Prelab 02 | Introduction to Python

The Python Programming Language

Python has syntax that makes teaching Programming concepts easier than it would be in many other languages. It has many of the advantages of other popular programming languages like R (1993) and Java (1995), even though Python (1991) predates them both!

The goal of this prelab, is to introduce some basic programming concepts, data structures, and types and to expose you to some of Python's syntax. A great resource is a free online webbook titled, "Automate the Boring Stuff with Python" by Al Sweigart and can be found at automatetheboringstuff.com. Portions of this introduction were taken from this webbook. Many of the vital topics will be presented here, but as always you are encouraged to learn more and that is a great resource for doing so.

Python Variables & Typing

Variables contain data. Variables can be created and initialized to values, optionally manipulated, and accessed later in code to see what value is retained by it.

Typing is a concept in programming that refers to what kind of information may reside in a variable. Python uses Dynamic Typing. Consider:

```
x = 5
y = '5'
z = 5.0
```

This assigns the value 5 to an integer variable called x, assigns the character '5' to a string variable called y, and assigns the number 5.0 to a floating point number, i.e. real number. This is called instantiation. *In Python, the word "instantiation" is not entirely accurate, but for now its appropriate.* How does Python know to assign 5 to an integer variable rather than a string variable. If quotes are used Python infers its a string, if decimals are used it infers a real number. A variable type can be found using Python's `type` function which takes any variable and will return print its type to the screen.

```
x = 5
type( x )
int
```

Python Operators

In programming languages, every piece of syntax has a name. This includes operators. The equals sign = is also called the assignment operator. The following are a list of Python operators.

Returns	Operator	Description	Example
Number	+ Addition	Adds values on either side of the operator.	**g = -2; b = 4** g + b = 2
	- Subtraction	Subtracts right-hand operand from left-hand operand.	g – b = -6
	* Multiplication	Multiplies values on either side of the operator.	g * b = -8
	/ Division	Divides left-hand operand by right hand operand.	b / g = -2 g / b = -0.5

			-b * g / 5 = 1.6
	**** Exponent	Performs exponential (power) calculation on operators.	g**b = 16
	// Integer Division (Floor Division)	Divides left-hand operand by right-hand operand, where the result is the quotient in which the digits after the decimal point are removed. But if one of the operands is negative, the result is floored, i.e., rounded away from zero (towards negative infinity).	g // b = -1 b // g = -2 20 // 6 = 3
	% Modulo/Modulus	Divides left hand operand by right-hand operand and returns remainder.	g % b = -1 b % g = 0 20 % 6 = 2
None	= Assignment	Assigns the value evaluated on the right-hand operand, to the variable on the left-hand operand.	x = 10 message = 'Hello'
	+=, -=, *=, /=, **=, //=, %=	Respectively adds, subtracts, multiples, divides, exponentiates, modulates, or integer divides the left-hand operand with the right-hand operand AND assigns the calculated value to the left-hand operand which MUST be a variable.	b += 1; (b = 5) b -= 1; (b = 4) b *= 2; (b = 8) b /= 2; (b = 4) b **= 2; (b = 16) b //= 3; (b = 5) b %= 2; (b = 1)
Returns	**Comparison Operator**	**Description**	**Example**
Boolean (Number)	> Greater than	Returns True if the left-hand operand is greater in value than the right-hand operand. Otherwise, False.	**g = -2; b = 4** g > b = False 'B' > 'A' = True
	< Less than	Returns True if the left-hand operand is lesser in value than the right-hand operand. Otherwise, False.	g < b = True 'B' < 'A' = False
	>= Greater than or equal to	Returns True if the left-hand operand is greater than or equal in value to the right-hand operand. Otherwise, False.	g >= -2 = True 'B' >= 'A' = True
	<= Less than or equal to	Returns True if the left-hand operand is less than or equal in value to the right-hand operand. Otherwise, False.	b <= 10 = True 'B' <= 'A' = False
	== Equal to	Returns True if the left-hand operand is equal in value to the right-hand operand. Otherwise, False.	g == b = False 'Hi' == 'Hi' = True
	!= Not equal to	Returns True if the left-hand operand is not equal in value to the right-hand operand. Otherwise, False.	g != b = True 'Hi' != 'Hi' = False
	not	Python keyword that returns True if the right-hand operand is False. Otherwise, False.	not True = False not g == b = True
	and	Python keyword that returns True if the left-hand operand **and** the right-hand operand are both True. Otherwise False.	True and True = True False and g < b = False g < b and 'j' == 'j' = True
	or	Python keyword that returns True if the left-hand operand **or** the right-hand operand are both True.	True or True = True False or g < b = True

		Otherwise `False`.	`g > b or 'j' != 'j' = False`

Most operators listed above are considered "binary operators" because they act on two operands, a left-hand and a right-hand operand. But there are also "unary operators" that act on single operands to produce a result. For instance, the – sign is a binary operator, *subtraction*, in the context that two operands exist on either side of it; but it is a unary operator, *negation*, if only one operand is to the right of it (`-3.5`).

Booleans

Booleans in Python are represented as `True` and `False`, which are case-sensitive[1] values. "Under the hood" Booleans are represented as `1` for `True` and `0` for `False`. An if statement in Python, however will register `0` as `False` and any other number as `True`, though this is not a best practice.

Python Types

The five standard types[2] in Python are the `Number`, `String`, `List`, `Tuple`, and `Dictionary`.

A note on Classes

Number

The four basic subtypes of `Number`[3] are `int` (integer), `long`, `float`, and `complex`. All subtypes are included here for the sake of thoroughness, but the major ones to focus on are `int` and `float`. The difference is that `integers` are discrete values often referred to as "whole numbers" and `floats` are floating-point decimals that may also be represented as a fraction.

Example Numbers

```
x = 10
y = 5.4
# Here, x is an integer and y is a float, however BOTH are Numbers
```

String

The String data type is appropriately named as such because "under the hood" the value is represented as a string of characters.

An array of characters makes up a string and include almost all the keys on a keyboard including the upper and lowercase letters, the numbers, and all whitespace characters like newline (Enter), Space, and Tab.

Python programs can also have text values called `strings`. Always surround your string in single quote (`'`) characters (as in `'Hello'` or `'Goodbye cruel world!'`) so Python knows where the string begins and ends. You can even have a string with no characters in it, `''`, called a blank or *empty* string.

If you ever see the error message `SyntaxError: EOL while scanning string literal`, you probably forgot the final single quote character at the end of the string, such as in this example:

1 https://en.wikipedia.org/wiki/Case_sensitivity
2 https://www.tutorialspoint.com/python/python_variable_types.htm
3 https://docs.python.org/3/library/stdtypes.html#numeric-types-int-float-complex

```
'Hello world!
  SyntaxError: EOL while scanning string literal
```

String Concatenation and Replication

The meaning of an operator may change based on the data types of the values next to it. For example, + is the addition operator when it operates on two integers or floating-point values. However, when + is used on two string values, it joins the strings as the string concatenation operator. Enter the following into the interactive shell:

```
'Alice' + 'Bob'
  'AliceBob'
```

The expression evaluates down to a single, new string value that combines the text of the two strings. However, if you try to use the + operator on a string and an integer value, Python will not know how to handle this, and it will display an error message.

```
'Alice' + 42
  Traceback (most recent call last):
    File "<pyshell#26>", line 1, in <module>
      'Alice' + 42
  TypeError: Can't convert 'int' object to str implicitly
```

The error message Can't convert `'int'` object to `str` implicitly means that Python thought you were trying to concatenate an integer to the string `'Alice'`. Your code will have to explicitly convert the `integer` to a `string`, because Python cannot do this automatically. Converting data types is done with the `str()`, `int()`, and `float()` functions (below).

More along these lines, there is an important distinction between numbers and number characters/numerical digits.

```
x = '5'
5 + x
  Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
  TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

Type Conversion

It is important to know how to convert between these types as necessary. The most prevalent type conversions that need to be done, are that of String Number, or Integer Float.

String ↔ Number

Strings are the data type that come in from the command line as arguments and from any type of text file. This might seem inconvenient at first, but it enables a touch of predictability when dealing with external data, which makes a program much more powerful.

Type the following into an interactive shell and see how the `int()` functions works to parse an integer type out of a string:

```
x = '42';
x
  '42'
int( x )
  42
```

This converts the string of digits into an integer which is recognized by Python as an item that can now be mathematically manipulated. We can also apply the `float()` function to parse a

floating-point number and also the `str()` function to create a string representation of a number:

```
x = '1.125'
x + 5
```

```
  Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
  TypeError: Can't convert 'int' object to str implicitly
```

```
float( x ) + 5
```

```
  6.125
```

```
int( x ) + 5
```

```
  Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
  ValueError: invalid literal for int() with base 10: '1.125'
```

```
y = 1234
'I love the number ' + y
```

```
  Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
  TypeError: Can't convert 'int' object to str implicitly
```

```
print( 'I love the number ' + str( y ) )
```

```
  I love the number 1234
```

```
print( 'I love the number', y )
```

```
  I love the number 1234
```

You may have noticed at the last statement the `print` function will attempt string conversion automatically.

Integer ↔ Float

This one is not as common, but can lead to headaches in some cases. The `int()` function will always take the floor of the `float`. If you prefer to round though, you should use the built-in `round()` function, or you can use the `ceil()` function which rounds up to the nearest integer.

```
x = 5.9
int( x )
  5
round( x )
  6
round( 5.3 )
  5
x = 2
x ** 4
  16
x ** 4.0
  16.0
x = float( x ) ** 4
x
  16.0
int( x )
  16
```

When a mathematical operation is done using integers and the result is a whole number also, the number stays an integer. The moment a float is used in any part of the expression, the result will also be a float, even if it is a whole number. Of course, you can always attempt to recast that number back to an integer if needed. As we'll see in the section describing lists, this may be important if you wish to access an index of a list with the variable holding a whole number.

List

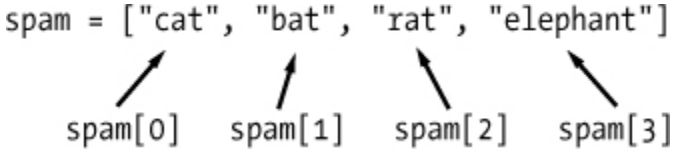
A *list* contains multiple values in an ordered sequence. The term *list value* refers to the list itself (which is a value that can be stored in a variable or passed to a function like any other value), not the values inside the list value. A list looks like this: `['cat', 'bat', 'rat', 'elephant']`. Just as string values are typed with quote characters to mark where the string begins and ends, a list begins with an opening square bracket and ends with a closing square bracket, `[]`. Values inside the list are also called items. Items are separated with commas (that is, they are comma-delimited). For example:

```
[1, 2, 3]
  [1, 2, 3]
['cat', 'bat', 'rat', 'elephant']
  ['cat', 'bat', 'rat', 'elephant']
['hello', 3.1415, True, None, 42]
  ['hello', 3.1415, True, None, 42]
spam = ['cat', 'bat', 'rat', 'elephant']
spam
  ['cat', 'bat', 'rat', 'elephant']
```

The `spam` variable is still assigned only one value: the list value. But the list value itself contains other values. The value `[]` is an empty list that contains no values, similar to `''`, the empty string.

Getting Individual Values in a List with Indexes

Say you have the list `['cat', 'bat', 'rat', 'elephant']` stored in a variable named `spam`. The Python code `spam[0]` would evaluate to `'cat'`, and `spam[1]` would evaluate to `'bat'`, and so on. The integer inside the square brackets that follows the list is called an index. The first value in the list is at index `0`, the second value is at index `1`, the third value is at index `2`, and so on. The following figure shows a list value assigned to `spam`, along with what the index expressions would evaluate to.



For example:

```
spam = ['cat', 'bat', 'rat', 'elephant']
spam[0]
  'cat'
spam[1]
  'bat'
['cat', 'bat', 'rat', 'elephant'][3]
  'elephant'
'Hello' + spam[0]
  'Hello cat'
```

Notice that the expression `'Hello ' + spam[0] (1)` evaluates to `'Hello ' + 'cat'` because `spam[0]` evaluates to the string `'cat'`.

Python will give you an `IndexError` error message if you use an index that exceeds the number of values in your list value.

```
spam = ['cat', 'bat', 'rat', 'elephant']
spam[10000]
```

```
  Traceback (most recent call last):
    File "<pyshell#9>", line 1, in <module>
      spam[10000]
  IndexError: list index out of range
```

Indexes can be only integer values, not floats. The following example will cause a `TypeError` error:

Lists can also contain other list values. The values in these lists of lists can be accessed using multiple indexes, like so:

```
spam = [['cat', 'bat'], [10, 20, 30, 40, 50]]
spam[0]
```

```
  ['cat', 'bat']
```

```
spam[0][1]
```

```
  'bat'
```

```
spam[1][4]
```

```
  50
```

The first index dictates which list value to use, and the second indicates the value within the list value. For example, `spam[0][1]` prints `'bat'`, the second value in the first list. If you only use one index, the program will print the full list value at that index.

Negative Indexes

While indexes start at `0` and go up, you can also use negative integers for the index. The integer value `-1` refers to the last index in a list, the value `-2` refers to the second-to-last index in a list, and so on. Enter the following into the interactive shell:

```
spam = ['cat', 'bat', 'rat', 'elephant']
spam[-1]
```

```
  'elephant'
```

```
spam[-3]
```

```
  'bat'
```

```
'The ' + spam[-1] + ' is afraid of the ' + spam[-3] + '.'
```

```
  'The elephant is afraid of the bat.'
```

Getting Sublists with Slices

Just as an index can get a single value from a list, a *slice* can get several values from a list, in the form of a new list. A slice is typed between square brackets, like an index, but it has two integers separated by a colon. Notice the difference between indexes and slices.

- `spam[2]` is a list with an index (one integer).
- `spam[1:4]` is a list with a slice (two integers).

In a slice, the first integer is the index where the slice starts. The second integer is the index where the slice ends. A slice goes up to, but will not include, the value at the second index. A slice evaluates to a new list value.

```
spam = ['cat', 'bat', 'rat', 'elephant']
spam[0:4]
  ['cat', 'bat', 'rat', 'elephant']
spam[1:3]
  ['bat', 'rat']
spam[0:-1]
  ['cat', 'bat', 'rat']
```

As a shortcut, you can leave out one or both indexes on either side of the colon in the slice. Leaving out the first index is the same as using `0`, or the beginning of the list. Leaving out the second index is the same as using the length of the list, which will slice to the end of the list.

```
spam = ['cat', 'bat', 'rat', 'elephant']
spam[:2]
  ['cat', 'bat']
spam[1:]
  ['bat', 'rat', 'elephant']
spam[:]
  ['cat', 'bat', 'rat', 'elephant']
```

Getting a List's Length with `len()`

The `len()` function will return the number of values that are in a list value passed to it, just like it can count the number of characters in a string value. Enter the following into the interactive shell:

```
spam = ['cat', 'dog', 'moose']
len(spam)
  3
```

Changing Values in a List with Indexes

Normally a variable name goes on the left side of an assignment statement, like `spam = 42`. However, you can also use an index of a list to change the value at that index. For example, `spam[1] = 'aardvark'` means "Assign the value at index `1` in the list spam to the string `'aardvark'`."

```
spam = ['cat', 'bat', 'rat', 'elephant']
spam[1] = 'aardvark'
spam
  ['cat', 'aardvark', 'rat', 'elephant']
spam[2] = spam[1]
spam
  ['cat', 'aardvark', 'aardvark', 'elephant']
spam[-1] = 12345
spam
  ['cat', 'aardvark', 'aardvark', 12345]
```

List Concatenation and List Replication

The + operator can combine two lists to create a new list value in the same way it combines two strings into a new string value. The * operator can also be used with a list and an integer value to replicate the list.

```
[1, 2, 3] + ['A', 'B', 'C']
  [1, 2, 3, 'A', 'B', 'C']
['X', 'Y', 'Z'] * 3
  ['X', 'Y', 'Z', 'X', 'Y', 'Z', 'X', 'Y', 'Z']
spam = [1, 2, 3]
spam = spam + ['A', 'B', 'C']
spam
  [1, 2, 3, 'A', 'B', 'C']
```

Removing Values from Lists with del Statements

The `del` statement will delete values at an index in a list. All of the values in the list after the deleted value will be moved up one index. For example, enter the following into the interactive shell:

```
spam = ['cat', 'bat', 'rat', 'elephant']
del spam[2]
spam
  ['cat', 'bat', 'elephant']
del spam[2]
spam
  ['cat', 'bat']
```

The `del` statement can also be used on a simple variable to delete it, as if it were an "unassignment" statement. If you try to use the variable after deleting it, you will get a `NameError` error because the variable no longer exists.

Dictionary

A dictionary, also known as a map in other languages, holds a collection of items similarly to a list, but of key-value pairs. The keys in a dictionary must be unique. A list is actually sort of like a dictionary, but whose keys are all integers starting at the index `0`.

Indexes for dictionaries can use many different data types, not just integers. Indexes for dictionaries are called keys, and a key with its associated value is called a key-value pair.

In code, a dictionary is typed with braces, `{}`.

```
myCat = { 'size': 'fat', 'color': 'gray', 'disposition': 'loud' }
```

This assigns a dictionary to the `myCat` variable. This dictionary's keys are `'size'`, `'color'`, and `'disposition'`. The values for these keys are `'fat'`, `'gray'`, and `'loud'`, respectively.

You can access these values through their keys:

```
myCat['size']
  'fat'
'My cat has ' + myCat['color'] + ' fur.'
  'My cat has gray fur.'
```

Dictionaries can still use integer values as keys, just like lists use integers for indexes, but they do not have to start at `0` and can be any number.

```
spam = { 12345: 'Luggage Combination', 42: 'The Answer' }
```

Dictionaries vs. Lists

Unlike lists, items in dictionaries are unordered. The first item in a list named spam would be `spam[0]`. But there is no "first" item in a dictionary. While the order of items matters for determining whether two lists are the same, it does not matter in what order the key-value pairs are typed in a dictionary.

```
spam = ['cats', 'dogs', 'moose']
bacon = ['dogs', 'moose', 'cats']
spam == bacon
```

```
  False
```

```
eggs = {'name': 'Zophie', 'species': 'cat', 'age': '8'}
ham = {'species': 'cat', 'age': '8', 'name': 'Zophie'}
eggs == ham
```

```
  True
```

Because dictionaries are not ordered, they can't be sliced like lists.

Trying to access a key that does not exist in a dictionary will result in a `KeyError` error message, much like a list's "out-of-range" `IndexError` error message. Enter the following into the interactive shell, and notice the error message that shows up because there is no `'color'` key:

```
spam = {'name': 'Zophie', 'age': 7}
spam['color']
```

```
  Traceback (most recent call last):
    File "<pyshell#1>", line 1, in <module>
      spam['color']
  KeyError: 'color'
```

Though dictionaries are not ordered, the fact that you can have arbitrary values for the keys allows you to organize your data in powerful ways. Say you wanted your program to store data about your friends' birthdays. You can use a dictionary with the names as keys and the birthdays as values. If you want to retrieve a birthday, you only need the name. Such information could be stored in list form: a list of names and a list of birthdays that are paired by their index. But, finding a birthday given a name is not as easy, you'd have to search through the list to find the name of interest in order to retrieve the birthday.

The **`keys()`**, **`values()`**, and **`items()`** Methods

There are three dictionary methods that will return list-like values of the dictionary's keys, values, or both keys and values: `keys()`, `values()`, and `items()`. The values returned by these methods are not true lists: They cannot be modified and do not have an `append()` method. But these data types (`dict_keys`, `dict_values`, and `dict_items`, respectively) *can* be used in `for` loops.

```
spam = { 'color': 'red', 'age': 42 }
for v in spam.values():
    print( v )
```

```
red
42
```

Here, a `for` loop iterates over each of the values in the `spam` dictionary. A `for` loop includes the the `for` keyword, a variable name, the `in` keyword and a call to the `range()` method. It is followed by an indented (tab) block of code that is executed for each iteration of the loop. A `for` loop can also iterate over the keys or both keys and values:

```
for k in spam.keys():
    print( k )

color
age

for i in spam.items():
    print( I )

('color', 'red')
('age', 42)
```

Using the `keys()`, `values()`, and `items()` methods, a for loop can iterate over the keys, values, or key-value pairs in a dictionary, respectively. Notice that the values in the `dict_items` value returned by the `items()` method are tuples of the key and value. *Tuples are not discussed here in detail; they are similar to lists, but immutable.*

If you want a true list from one of these methods, pass its list-like return value to the `list()` function.

```
spam = { 'color': 'red', 'age': 42 }
spam.keys()
  dict_keys(['color', 'age'])

list( spam.keys() )
  ['color', 'age']
```

The `list( spam.keys() )` line takes the `dict_keys` value returned from `keys()` and passes it to `list()`, which then returns a list value of `['color', 'age']`. You can also use the multiple assignment trick in a `for` loop to assign the key and value to separate variables.

```
spam = { 'color': 'red', 'age': 42 }
for k, v in spam.items():
    print( 'Key: ' + k + ' Value: ' + str( v ) )


Key: age Value: 42
Key: color Value: red
```

The **`get()`** Method

It's tedious to check whether a key exists in a dictionary before accessing that key's value. Fortunately, dictionaries have a `get()` method that takes two arguments: the key of the value to retrieve and a fallback value to return if that key does not exist.

Enter the following into the interactive shell:

```
picnicItems = { 'apples': 5, 'cups': 2 }
'I am bringing ' + str( picnicItems.get( 'cups', 0 ) ) + ' cups.'
  'I am bringing 2 cups.'

'I am bringing ' + str(picnicItems.get('eggs', 0)) + ' eggs.'
  'I am bringing 0 eggs.'
```

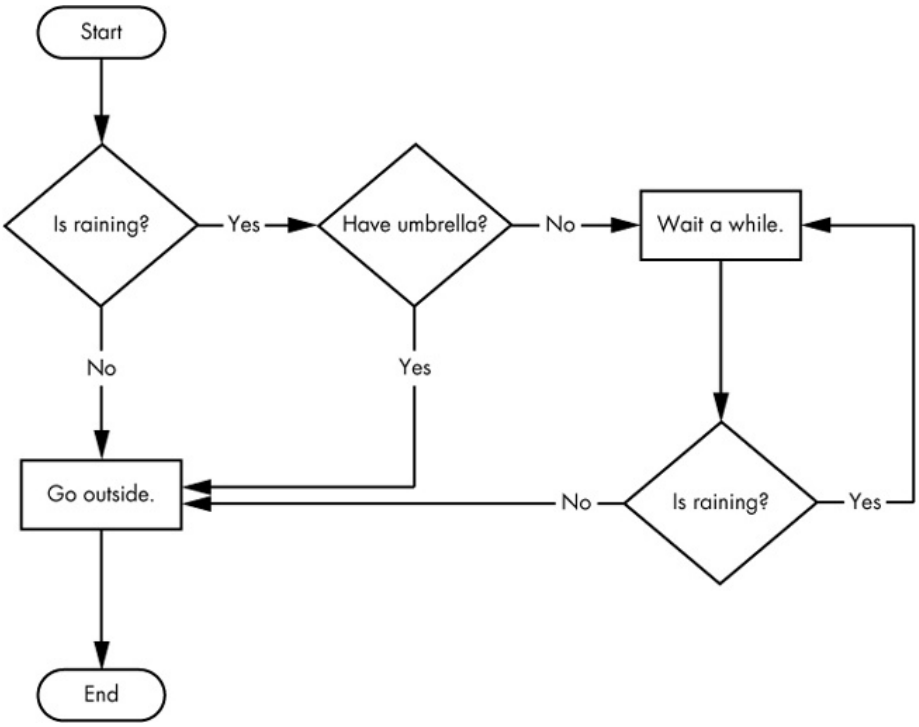
Because there is no `'eggs'` key in the `picnicItems` dictionary, the default value `0` is returned by the `get()` method. Without using `get()`, the code would have caused an error message, such as in the following example:

```
picnicItems = { 'apples': 5, 'cups': 2 }
'I am bringing ' + str( picnicItems['eggs'] ) + ' eggs.'
  Traceback (most recent call last):
    File "<pyshell#34>", line 1, in <module>
      'I am bringing ' + str(picnicItems['eggs']) + ' eggs.'
  KeyError: 'eggs'
```

CONTROL FLOW IN PYTHON

`if` statements, `for` loops, and `while` loops form the basis for flow control in Python as in any other language. A program is just a series of instructions. But the real strength of programming isn't just running (or *executing*) one instruction after another like a weekend errand list. Based on how the expressions evaluate, the program can decide to skip instructions, repeat them, or choose one of several instructions to run. In fact, you almost never want your programs to start from the first line of code and simply execute every line, straight to the end. *Flow control statements can decide which Python instructions to execute under which conditions.*

These flow control statements directly correspond to the symbols in a flowchart. The following figure shows a flowchart for what to do if it is raining. Follow the path made by the arrows from Start to End.



In a flowchart, there is usually more than one way to go from the start to the end. The same is true for lines of code in a computer program. Flowcharts represent these branching points with diamonds, while the other steps are represented with rectangles. The starting and ending steps are represented with rounded rectangles.

These flow control statements are only possible to understand once you have learned how to represent those **yes** and **no** options, and you need to understand how to write those branching points as Python code. Booleans are the key to this.

If Statements

The most common type of flow control statement is the `if` statement. An `if` statement's clause (that is, the block following the `if` statement) will execute if the statement's condition is `True`. The clause is skipped if the condition is `False`.

In plain English, an `if` statement could be read as, "If this condition is true, execute the code in the clause." In Python, an `if` statement consists of the following:

- The `if` keyword
- A condition (that is, an expression that evaluates to `True` or `False`)
- A colon – `:`
- Starting on the next line, an indented block of code (called the `if` clause)

For example, let's say you have some code that checks to see whether someone's name is Alice. (Pretend `name` was assigned some value earlier.)

```
if name == 'Alice':
    print( 'Hi, Alice.' )
```

All flow control statements end with a colon and are followed by a new block of code (the clause). This `if` statement's clause is the block with `print( 'Hi, Alice.' )`.

`else` Statements

An `if` clause can optionally be followed by an `else` statement. The `else` clause is executed only when the `if` statement's condition is `False`. In plain English, an `else` statement could be read as, "If this condition is true, execute this code. Or else, execute that code." An `else` statement doesn't have a condition, and in code, an `else` statement always consists of the following:

- The `else` keyword
- A colon – `:`
- Starting on the next line, an indented block of code (called the `else` clause)

Returning to the Alice example, let's look at some code that uses an `else` statement to offer a different greeting if the person's name isn't Alice.

```
name = 'Bob'
if name == 'Alice':
    print( 'Hi, Alice.' )
else:
    print( 'Hello, stranger.' )


Hello, stranger.
```

`elif` Statements

While only one of the `if` or `else` clauses will execute, you may have a case where you want one of *many* possible clauses to execute. The `elif` statement is an "else if" statement that always follows an `if` or another `elif` statement. It provides another condition that is checked only if all of the previous conditions were `False`. In code, an `elif` statement always consists of the following:

- The `elif` keyword
- A condition (that is, an expression that evaluates to `True` or `False`)

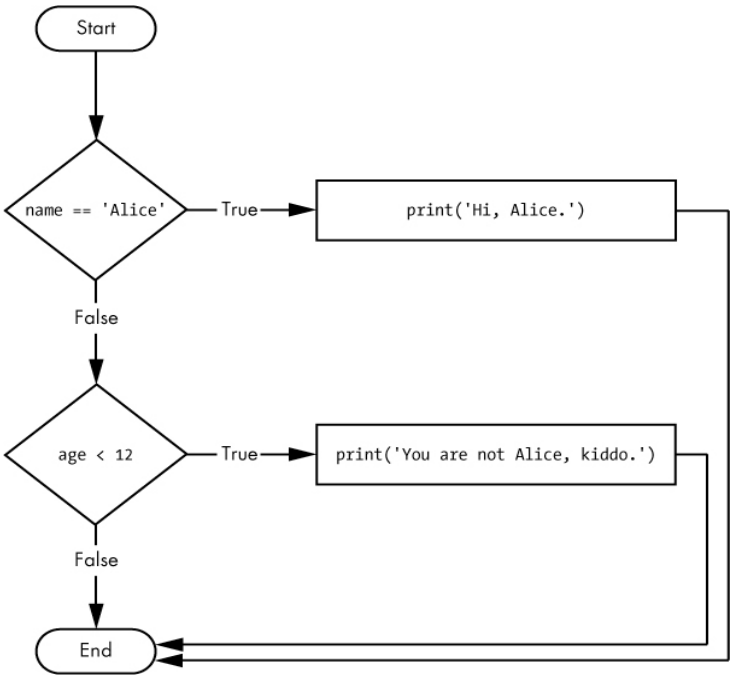
- A colon – :
- Starting on the next line, an indented block of code (called the `elif` clause)

Let's add an `elif` to the name checker to see this statement in action.

```
name = 'Bob'
age = 5
if name == 'Alice':
    print('Hi, Alice.')
elif age < 12:
    print('You are not Alice, kiddo.')
```

```
You are not Alice, kiddo.
```

This time, you check the person's age, and the program will tell them something different if they're younger than 12.



The `elif` clause executes if `age < 12` is `True` and `name == 'Alice'` is `False`. However, if both of the conditions are `False`, then both of the clauses are skipped. It is *not* guaranteed that at least one of the clauses will be executed. When there is a chain of `elif` statements, only one or none of the clauses will be executed. Once one of the statements' conditions is found to be `True`, the rest of the `elif` clauses are automatically skipped.

While loops

A while loop keeps reiterating a code block while a Boolean/logical state is `true`, and terminates the code block if `false`. Thus, you can make a block of code execute over and over again with a `while` statement. In code, a `while` statement always consists of the following:

- The `while` keyword
- A condition (that is, an expression that evaluates to `True` or `False`)
- A colon – :
- Starting on the next line, an indented block of code (called the `while` clause)

You can see that a `while` statement looks similar to an `if` statement. The difference is in how they behave. At the end of an `if` clause, the program execution continues after the `if`

statement. But at the end of a `while` clause, the program execution jumps back to the start of the while statement. The `while` clause is often called the *while loop* or just the *loop*.

Let's look at an `if` statement and a while loop that use the same condition and take the same actions based on that condition.

Here is the code with an `if` statement:

```
spam = 0
if spam < 5:
    print( 'Hello, world.' )
    spam = spam + 1


Hello, world.
```

Here is the code with a `while` statement:

```
spam = 0
while spam < 5:
    print('Hello, world.')
    spam = spam + 1


Hello, world.
Hello, world.
Hello, world.
Hello, world.
Hello, world.
```

These statements are similar—both `if` and `while` check the value of `spam`, and if it's less than five, they print a message. But when you run these two code snippets, something very different happens for each one. For the `if` statement, the output is simply `"Hello, world."`. But for the while statement, it's `"Hello, world."` repeated five times!

break Statements

There is a shortcut to getting the program execution to break out of a `while` loop's clause early. If the execution reaches a `break` statement, it immediately exits the clause of the while loop. In code, a `break` statement simply contains the break keyword.

Here's a program that uses a `break` statement to escape the loop.

```
while name != 'your name':
  print( 'Please type your name.' )
  name = input()
  if name == 'your name':
       break
print( 'Thank you!' )
```

The first line creates an *infinite loop*; it is a `while` loop whose condition is always `True`. (The expression `True`, after all, always evaluates down to the value `True`.) The program execution will always enter the loop and will exit it only when a `break` statement is executed. (An infinite loop that *never* exits is a common programming bug.)

This program asks the user to type **`your name`**. However, while the execution is still inside the `while` loop, an if statement gets executed to check whether `name` is equal to `'your name'`. If this condition is `True`, the `break` statement is run, and the execution moves out of the loop to `print( 'Thank you!' )`. Otherwise, the `if` statement's clause with the `break` statement is skipped, which puts the execution at the end of the `while` loop. At this point, the program execution jumps back to the start of the `while` statement to recheck the condition. Since this condition is merely the `True` Boolean value, the execution enters the loop to ask the user to type `your name` again.

If you ever run a program that has a bug causing it to get stuck in an infinite loop, press `Ctrl+C`. This will send a `KeyboardInterrupt` error to your program and cause it to stop immediately.

`continue` Statements

Like `break` statements, `continue` statements are used inside loops. When the program execution reaches a `continue` statement, the program execution immediately jumps back to the start of the loop and reevaluates the loop's condition. (This is also what happens when the execution reaches the end of the loop.)

Let's use `continue` to write a program that asks for a name and password.

```
while True:
  print('Who are you?')
  name = input()
  if name != 'Joe':
       continue
  print('Hello, Joe. What is the password? (It is a fish.)')
  password = input()
  if password == 'swordfish':
       print('Access granted.')
       break
```

If the user enters any name besides `Joe`, the `continue` statement causes the program execution to jump back to the start of the loop. When it reevaluates the condition, the execution will always enter the loop, since the condition is simply the value `True`. Once they make it past that `if` statement, the user is asked for a password. If the password entered is `swordfish`, then the `break` statement is run, and the execution jumps out of the `while` loop to print `Access granted`. Otherwise, the execution continues to the end of the `while` loop, where it then jumps back to the start of the loop.

For Loops and the `range()` Function

The `while` loop keeps looping while its condition is `True` (which is the reason for its name), but what if you want to execute a block of code only a certain number of times? You can do this with a `for` loop statement and the `range()` function. In code, a `for` statement looks something like `for i in range( 5 ):` and always includes the following:

- The `for` keyword
- A variable name
- The `in` keyword
- A call to the `range()` method with up to three integers passed to it
- A colon – `:`
- Starting on the next line, an indented block of code (called the `for` clause)

Let's create a `for` loop.

```
print( 'My name is' )
for i in range( 5 ):
    print( 'Jimmy Five Times (' + str( i ) + ')' )


My name is
Jimmy Five Times (0)
Jimmy Five Times (1)
Jimmy Five Times (2)
Jimmy Five Times (3)
Jimmy Five Times (4)
```

The code in the `for` loop's clause is run five times. The first time it is run, the variable `i` is set to `0`. The `print()` call in the clause will print `Jimmy Five Times (0)`. After Python finishes an iteration through all the code inside the clause of the `for` loop, the execution goes back to the top of the loop, and the `for` statement increments `i` by one. This is why `range(5)` results in

five iterations through the clause, with `i` being set to `0`, then `1`, then `2`, then `3`, and then `4`. The variable `i` will go up to, but will not include, the integer passed to `range()`.

Note

> You can use `break` and `continue` statements inside `for` loops as well. The `continue` statement will continue to the next value of the `for` loop's counter, as if the program execution had reached the end of the loop and returned to the start. In fact, you can use `continue` and `break` statements only inside `while` and `for` loops. If you try to use these statements elsewhere, Python will give you an error.

As another `for` loop example, lets add up all the numbers from `0` to `100`.

```
total = 0
for num in range( 101 ):
  total = total + num
print( total )
```

```
5050
```

The result should be `5,050`. When the program first starts, the total variable is set to `0`. The `for` loop then executes `total = total + num` 101 times. Why 101? Because by default `range()` starts at 0 and we want to include 100 and so need 101 iterations. Alternatively, we could use `range( 100 )` and `total = total + num + 1`, or instruct range to be `range( 1, 101)`.

An Equivalent while Loop

You can actually use a `while` loop to do the same thing as a `for` loop; `for` loops are just more concise. Let's use a `while` loop equivalent of a `for` loop.

```
print( 'My name is' )
i = 0
while i < 5:
    print( 'Jimmy Five Times (' + str( i ) + ')' )
    i = i + 1
```

```
My name is
Jimmy Five Times (0)
Jimmy Five Times (1)
Jimmy Five Times (2)
Jimmy Five Times (3)
Jimmy Five Times (4)
```

Using `for` Loops with Lists

Technically, a for loop repeats the code block once for each value in a list or list-like value. For example, if you ran this code:

```
for i in range( 4 ):
    print( i )
```

```
0
1
2
3
```

This is the output because the return value from `range( 4 )` is a list-like value that Python considers similar to `[0, 1, 2, 3]`. The following program has the same output as the previous one:

```
for i in [0, 1, 2, 3]:
    print( i )
```

What the previous `for` loop actually does is loop through its clause with the variable `i` set to a successive value in the `[0, 1, 2, 3]` list in each iteration.

Note: *The term* list-like *to refer to data types that are technically named* sequences*. You don't need to know the technical definitions of this term, though.*

A common Python technique is to use `range( len( someList ) )` with a `for` loop to iterate over the indexes of a list. For example:

```
supplies = ['pens', 'staplers', 'flame-throwers', 'binders']
for i in range(len(supplies)):
    print('Index ' + str(i) + ' in supplies is: ' + supplies[i])
```

```
Index 0 in supplies is: pens
Index 1 in supplies is: staplers
Index 2 in supplies is: flame-throwers
Index 3 in supplies is: binders
```

Using `range( len( supplies ) )` in the previously shown `for` loop is handy because the code in the loop can access the index (as the variable `i`) and the value at that index (as `supplies[i]`). Best of all, `range( len( supplies ) )` will iterate through all the indexes of `supplies`, no matter how many items it contains.

The **`in`** and **`not in`** Operators

You can determine whether a value is or isn't in a list with the `in` and `not in` operators. Like other operators, `in` and `not in` are used in expressions and connect two values: a value to look for in a list and the list where it may be found. These expressions will evaluate to a Boolean value.

```
'howdy' in ['hello', 'hi', 'howdy', 'heyas']
  True
spam = ['hello', 'hi', 'howdy', 'heyas']
'cat' in spam
  False
'howdy' not in spam
  False
'cat' not in spam
  True
```

For example, the following program lets the user type in a pet name and then checks to see whether the name is in a list of pets.

```
myPets = ['Zophie', 'Pooka', 'Fat-tail']
print( 'Enter a pet name:' )
name = input()
if name not in myPets:
    print( 'I do not have a pet named ' + name )
else:
    print( name + ' is my pet.' )
```

The output may look something like this:

```
Enter a pet name:
Footfoot
I do not have a pet named Footfoot
```

Python File IO

IO (Input/Output) refers to the ability to read (input) and write (output) from and to a file. Using the open function, you create what is called a file "handle" or "stream" that allows the program to access or alter the information in a given file (should it have permissions to do so). When dealing with file streams, it is important to remember to `close` them after they have been opened and utilized. Not doing so can result in adverse effects in your program and on the contents of the file[4].

Input

Read from command line with the `input()` Function

The `input()` function waits for the user to type some text on the keyboard and press `ENTER`.

```
myName = input()
```

This function call evaluates to a string equal to the user's text, and the previous line of code assigns the `myName` variable to this string value.

You can think of the `input()` function call as an expression that evaluates to whatever string the user typed in. If the user entered `'Al'`, then the expression would evaluate to `myName = 'Al'`.

Read from File

Try using the `open` command in a Jupyter Notebook on the course server to open a file. Since we have no intention of changing this file, make sure to use the `'r'` or read mode as the second argument in the `open` function to imply we only wish to **r**ead from this file.

```
filename = '/usr/ref/dmel/genome/dmel-all-chromosome-r6.15.fasta'
dmel_genome = open( filename, 'r' )
 . . .
dmel_genome.close()
```

This opens and closes a reference genome for *Drosophila melanogaster*. While it is open the lines of the file are completely at the program's disposal to do any calculation desired. Once the file is closed however, it can no longer be accessed unless it is opened again.

Output

Write to command line with the `print()` Function

There are times where you will simply want to inform the user of something and would prefer to print to the command line.

The following call to `print()` actually contains the expression `'It is good to meet you, ' + myName` between the parentheses.

```
print( 'It is good to meet you, ' + myName )
  It is good to meet you Al
```

Remember that expressions can always evaluate to a single value. If `'Al'` is the value stored in `myName` on the previous line, then this expression evaluates to `'It is good to meet you, Al'`. This single string value is then passed to `print()`, which prints it on the screen.

4https://stackoverflow.com/a/29536383/2787584

Write to file

In the case a program should create files with data written to them or even append to pre-existing files, then the file handle object created via the open function can be used to do so.

Contrary to last time we used `open`, we will use the `'w'` mode here to imply we only wish to **w**rite to the file.

```
filename = 'testFile.txt'
testFile = open( filename, 'w' )
testFile.write( 'Hey there!\n' )
testFile.close()
```

After closing this file you can run `cat` in the Bash shell to see the contents of the file.

Using `'w'` will cause the `write` method to overwrite all the contents that were in that file if it existed already. If this is not the desire, use the `'a'` for append to add to the file instead.

Python Scope

Scope is not something thought about over actively when coding, but if forgotten can create headaches! Variables only tend to exist in the "context" or scope they were created in.

In fact, there is an operator for telling a program to change scope to become more, or less specific. In Python it is leading whitespace on newlines after a `:` colon.

A great example of where scope is important are the `for` and `while` loops we discussed earlier, but also in functions.

Functions

You're already familiar with the `print()`, `input()`, and `len()` functions from the previous chapters. Python provides several **builtin** functions like these, but you can also write your own functions. A *function* is like a mini-program within a program.

To better understand how functions work, let's create one.

```
def hello():
  print( 'Howdy!' )
  print( 'Howdy!!!' )
  print( 'Hello there.' )

hello()
hello()
hello()
```

The first line is a `def` statement, which defines a function named `hello()`. The code in the block that follows the `def` statement is the body of the function. This code is executed when the function is called, **not** when the function is first defined.

The `hello()` lines after the function are function calls. In code, a function call is just the function's name followed by parentheses, possibly with some number of arguments in between the parentheses. When the program execution reaches these calls, it will jump to the top line in the function and begin executing the code there. When it reaches the end of the function, the execution returns to the line that called the function and continues moving through the code as before.

Since this program calls `hello()` three times, the code in the `hello()` function is executed three times. When you run this program, the output looks like this:

```
Howdy!
Howdy!!!
Hello there.
Howdy!
Howdy!!!
Hello there.
Howdy!
Howdy!!!
Hello there.
```

A major purpose of functions is to group code that gets executed multiple times.

Without a function defined, you would have to copy and paste this code each time, and the program would look like this:

```
print( 'Howdy!' )
print( 'Howdy!!!' )
print( 'Hello there.' )
print( 'Howdy!' )
print( 'Howdy!!!' )
print( 'Hello there.' )
print( 'Howdy!' )
print( 'Howdy!!!' )
print( 'Hello there.' )
```

In general, you always want to **avoid duplicating code**, because if you ever decide to update the code—if, for example, you find a bug you need to fix—you'll have to remember to change the code everywhere you copied it.

Back to Scope

Parameters and variables that are assigned in a called function are said to exist in that function's *local scope*. Variables that are assigned outside all functions are said to exist in the *global scope*. A variable that exists in a local scope is called a *local variable*, while a variable that exists in the global scope is called a *global variable*. A variable must be one or the other; it cannot be both local and global.

Think of a *scope* as a container for variables. When a scope is destroyed, all the values stored in the scope's variables are forgotten. There is only one global scope, and it is created when your program begins. When your program terminates, the global scope is destroyed, and all its variables are forgotten. Otherwise, the next time you ran your program, the variables would remember their values from the last time you ran it.

A local scope is created whenever a function is called. Any variables assigned in this function exist within the local scope. When the function returns, the local scope is destroyed, and these variables are forgotten. The next time you call this function, the local variables will not remember the values stored in them from the last time the function was called.

Scopes matter for several reasons:

- Code in the global scope cannot use any local variables.
- However, a local scope can access global variables.
- Code in a function's local scope cannot use variables in any other local scope.
- You can use the same name for different variables if they are in different scopes. That is, there can be a local variable named `spam` and a global variable also named `spam`.

The reason Python has different scopes instead of just making everything a global variable is so that when variables are modified by the code in a particular call to a function, the function interacts with the rest of the program only through its parameters and the return value. This narrows down the list code lines that may be causing a bug. If your program contained nothing but global variables and had a bug because of a variable being set to a bad value, then it would be hard to track down where this bad value was set. It could have been set from

anywhere in the program—and your program could be hundreds or thousands of lines long! But if the bug is because of a local variable with a bad value, you know that only the code in that one function could have set it incorrectly.

While using global variables in small programs is fine, it is a bad habit to rely on global variables as your programs get larger and larger.

Local Variables Cannot Be Used in the Global Scope

Consider this program, which will cause an error when you run it:

```
def spam():
    eggs = 31337
spam()
print( eggs )
```

If you run this program, the output will look like this:

```
Traceback (most recent call last):
  File "C:/test3784.py", line 4, in <module>
    print(eggs)
NameError: name 'eggs' is not defined
```

The error happens because the `eggs` variable exists only in the local scope created when `spam()` is called. Once the program execution returns from `spam`, that local scope is destroyed, and there is no longer a variable named `eggs`. So when your program tries to run `print( eggs )`, Python gives you an error saying that `eggs` is not defined. Therefore, only global variables can be used in the global scope.

Local Scopes Cannot Use Variables in Other Local Scopes

A new local scope is created whenever a function is called, including when a function is called from another function. Consider this program:

```
def spam():
  eggs = 99
  bacon()
  print( eggs )

def bacon():
  ham = 101
  eggs = 0
spam()


99
```

When the program starts, the `spam()` function is called, and a local scope is created. The local variable `eggs` is set to `99`. Then the `bacon()` function is called, and a second local scope is created. Multiple local scopes can exist at the same time. In this new local scope, the local variable `ham` is set to `101`, and a local variable `eggs`—which is different from the one in `spam()`'s local scope—is also created and set to `0`.

When `bacon()` returns, the local scope for that call is destroyed. The program execution continues in the `spam()` function to print the value of `eggs`, and since the local scope for the call to `spam()` still exists here, the `eggs` variable is set to `99`. This is what the program prints.

The upshot is that local variables in one function are completely separate from the local variables in another function.

Global Variables Can Be Read from a Local Scope

Consider the following program:

```
def spam():
    print( eggs )
eggs = 42
```

```
spam()
42
```

Since there is no parameter named `eggs` or any code that assigns `eggs` a value in the `spam()` function, when `eggs` is used in `spam()`, Python considers it a reference to the global variable `eggs`. This is why `42` is printed when the previous program is run.