arguments (*sep* and *maxsplit*). The *sep* argument stands for "separator". It can be used to identify how the string should be split up (e.g., whitespace characters like space, tab, return, newline; specific punctuation (e.g., comma, dashes)). If the *sep* argument is not provided, the default separator is whitespace.

True to its name, the *maxsplit* argument provides the maximum number of splits. The argument gives maxsplit + 1 number of elements in the new list, with the remaining string being returned as the last element in the list. You can read more about these methods in the Python documentation too.

Here are some examples for the .split() method.

1. A basic split method:

new\_str = "The cow jumped over the moon."

new\_str.split()```

Output is:

```Python

['The', 'cow', 'jumped', 'over', 'the', 'moon.']```

2. Here the separator is space, and the maxsplit argument is set to 3.

```Python

new\_str.split(' ', 3) ```

Output is:

```Python

['The', 'cow', 'jumped', 'over the moon.']```

3. Using '.' or period as a separator.

```Python

new\_str.split('.')```

Output is:

```Python

['The cow jumped over the moon', '']```

4. Using no separators but having a maxsplit argument of 3.

```Python

new\_str.split(None, 3)```

Output is:

```Python

['The', 'cow', 'jumped', 'over the moon.']```

Lesson Summary

You learned a ton in this lesson! To summarize, here's a recap of the data types and operators we covered.

**Data Types**

We covered four important data types that you'll use all the time in programming:

| **Data Type** | **Constructor** | **Example** | |---|---|---|---|---| | int | int() | 5 | | float | float() | 6.5 | | string | '' or "" or str() | "this is a string" | | bool | bool() | True or False |

**Operators**

We also covered four useful sets of operators:

A diagram of mathematical operations

AI-generated content may be incorrect.

A diagram of a keyword

AI-generated content may be incorrect.

What's Next?

Now that you are familiar with some basic data types and operators, in the next lesson, you'll learn about **data structures**, where you organize and group together these data types into different containers. You'll also learn about the two remaining types of operators in Python, along with more useful built-in functions and methods.

Additional Practice Resources

When many students are getting started, they often always want more practice. There are a number of great websites you can use for coding exercises and solutions. Two that you should definitely take advantage of are **[HackerRank(opens in a new tab)](https://www.hackerrank.com/domains/python" \t "_blank)** and **[Codewars(opens in a new tab)](https://www.codewars.com/dashboard" \t "_blank)**.

**Note:** You may find some of the exercises require knowledge on concepts you haven't learned yet. Feel free to google them, or wait until you've gone through all the lessons in this course.

I encourage you to create a profile on both and commit to improving your Python programming skills! As you get better, you can advance to harder problems and sites with even greater challenges. If you spend a lot of time on this, you'll really become a Python programming master.

**Data Analysis**

Data Structures

Welcome to this lesson on Data Structures! You'll learn about:

* Types of Data Structures: Lists, Tuples, Sets, Dictionaries, Compound Data Structures
* Operators: Membership, Identity
* Built-In Functions or Methods

**List and Membership Operators**

There are three videos as a part of this page. Be sure to check them out along with the additional helpful reminders!

**Examples**

months = ['January', 'February', 'March', 'April', 'May', 'June', 'July', 'August', 'September', 'October', 'November', 'December']

print(months[0]) # January  
print(months[1]) # February  
print(months[7]) # August  
print(months[-1]) # December  
print(months[25]) # IndexError: list index out of range

Lists!

**Data structures** are containers that organize and group data types together in different ways. A **list** is one of the most common and basic data structures in Python.

You saw here that you can create a list with square brackets. Lists can contain any mix and match of the data types you have seen so far.

list\_of\_random\_things = [1, 3.4, 'a string', True]

This is a list of 4 elements. All ordered containers (like lists) are indexed in python using a starting index of 0. Therefore, to pull the first value from the above list, we can write:

>>> list\_of\_random\_things[0]

1

It might seem like you can pull the last element with the following code, but this actually won't work:

>>> list\_of\_random\_things[**len**(list\_of\_random\_things)]

*---------------------------------------------------------------------------*

IndexError Traceback (most recent **call** **last**)

<ipython-input-34-f88b03e5c60e> in <module>()

*----> 1 lst[len(lst)]*

IndexError: list **index** **out** **of** range

However, you can retrieve the last element by reducing the index by 1. Therefore, you can do the following:

>>> list\_of\_random\_things[len(list\_of\_random\_things) - 1]

True

Alternatively, you can index from the end of a list by using negative values, where -1 is the last element, -2 is the second to last element and so on.

>>> list\_of\_random\_things[-1]

True

>>> list\_of\_random\_things[-2]

a string

Mutability and Order

**Mutability** is about whether or not we can change an object once it has been created. If an object (like a list or string) can be changed (like a list can), then it is called **mutable**. However, if an object cannot be changed with creating a completely new object (like strings), then the object is considered **immutable**.

>>> my\_lst = [1, 2, 3, 4, 5]

>>> my\_lst[0] = 'one'

>>> **print**(my\_lst)

['one', 2, 3, 4, 5]

As shown above, you are able to replace 1 with 'one' in the above list. This is because lists are **mutable**.

However, the following does not work:

>>> greeting = "Hello there"

>>> greeting[0] = 'M'

This is because strings are **immutable**. This means to change this string, you will need to create a completely new string.

There are two things to keep in mind for each of the data types you are using:

1. Are they **mutable**?
2. Are they **ordered**?

**Order** is about whether the position of an element in the object can be used to access the element. **Both strings and lists are ordered.** We can use the order to access parts of a list and string.

However, you will see some data types in the next sections that will be unordered. For each of the upcoming data structures you see, it is useful to understand how you index, are they mutable, and are they ordered. Knowing this about the data structure is really useful!

Additionally, you will see how these each have different methods, so why you would use one data structure vs. another is largely dependent on these properties, and what you can easily do with it!

**Useful Functions for Lists I**

1. len() returns how many elements are in a list.
2. max() returns the greatest element of the list. How the greatest element is determined depends on what type objects are in the list. The maximum element in a list of numbers is the largest number. The maximum elements in a list of strings is element that would occur last if the list were sorted alphabetically. This works because the the max function is defined in terms of the greater than comparison operator. The max function is undefined for lists that contain elements from different, incomparable types.
3. min() returns the smallest element in a list. min is the opposite of max, which returns the largest element in a list.
4. sorted() returns a copy of a list in order from smallest to largest, leaving the list unchanged.
5. Useful Functions for Lists II
6. **join method**
7. Join is a string method that takes a list of strings as an argument, and returns a string consisting of the list elements joined by a separator string.
8. new\_str = "\n".join(["fore", "aft", "starboard", "port"])
9. **print**(new\_str)
10. Output:
11. fore
12. aft
13. starboard
14. port
15. In this example we use the string "\n" as the separator so that there is a newline between each element. We can also use other strings as separators with .join. Here we use a hyphen.
16. name = "-".join(["García", "O'Kelly"])
17. **print**(name)
18. Output:
19. García-O'Kelly
20. It is important to remember to separate each of the items in the list you are joining with a comma (,). Forgetting to do so will not trigger an error, but will also give you unexpected results.
21. **append method**
22. A helpful method called append adds an element to the end of a list.
23. letters = ['a', 'b', 'c', 'd']
24. letters.append('z')
25. **print**(letters)
26. Output:
27. ['a', 'b', 'c', 'd', 'z']
28. Try It Out!

**Tuples**

A tuple is another useful container. It's a data type for immutable ordered sequences of elements. They are often used to store related pieces of information. Consider this example involving latitude and longitude:

location = (13.4125, 103.866667)

**print**("Latitude:", location[0])

**print**("Longitude:", location[1])

Tuples are similar to lists in that they store an ordered collection of objects which can be accessed by their indices. Unlike lists, however, tuples are immutable - you can't add and remove items from tuples, or sort them in place.

Tuples can also be used to assign multiple variables in a compact way.

dimensions = 52, 40, 100

length, width, height = dimensions

**print**("The dimensions are {} x {} x {}".format(length, width, height))

The parentheses are optional when defining tuples, and programmers frequently omit them if parentheses don't clarify the code.

In the second line, three variables are assigned from the content of the tuple dimensions. This is called **tuple unpacking**. You can use tuple unpacking to assign the information from a tuple into multiple variables without having to access them one by one and make multiple assignment statements.

If we won't need to use dimensions directly, we could shorten those two lines of code into a single line that assigns three variables in one go!

length, width, height = 52, 40, 100

**print**("The dimensions are {} x {} x {}".format(length, width, height))

**Sets**

A **set** is a data type for mutable unordered collections of unique elements. One application of a set is to quickly remove duplicates from a list.

numbers = [1, 2, 6, 3, 1, 1, 6]

unique\_nums = set(numbers)

**print**(unique\_nums)

This would output:

{1, 2, 3, 6}

Sets support the in operator the same as lists do. You can add elements to sets using the add method, and remove elements using the pop method, similar to lists. Although, when you pop an element from a set, a random element is removed. Remember that sets, unlike lists, are unordered so there is no "last element".

fruit = {"apple", "banana", "orange", "grapefruit"} *# define a set*

**print**("watermelon" **in** fruit) *# check for element*

fruit.add("watermelon") *# add an element*

**print**(fruit)

**print**(fruit.pop()) *# remove a random element*

**print**(fruit)

This outputs:

False

{'grapefruit', 'orange', 'watermelon', 'banana', 'apple'}

grapefruit

{'orange', 'watermelon', 'banana', 'apple'}

Other operations you can perform with sets include those of mathematical sets. Methods like union, intersection, and difference are easy to perform with sets, and are much faster than such operators with other containers.

**Dictionaries and Identity Operators**

**Dictionaries**

A **dictionary** is a mutable data type that stores mappings of unique keys to values. Here's a dictionary that stores elements and their atomic numbers.

elements = {"hydrogen": 1, "helium": 2, "carbon": 6}

Dictionaries can have keys of any immutable type, like integers or tuples, not just strings. It's not even necessary for every key to have the same type! We can look up values or insert new values in the dictionary using square brackets that enclose the key.

**print**(elements["helium"]) *# print the value mapped to "helium"*

elements["lithium"] = 3 *# insert "lithium" with a value of 3 into the dictionary*

We can check whether a value is in a dictionary the same way we check whether a value is in a list or set with the in keyword. Dicts have a related method that's also useful, get. get looks up values in a dictionary, but unlike square brackets, get returns None (or a default value of your choice) if the key isn't found.

**print**("carbon" **in** elements)

**print**(elements.get("dilithium"))

This would output:

True

None

Carbon is in the dictionary, so True is printed. Dilithium isn’t in our dictionary so None is returned by get and then printed. If you expect lookups to sometimes fail, get might be a better tool than normal square bracket lookups because errors can crash your program.

**Identity Operators**

| **Keyword** | **Operator** |
| --- | --- |
| is | evaluates if both sides have the same identity |
| is not | evaluates if both sides have different identities |

You can check if a key returned None with the is operator. You can check for the opposite using is not.

n = elements.get("dilithium")

**print**(n **is** None)

**print**(n **is** **not** None)

This would output:

True

False

Compound Data Structures

We can include containers in other containers to create compound data structures. For example, this dictionary maps keys to values that are also dictionaries!

elements = {"hydrogen": {"number": 1,

"weight": 1.00794,

"symbol": "H"},

"helium": {"number": 2,

"weight": 4.002602,

"symbol": "He"}}

We can access elements in this nested dictionary like this.

helium = elements["helium"] *# get the helium dictionary*

hydrogen\_weight = elements["hydrogen"]["weight"] *# get hydrogen's weight*

You can also add a new key to the element dictionary.

oxygen = {"number":8,"weight":15.999,"symbol":"O"} *# create a new oxygen dictionary*

elements["oxygen"] = oxygen *# assign 'oxygen' as a key to the elements dictionary*

**print**('elements = ', elements)

Output is:

elements = {"hydrogen": {"number": 1,

"weight": 1.00794,

"symbol": 'H'},

"helium": {"number": 2,

"weight": 4.002602,

"symbol": "He"},

"oxygen": {"number": 8,

"weight": 15.999,

"symbol": "O"}}```

In this lesson, we covered four important data structures in Python:

| **Data Structure** | **Ordered** | **Mutable** | **Constructor** | **Example** |
| --- | --- | --- | --- | --- |
| List | Yes | Yes | [ ] or list() | [5.7, 4, 'yes', 5.7] |
| Tuple | Yes | No | ( ) or tuple() | (5.7, 4, 'yes', 5.7) |
| Set | No | Yes | {}\* or set() | {5.7, 4, 'yes'} |
| Dictionary | No | No\*\* | { } or dict() | {'Jun': 75, 'Jul': 89} |

\* You can use curly braces to define a set like this: {1, 2, 3}. However, if you leave the curly braces empty like this: {} Python will instead create an empty dictionary. So to create an empty set, use set().  
\*\* A dictionary itself is mutable, but each of its individual keys must be immutable.

**Control Flow**

Welcome to this lesson on Control Flow! Control flow is the sequence in which your code is run. Here, we'll learn about several tools in Python we can use to affect our code's control flow:

* Conditional Statements
* Boolean Expressions
* For and While Loops
* Break and Continue
* Zip and Enumerate
* List Comprehensions

**Conditional Statements**

If Statement

An if statement is a conditional statement that runs or skips code based on whether a condition is true or false. Here's a simple example.

**if** phone\_balance < 5:

phone\_balance += 10

bank\_balance -= 10

Let's break this down.

1. An if statement starts with the if keyword, followed by the condition to be checked, in this case phone\_balance < 5, and then a colon. The condition is specified in a boolean expression that evaluates to either True or False.
2. After this line is an indented block of code to be executed if that condition is true. Here, the lines that increment phone\_balance and decrement bank\_balance only execute if it is true that phone\_balance is less than 5. If not, the code in this if block is simply skipped.

**Use Comparison Operators in Conditional Statements**

You have learned about Python's comparison operators (e.g. == and !=) and how they are different from assignment operators (e.g. =). In conditional statements, you want to use comparison operators. For example, you'd want to use if x == 5 rather than if x = 5. If your conditional statement is causing a syntax error or doing something unexpected, check whether you have written == or =!

If, Elif, Else

In addition to the if clause, there are two other optional clauses often used with an if statement. For example:

**if** season == 'spring':

**print**('plant the garden!')

**elif** season == 'summer':

**print**('water the garden!')

**elif** season == 'fall':

**print**('harvest the garden!')

**elif** season == 'winter':

**print**('stay indoors!')

**else**:

**print**('unrecognized season')

1. if: An if statement must always start with an if clause, which contains the first condition that is checked. If this evaluates to True, Python runs the code indented in this if block and then skips to the rest of the code after the if statement.
2. elif: elif is short for "else if." An elif clause is used to check for an additional condition if the conditions in the previous clauses in the if statement evaluate to False. As you can see in the example, you can have multiple elif blocks to handle different situations.
3. else: Last is the else clause, which must come at the end of an if statement if used. This clause doesn't require a condition. The code in an else block is run if all conditions above that in the if statement evaluate to False.

**Indentation**

Some other languages use braces to show where blocks of code begin and end. In Python we use indentation to enclose blocks of code. For example, if statements use indentation to tell Python what code is inside and outside of different clauses.

In Python, indents conventionally come in multiples of four spaces. Be strict about following this convention, because changing the indentation can completely change the meaning of the code. If you are working on a team of Python programmers, it's important that everyone follows the same indentation convention!

**Spaces or Tabs?**

The [**Python Style Guide(opens in a new tab)**](https://www.python.org/dev/peps/pep-0008/#tabs-or-spaces) recommends using 4 spaces to indent, rather than using a tab. Whichever you use, be aware that "Python 3 disallows mixing the use of tabs and spaces for indentation."

E.g

points = 174

**if** points <= 50:

result = "Congratulations! You won a wooden rabbit!"

**elif** points <= 150:

result = "Oh dear, no prize this time."

**elif** points <= 180:

result = "Congratulations! You won a wafer-thin mint!"

**else**:

result = "Congratulations! You won a penguin!"

**print**(result)

**Output:**

Congratulations! You won a wafer-thin mint!

We use <= instead of the < operator, since it was stated that the upper bound is inclusive. Notice that in each condition, we check if points is in a prize bracket by checking if points is less than or equal to the upper bound; we didn't have to check if it was greater than the lower bound. Let's see why this is the case.

* When points = 174, it first checks if points <= 50, which evaluates to False. We don't have to check if it is also greater than 0, since it is stated in the problem that points will always be a positive integer up to 200.
* Since the first condition evaluates to False, it moves on to check the next condition, points <= 150. **We don't need to check if it is also greater than 50 here!** We already know this is the case because the first condition has to have evaluated to False in order to get to this point. If we know points <= 50 is False, then points > 50 must be True!
* Finally, we check if points <= 180, which evaluates to True. We now know that points is in the 151 - 180 bracket.
* The last prize bracket, 181-200, is caught in the else clause, since there is no other possible value of the prize after checking the previous conditions.

**Complex Boolean Expressions**

If statements sometimes use more complicated boolean expressions for their conditions. They may contain multiple comparisons operators, logical operators, and even calculations. Examples:

**if** 18.5 <= weight / height\*\*2 < 25:

**print**("BMI is considered 'normal'")

**if** is\_raining **and** is\_sunny:

**print**("Is there a rainbow?")

**if** (**not** unsubscribed) **and** (location == "USA" **or** location == "CAN"):

**print**("send email")

For really complicated conditions you might need to combine some ands, ors and nots together. Use parentheses if you need to make the combinations clear.

However simple or complex, the condition in an if statement must be a boolean expression that evaluates to either True or False and it is this value that decides whether the indented block in an if statement executes or not.

Quiz Solution: Evaluate composed boolean expressions

altitude < 1000 **and** speed > 100

altitude < 1000 is False, so we don't even need to check the second condition - the whole expression is False.

(propulsion == "Jet" **or** propulsion == "Turboprop") **and** speed < 300 **and** altitude > 20000

propulsion == "Jet" is False, and propulsion == "Turboprop" is False, so the whole expression inside the parentheses is False. It is combined with the other expressions with and, so we don't even need to check these - the whole expression must be False because the first part is False.

**not** (speed > 400 **and** propulsion == "Propeller")

To work this one out, we need to look at the inside of the parentheses first, then apply not to that. speed > 400 is False, and because we are using and this makes the whole of the expression inside the parentheses False. Applying not reverses this, so this expression is True.

(altitude > 500 **and** speed > 100) **or** **not** propulsion == "Propeller"

Let's start by looking inside the parentheses. altitude > 500 is True, and speed is greater than 100, so the expression inside the parenthesis is True. Whatever the value of the other expression, because they are connected by or, the whole expression will evaluate to True.

**For Loops**

Python has two kinds of loops - for loops and while loops. A for loop is used to "iterate", or do something repeatedly, over an **iterable**.

An **iterable** is an object that can return one of its elements at a time. This can include sequence types, such as strings, lists, and tuples, as well as non-sequence types, such as dictionaries and files.

**Example**

Let's break down the components of a for loop, using this example with the list cities:

cities = ['new york city', 'mountain view', 'chicago', 'los angeles']

**for** city **in** cities:

**print**(city)

**print**("Done!")

**Components of a for Loop**

1. The first line of the loop starts with the for keyword, which signals that this is a for loop
2. Following that is city in cities, indicating city is the iteration variable, and cities is the iterable being looped over. In the first iteration of the loop, city gets the value of the first element in cities, which is “new york city”.
3. The for loop heading line always ends with a colon :
4. Following the for loop heading is an indented block of code, the body of the loop, to be executed in each iteration of this loop. There is only one line in the body of this loop - print(city).
5. After the body of the loop has executed, we don't move on to the next line yet; we go back to the for heading line, where the iteration variable takes the value of the next element of the iterable. In the second iteration of the loop above, city takes the value of the next element in cities, which is "mountain view".
6. This process repeats until the loop has iterated through all the elements of the iterable. Then, we move on to the line that follows the body of the loop - in this case, print("Done!"). We can tell what the next line after the body of the loop is because it is unindented. Here is another reason why paying attention to your indentation is very important in Python!

Executing the code in the example above produces this output:

new york city

mountain view

chicago

los angeles

Done!

You can name iteration variables however you like. A common pattern is to give the iteration variable and iterable the same names, except the singular and plural versions respectively (e.g., 'city' and 'cities').

**Using the range() Function with for Loops**

range() is a built-in function used to create an iterable sequence of numbers. You will frequently use range() with a for loop to repeat an action a certain number of times. Any variable can be used to iterate through the numbers, but Python programmers conventionally use i, as in this example:

**for** i **in** range(3):

**print**("Hello!")

Output:

Hello!

Hello!

Hello!

**range(start=0, stop, step=1)**

The range() function takes three integer arguments, the first and third of which are optional:

* The 'start' argument is the first number of the sequence. If unspecified, 'start' defaults to 0.
* The 'stop' argument is 1 more than the last number of the sequence. This argument must be specified.
* The 'step' argument is the difference between each number in the sequence. If unspecified, 'step' defaults to 1.

Notes on using range():

* If you specify one integer inside the parentheses withrange(), it's used as the value for 'stop,' and the defaults are used for the other two.  
  **e.g. -** range(4) returns 0, 1, 2, 3
* If you specify two integers inside the parentheses withrange(), they're used for 'start' and 'stop,' and the default is used for 'step.'  
  **e.g. -** range(2, 6) returns 2, 3, 4, 5
* Or you can specify all three integers for 'start', 'stop', and 'step.'  
  **e.g. -** range(1, 10, 2) returns 1, 3, 5, 7, 9

**Creating and Modifying Lists**

In addition to extracting information from lists, as we did in the first example above, you can also create and modify lists with for loops. You can **create** a list by appending to a new list at each iteration of the for loop like this:

*## Creating a new list*

cities = ['new york city', 'mountain view', 'chicago', 'los angeles']

capitalized\_cities = []

**for** city **in** cities:

capitalized\_cities.append(city.title())

**Modifying** a list is a bit more involved, and requires the use of the range() function.

We can use the range() function to generate the indices for each value in the cities list. This lets us access the elements of the list with cities[index] so that we can modify the values in the cities list in place.

cities = ['new york city', 'mountain view', 'chicago', 'los angeles']

**for** index **in** range(len(cities)):

cities[index] = cities[index].title()

**Iterating Through Dictionaries with For Loops**

When you iterate through a dictionary using a for loop, doing it the normal way (for n in some\_dict) will only give you access to the **keys** in the dictionary - which is what you'd want in some situations. In other cases, you'd want to iterate through both the **keys** and **values** in the dictionary. Let's see how this is done in an example. Consider this dictionary that uses names of actors as keys and their characters as values.

cast = {

"Jerry Seinfeld": "Jerry Seinfeld",

"Julia Louis-Dreyfus": "Elaine Benes",

"Jason Alexander": "George Costanza",

"Michael Richards": "Cosmo Kramer"

}

Iterating through it in the usual way with a for loop would give you just the keys, as shown below:

**for** key **in** cast:

**print**(key)

This outputs:

Jerry Seinfeld

Julia Louis-Dreyfus

Jason Alexander

Michael Richards

If you wish to iterate through both keys and values, you can use the built-in method items like this:

**for** key, value **in** cast.items():

**print**("Actor: {} Role: {}".format(key, value))

This outputs:

Actor: Jerry Seinfeld Role: Jerry Seinfeld

Actor: Julia Louis-Dreyfus Role: Elaine Benes

Actor: Jason Alexander Role: George Costanza

Actor: Michael Richards Role: Cosmo Kramer

items is an awesome method that returns tuples of key, value pairs, which you can use to iterate over dictionaries in for loops.

**While Loops**

For loops are an example of "definite iteration" meaning that the loop's body is run a predefined number of times. This differs from "indefinite iteration" which is when a loop repeats an unknown number of times and ends when some condition is met, which is what happens in a while loop. Here's an example of a while loop.

card\_deck = [4, 11, 8, 5, 13, 2, 8, 10]

hand = []

*## adds the last element of the card\_deck list to the hand list*

*## until the values in hand add up to 17 or more*

**while** sum(hand) < 17:

hand.append(card\_deck.pop())

This example features two new functions. sum returns the sum of the elements in a list, and pop is a list method that removes the last element from a list and returns it.

**Components of a While Loop**

1. The first line starts with the while keyword, indicating this is a while loop.
2. Following that is a condition to be checked. In this example, that's sum(hand) <= 17.
3. The while loop heading always ends with a colon :.
4. Indented after this heading is the body of the while loop. If the condition for the while loop is true, the code lines in the loop's body will be executed.
5. We then go back to the while heading line, and the condition is evaluated again. This process of checking the condition and then executing the loop repeats until the condition becomes false.
6. When the condition becomes false, we move on to the line following the body of the loop, which will be unindented.

The indented body of the loop should modify at least one variable in the test condition. If the value of the test condition never changes, the result is an infinite loop!

While Loops

For loops are an example of "definite iteration" meaning that the loop's body is run a predefined number of times. This differs from "indefinite iteration" which is when a loop repeats an unknown number of times and ends when some condition is met, which is what happens in a while loop. Here's an example of a while loop.

card\_deck = [4, 11, 8, 5, 13, 2, 8, 10]

hand = []

*## adds the last element of the card\_deck list to the hand list*

*## until the values in hand add up to 17 or more*

**while** sum(hand) < 17:

hand.append(card\_deck.pop())

This example features two new functions. sum returns the sum of the elements in a list, and pop is a list method that removes the last element from a list and returns it.

**Components of a While Loop**

1. The first line starts with the while keyword, indicating this is a while loop.
2. Following that is a condition to be checked. In this example, that's sum(hand) <= 17.
3. The while loop heading always ends with a colon :.
4. Indented after this heading is the body of the while loop. If the condition for the while loop is true, the code lines in the loop's body will be executed.
5. We then go back to the while heading line, and the condition is evaluated again. This process of checking the condition and then executing the loop repeats until the condition becomes false.
6. When the condition becomes false, we move on to the line following the body of the loop, which will be unindented.

The indented body of the loop should modify at least one variable in the test condition. If the value of the test condition never changes, the result is an infinite loop!

For Loops Vs. While Loops

Now that you are familiar with both for and while loops, let's consider when it's most helpful to use each of them.

**for** loops are ideal when the **number of iterations is known or finite**.

Examples:

* When you have an iterable collection (list, string, set, tuple, dictionary)
  + for name in names:
* When you want to iterate through a loop for a definite number of times, using range()
  + for i in range(5):

**while** loops are ideal when the **iterations need to continue until a condition is met**.

Examples:

* When you want to use comparison operators
  + while count <= 100:
* When you want to loop based on receiving specific user input.
  + while user\_input == 'y':

Break, Continue

Sometimes we need more control over when a loop should end, or skip an iteration. In these cases, we use the break and continue keywords, which can be used in both for and while loops.

* break terminates a loop
* continue skips one iteration of a loop

Watch the video and experiment with the examples below to see how these can be helpful.

headlines = ["Local Bear Eaten by Man",

"Legislature Announces New Laws",

"Peasant Discovers Violence Inherent in System",

"Cat Rescues Fireman Stuck in Tree",

"Brave Knight Runs Away",

"Papperbok Review: Totally Triffic"]

news\_ticker = ""

**for** headline **in** headlines:

news\_ticker += headline + " "

**if** len(news\_ticker) >= 140:

news\_ticker = news\_ticker[:140]

**break**

**print**(news\_ticker)

**Output:**

Local Bear Eaten by Man Legislature Announces New Laws Peasant Discovers Violence Inherent in System Cat Rescues Fireman Stuck in Tree Brave

**Zip and Enumerate**

zip and enumerate are useful built-in functions that can come in handy when dealing with loops.

**Zip**

zip returns an iterator that combines multiple iterables into one sequence of tuples. Each tuple contains the elements in that position from all the iterables. For example, printing

list(zip(['a', 'b', 'c'], [1, 2, 3])) would output [('a', 1), ('b', 2), ('c', 3)].

Like we did for range() we need to convert it to a list or iterate through it with a loop to see the elements.

You could unpack each tuple in a for loop like this.

letters = ['a', 'b', 'c']

nums = [1, 2, 3]

**for** letter, num **in** zip(letters, nums):

**print**("{}: {}".format(letter, num))

In addition to zipping two lists together, you can also unzip a list into tuples using an asterisk.

some\_list = [('a', 1), ('b', 2), ('c', 3)]

letters, nums = zip(\*some\_list)

This would create the same letters and nums tuples we saw earlier.

**Enumerate**

enumerate is a built in function that returns an iterator of tuples containing indices and values of a list. You'll often use this when you want the index along with each element of an iterable in a loop.

letters = ['a', 'b', 'c', 'd', 'e']

**for** i, letter **in** enumerate(letters):

**print**(i, letter)

This code would output:

0 a

1 b

2 c

3 d

4 e

**List Comprehensions**

In Python, you can create lists really quickly and concisely with list comprehensions. This example from earlier:

capitalized\_cities = []

**for** city **in** cities:

capitalized\_cities.append(city.title())

can be reduced to:

capitalized\_cities = [city.title() **for** city **in** cities]

List comprehensions allow us to create a list using a for loop in one step.

You create a list comprehension with brackets [], including an expression to evaluate for each element in an iterable. This list comprehension above calls city.title() for each element city in cities, to create each element in the new list, capitalized\_cities.

**Conditionals in List Comprehensions**

You can also add conditionals to list comprehensions (listcomps). After the iterable, you can use the if keyword to check a condition in each iteration.

squares = [x\*\*2 **for** x **in** range(9) **if** x % 2 == 0]

The code above sets squares equal to the list [0, 4, 16, 36, 64], as x to the power of 2 is only evaluated if x is even. If you want to add an else, you will get a syntax error doing this.

squares = [x\*\*2 **for** x **in** range(9) **if** x % 2 == 0 **else** x + 3]

If you would like to add else, you have to move the conditionals to the beginning of the listcomp, right after the expression, like this.

squares = [x\*\*2 **if** x % 2 == 0 **else** x + 3 **for** x **in** range(9)]

List comprehensions are not found in other languages, but are very common in Python.

**Quiz Solution**: Multiples of Three

multiples\_3 = [x \* 3 **for** x **in** range(1, 21)]

**print**(multiples\_3)

**Output:**

[3, 6, 9, 12, 15, 18, 21, 24, 27, 30, 33, 36, 39, 42, 45, 48, 51, 54, 57, 60]

**Functions**

Welcome to this lesson on Functions! You'll learn about:

* Defining Functions
* Variable Scope
* Documentation
* Lambda Expressions
* Iterators and Generators

You can think about functions as a way to take what you have already learned how to do, and put it in a holder that allows you to use it over and over again in an easy to use container.

Defining Functions

Example of a function definition:

**def** **cylinder\_volume**(height, radius):

pi = 3.14159

**return** height \*pi\* radius \*\* 2

After defining the cylinder\_volume function, we can **call** the function like this.

cylinder\_volume(10, 3)

This is called a **function call** statement.

A function definition includes several important parts.

**Function Header**

Let's start with the function header, which is the first line of a function definition.

1. The function header always starts with the def keyword, which indicates that this is a **function definition**.
2. Then comes the **function name** (here, cylinder\_volume), which follows the same naming conventions as variables. You can revisit the naming conventions below.
3. Immediately after the name are *parentheses* that may include **arguments** separated by commas (here, height and radius). Arguments, or **parameters**, are values that are passed in as **inputs** when the function is called, and are used in the function body. If a function doesn't take arguments, these parentheses are left empty.
4. The header always end with a colon :.

**Function Body**

The rest of the function is contained in the body, which is where the function does its work.

1. The **body** of a function is the code indented after the header line. Here, it's the two lines that define pi and return the volume.
2. Within this body, we can refer to the **argument variables** and define new variables, which can only be used within these indented lines.
3. The body will often include a return statement, which is used to send back an **output value** from the function to the statement that called the function. A return statement consists of the return keyword followed by an expression that is evaluated to get the output value for the function. If there is no return statement, the function simply returns None.

Below, you'll find a code editor where you can experiment with this.

**Naming Conventions for Functions**

Function names follow the same naming conventions as variables.

1. Only use ordinary letters, numbers and underscores in your function names. They can’t have spaces, and need to start with a letter or underscore.
2. You can’t use Python's reserved words or keywords for function names, as discussed earlier with variable names. Here again is that [**table of Python's reserved words(opens in a new tab)**](https://docs.python.org/3/reference/lexical_analysis.html#keywords).
3. Try to use descriptive names that can help readers understand what the function does.

**Default Arguments**

We can add default arguments in a function to have default values for parameters that are unspecified in a function call.

**def** **cylinder\_volume**(height, radius=5):

pi = 3.14159

**return** height \*pi\* radius \*\* 2

In the example above, radius is set to 5 if that parameter is omitted in a function call. If we call cylinder\_volume(10), the function will use 10 as the height and 5 as the radius. However, if we call cylinder\_volume(10, 7) the 7 will simply overwrite the default value of 5.

Also notice here we are passing values to our arguments by position. It is possible to pass values in two ways - **by position** and **by name**. Each of these function calls are evaluated the same way.

cylinder\_volume(10, 7) *# pass in arguments by position*

cylinder\_volume(height=10, radius=7) *# pass in arguments by name*

**Variable Scope**

**Variable scope** refers to which parts of a program a variable can be referenced, or used, from.

It's important to consider scope when using variables in functions. If a variable is created inside a function, it can only be used within that function. Accessing it outside that function is not possible.

*## This will result in an error*

**def** **some\_function**():

word = "hello"

**print**(word)

In the example above and the example below, word is said to have scope that is only **local** to each function. This means you can use the same name for different variables that are used in different functions.

*## This works fine*

**def** **some\_function**():

word = "hello"

**def** **another\_function**():

word = "goodbye"

Variables defined outside functions, as in the example below, can still be accessed within a function. Here, word is said to have a **global scope**.

*## This works fine*

word = "hello"

**def** **some\_function**():

**print**(word)

some\_function()```

Notice that we can still access the value of the **global** variable `word` within this function. However, the value of a **global** variable can **not** be \_\_modified\_\_ inside the function. If you want to modify that variable's value inside this function, it should be passed in as an argument. You'll see more on this **in** the next quiz.

Scope **is** essential to understanding how information **is** passed throughout programs **in** Python **and** really any programming language.

More on Variable Scope

When you program, you'll often find that similar ideas come up again and again. You'll use variables for things like counting, iterating and accumulating values to return. In order to write readable code, you'll find yourself wanting to use similar names for similar ideas. As soon as you put multiple piece of code together (for instance, multiple functions or function calls in a single script) you might find that you want to use the same name for two separate concepts.

Fortunately, you don't need to come up with new names endlessly. Reusing names for objects is OK as long as you keep them in separate scope.

**Good practice:** It is best to define variables in the smallest scope they will be needed in. While functions *can* refer to variables defined in a larger scope, this is very rarely a good idea since you may not know what variables you have defined if your program has a lot of variables.

**Documentation**

Documentation is used to make your code easier to understand and use. Functions are especially readable because they often use documentation strings, or docstrings. Docstrings are a type of comment used to explain the purpose of a function, and how it should be used. Here's a function for population density with a docstring.

**def** **population\_density**(population, land\_area):

"""Calculate the population density of an area. """

**return** population / land\_area

Docstrings are surrounded by triple quotes. The first line of the docstring is a brief explanation of the function's purpose. If you feel that this is sufficient documentation you can end the docstring at this point; single line docstrings are perfectly acceptable, as in the example above.

**def** **population\_density**(population, land\_area):

"""Calculate the population density of an area.

INPUT:

population: int. The population of that area

land\_area: int or float. This function is unit-agnostic, if you pass in values in terms

of square km or square miles the function will return a density in those units.

OUTPUT:

population\_density: population / land\_area. The population density of a particular area.

"""

**return** population / land\_area

If you think that a longer description would be appropriate for the function, you can add more information after the one-line summary. In the example above, you can see that we wrote an explanation of the function's arguments, stating the purpose and types of each one. It's also common to provide some description of the function's output.

Every piece of the docstring is optional, however, docstrings are a part of good coding practice. You can read more about docstring conventions [**here(opens in a new tab)**](https://www.python.org/dev/peps/pep-0257).

**Lambda Expressions**

You can use lambda expressions to create anonymous functions. That is, functions that don’t have a name. They are helpful for creating quick functions that aren’t needed later in your code. This can be especially useful for higher order functions, or functions that take in other functions as arguments.

With a lambda expression, this function:

**def** **multiply**(x, y):

**return** x \* y

can be reduced to:

multiply = **lambda** x, y: x \* y

Both of these functions are used in the same way. In either case, we can call multiply like this:

multiply(4, 7)

This returns 28.

**Components of a Lambda Function**

1. The lambda keyword is used to indicate that this is a lambda expression.
2. Following lambda are one or more arguments for the anonymous function separated by commas, followed by a colon :. Similar to functions, the way the arguments are named in a lambda expression is arbitrary.
3. Last is an expression that is evaluated and returned in this function. This is a lot like an expression you might see as a return statement in a function.

With this structure, lambda expressions aren’t ideal for complex functions, but can be very useful for short, simple functions.

**Quiz Solution: Lambda with Map**

numbers = [

[34, 63, 88, 71, 29],

[90, 78, 51, 27, 45],

[63, 37, 85, 46, 22],

[51, 22, 34, 11, 18]

]

averages = list(map(**lambda** x: sum(x) / len(x), numbers))

**print**(averages)

**Output:**

[57.0, 58.2, 50.6, 27.2]

Quiz Solution: Lambda with Filter

cities = ["New York City", "Los Angeles", "Chicago", "Mountain View", "Denver", "Boston"]

short\_cities = list(filter(**lambda** x: len(x) < 10, cities))

**print**(short\_cities)

**Output:**

['Chicago', 'Denver', 'Boston']

**Scripting**

Welcome to this lesson on scripting! You’ll learn about:

* Python Installation and Environment Setup
* Running and Editing Python Scripts
* Interacting with User Input
* Handling Exceptions
* Reading and Writing Files
* Importing Local, Standard, and Third-Party Modules
* Experimenting with an Interpreter

Run a Python Script!

1. Download the zip file first\_script attached at the bottom of this page (click it to unzip the file, then move it to an appropriate directory on your computer). This might be a good time to set up a new directory for your learning if you don't have one already.
2. Open your terminal and use cd to navigate to the directory containing that downloaded file.
3. Now that you’re in the directory with the file, you can run it by typing python first\_script.py and pressing enter. Note: You may have to enter python3 instead of python to execute Python 3 if you have both versions installed on your computer.

You’ll know you’ve run the script successfully if you see this message printed to your terminal:

Congratulations on running this script!!

**Supporting Materials**

* [**first\_script**](https://video.udacity-data.com/topher/2018/March/5aa86fd0_first-script.py/first-script.py.zip)

Scripting with Raw Input

We can get raw input from the user with the built-in function input, which takes in an optional string argument that you can use to specify a message to show to the user when asking for input.

name = input("Enter your name: ")

**print**("Hello there, {}!".format(name.title()))

This prompts the user to enter a name and then uses the input in a greeting. The input function takes in whatever the user types and stores it as a string. If you want to interpret their input as something other than a string, like an integer, as in the example below, you need to wrap the result with the new type to convert it from a string.

num = int(input("Enter an integer"))

**print**("hello" \* num)

We can also interpret user input as a Python expression using the built-in function eval. This function evaluates a string as a line of Python.

result = eval(input("Enter an expression: "))

**print**(result)

If the user inputs 2 \* 3, this outputs 6.

Errors and Exceptions

* **Syntax errors** occur when Python can’t interpret our code, since we didn’t follow the correct syntax for Python. These are errors you’re likely to get when you make a typo, or you’re first starting to learn Python.
* **Exceptions** occur when unexpected things happen during execution of a program, even if the code is syntactically correct. There are different types of built-in exceptions in Python, and you can see which exception is thrown in the error message.
* When you handle an exception, you can still access its error message like this:
* **try**:
* *# some code*
* **except** ZeroDivisionError **as** e:
* *# some code*
* **print**("ZeroDivisionError occurred: {}".format(e))
* This would print something like this:
* ZeroDivisionError occurred: integer division or modulo by zero
* So you can still access error messages, even if you handle them to keep your program from crashing!
* If you don't have a specific error you're handling, you can still access the message like this:
* **try**:
* *# some code*
* **except** Exception **as** e:
* *# some code*
* **print**("Exception occurred: {}".format(e))
* Exception is just the base class for all built-in exceptions. You can learn more about Python's exceptions [**here(opens in a new tab)**](https://docs.python.org/3/library/exceptions.html#bltin-exceptions).
* Importing Local Scripts
* We can actually import Python code from other scripts, which is helpful if you are working on a bigger project where you want to organize your code into multiple files and reuse code in those files. If the Python script you want to import is in the same directory as your current script, you just type import followed by the name of the file, without the .py extension.
* **import** useful\_functions
* It's the standard convention for import statements to be written at the top of a Python script, each one on a separate line. This import statement creates a **module** object called useful\_functions. Modules are just Python files that contain definitions and statements. To access objects from an imported module, you need to use dot notation.
* **import** useful\_functions
* useful\_functions.add\_five([1, 2, 3, 4])
* We can add an alias to an imported module to reference it with a different name.
* **import** useful\_functions **as** uf
* uf.add\_five([1, 2, 3, 4])
* **Using a main block**
* To avoid running executable statements in a script when it's imported as a module in another script, include these lines in an if \_\_name\_\_ == "\_\_main\_\_" block. Or alternatively, include them in a function called main() and call this in the if main block.
* Whenever we run a script like this, Python actually sets a special built-in variable called \_\_name\_\_ for any module. When we run a script, Python recognizes this module as the main program, and sets the \_\_name\_\_ variable for this module to the string "\_\_main\_\_". For any modules that are imported in this script, this built-in \_\_name\_\_ variable is just set to the name of that module. Therefore, the condition if \_\_name\_\_ == "\_\_main\_\_"is just checking whether this module is the main program.

The Standard Library

You can discover new modules at the [**Python Module of the Week(opens in a new tab)**](https://pymotw.com/3/) blog.

Techniques for Importing Modules

There are other variants of import statements that are useful in different situations.

1. To import an individual function or class from a module:

**from** module\_name **import** object\_name

1. To import multiple individual objects from a module:

**from** module\_name **import** first\_object, second\_object

1. To rename a module:

**import** module\_name **as** new\_name

1. To import an object from a module and rename it:

**from** module\_name **import** object\_name **as** new\_name

1. To import every object individually from a module (DO NOT DO THIS):

**from** module\_name **import** \*

1. If you really want to use all of the objects from a module, use the standard import module\_name statement instead and access each of the objects with the dot notation.

**import** module\_name

Modules, Packages, and Names

In order to manage the code better, modules in the Python Standard Library are split down into sub-modules that are contained within a package. A **package** is simply a module that contains sub-modules. A sub-module is specified with the usual dot notation.

Modules that are submodules are specified by the package name and then the submodule name separated by a dot. You can import the submodule like this.

**import** package\_name.submodule\_name

Experimenting with an Interpreter

Start your Python interactive interpreter by entering the command python in your terminal. You can type here to interact with Python directly. This is an awesome place to experiment and try bits of Python code at a time. Just enter Python code, and the output will appear on the next line.

>>> type(5.23)

<**class** 'float'>

In the interpreter, the value of the last line in a prompt will be outputted automatically. If you had multiple lines where you’d want to output values, you’d still have to use print.

If you start to define a function you will see a change in the prompt, to signify that this is a continuation line. You'll have to include your own indentation as you define the function.

>>> **def** **cylinder\_volume**(height, radius):

... pi = 3.14159

... **return** height \*pi\* radius \*\* 2

A drawback of the interpreter is that it’s tricky to edit code. If you made a mistake when typing this function, or forgot to indent the body of the function, you can't use the mouse to click your cursor where you want it. You have to navigate with arrow keys to move the cursor forwards and backwards through the line itself for editing. It would be helpful for you to learn useful shortcuts for actions like moving to the beginning or end of the line.

Notice I can reference any objects I defined earlier in the interpreter!

>>> cylinder\_**volume**(10, 3)

282.7431

One useful trick is using the up and down arrow to cycle through your recent commands at the interactive prompt. This can be useful to re-run or adapt code you've already tried.

To quit the Python interactive interpreter, use the command exit() or hit ctrl-D on mac or linux, and ctrl-Z then Enter for windows.

**IPython**

There is actually an awesome alternative to the default Python interactive interpreter, IPython, which comes with many additional features.

* tab completion
* ? for details about an object
* ! to execute system shell commands
* syntax highlighting!

and a lot more you can find [**here(opens in a new tab)**](https://ipython.org/ipython-doc/3/interactive/tutorial.html)!