Table of Contents

Introduction

This document describes some of the different types of simple statements in Python that will be covered in this course. A list of videos along with their timestamps will be included for more examples and explanations.

For a full list of simple statements that exist in Python and their explanations, consult the simple statements section in the official Python Documentation.

Videos

- What is a Python program (0:12 - 1:44)
    - Covers what is defined as a (Python) program, some types of statements in Python
- The first program: Hello World (0:00 - 7:21)
    - Covers how to create simple program to print various messages
- Input and output (0:00 - 9:33)
    - Covers the built-in input function, how to obtain an input, process it, and display it in the output.
- Creating objects with literals (0:53 - 3:22)
    - Covers how to create objects of type str, float and int using literals
- Expressions (3:28 - 4:50)
    - Covers some of the kinds of expressions available in Python
- Simple statements (4:51 - 7:22)
    - Covers what are simple statements, how expressions can be simple statements, and includes examples
- Lists and the subscription order (2:00 - 5:00)
    - Covers how to access items from a list using subscription, and splicing lists
- Assignment statements (0:00 - 9:57)
    - Covers assignment statements, how they work, how to use them, and includes examples

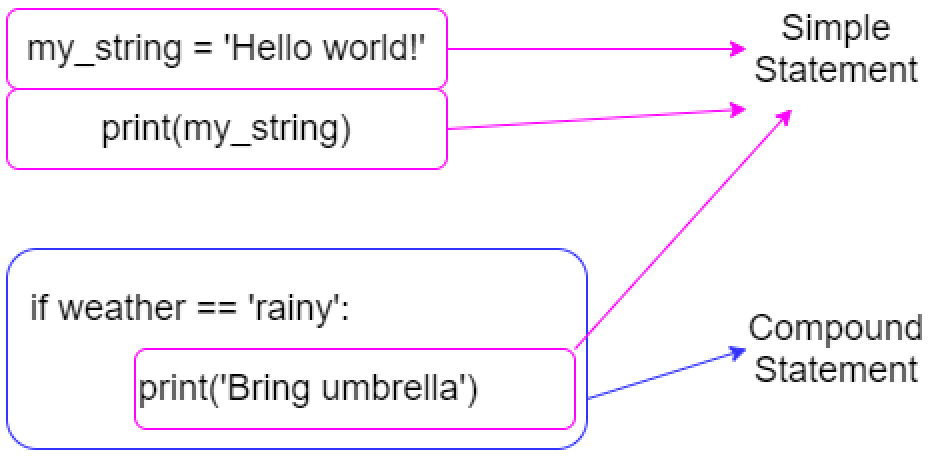
1 Statements

A program describes computations that a computer then executes. A Python program is made up of statements that are evaluated by the Python interpreter, and executed by the machine. There are many types of statements in Python, some of which can be used for:

- Creating data objects
- Giving data object names
- Performing operations on data objects
- Executing statements conditionally or repeatedly

In general, Python statements are organized into two categories: simple statements and compound statements.

In a Python program, oftentimes we'll see statements that span only one line of code. These are called 'simple statements'. These types of statements are 'simple' as compared to 'compound statements', which span multiple lines, and contain simple statements.



```
my_string = 'Hello world!'
print(my_string)

if weather == 'rainy':
    print('Bring umbrella')
```

We'll be covering some of the simple statements that you will encounter in this course; however, a full list of simple statements that exist in Python can be found in the simple statements section of the official Python documentation.

2 Expression Statements

The most fundamental type of statement is an *expression statement*, which consists of an expression. An expression is used to represent certain values (e.g. a string or number), and evaluates to an object. Expressions can be built upon each other, allowing us to combine values and operators, which evaluate into new objects.

There are many kinds of expressions in Python. We’ll be covering atomic, unary and binary expressions, subscription, slicing, attribute references and function calls in this section.

2.1 Atomic Expressions

An atomic expression is the most basic type of expression that evaluates an object. Examples of atomic expression are as follows:

An Identifier	`my_string`	Evaluates to an object that is bound to the identifier
A literal (int, float, str)	`3`	Evaluates to an int object representing the number *three*
	`5.6`	Evaluates to an int object representing the number *five-point-six*
	`'hello'`	Evaluates to a str object representing the sequence of characters *hello*
A list display	`[1, 'hello']`	Evaluates to a list object that holds 2 references: an int object representing the number *one*, and a str object representing the sequence of characters *hello*
The keywords True, False, and None	`True`	Evaluates to the bool object *True*
	`False`	Evaluates to the bool object *False*
	`None`	Evaluates to the NoneType object *None*

2.2 Function Calls

Functions typically represent computations or operations on objects that are commonly used in programs. For certain computations that are frequently used, the lines of code that are used to perform the computation are grouped together, and defined as its own function. Thus, rather than writing the same lines of code repeatedly for a single computation, we can call the function that does it all for us in one line. An example of this can be seen with the `print()` function, where it is used so often in programs that it would be inefficient to write out all the lines needed to perform the computation to display an output.

Function calls are expressions that *evaluate to an object*. This object is the value returned by the function. We can see the return value of a function by writing an assignment statement (will be introduced later), which saves the result of the function call to a variable.

`>>> a = len('hello')`
`>>> a`
`5`
	For example, the `len('hello')` function returns an object of type int representing the number *five*. The result of the function is saved to the variable `a`, which displays the number *five* when evaluated in the Python shell. Thus we can say that this function call `len('hello')` returns a number *five*.

In addition to returning values, some functions may also have ‘side effects’. This means that some functions may be able to change an object, or display something on the screen, open a file, or control the computer hardware in some other way. This can be seen in the following example:

```
>>> a = print('hello')
hello
>>> a
>>>
>>> print(a)
None
```

Here, the value returned by the `print()` function is saved to the variable `a`. The `print()` function has a side effect that, in this case, displays the sequence of characters *hello* in the output. Evaluating `a` in the Python shell does not display anything when evaluating the None object. We can verify that `a` is indeed bound to the NoneType object by calling the print function with the argument `a`, which forces a display of the None object in the Python shell.

Most functions can be called using this format (built-in and user-defined):

function_name(arguments)

- Function name: A function name is an identifier that references an object in memory. Some names are pre-bound to certain built-in functions (eg. `len()` or `print()`).
- Arguments: Arguments can be one or several, comma-separated, expressions. They evaluate to objects that are then passed to the function, so that the function can do its job. For some functions, arguments are optional (e.g. `print()`), for others, they are necessary, and not providing arguments will cause an error to occur. For example the *len* function computes the size of an object and therefore needs an object as an argument (e.g., `len('abc')`).

Some examples of function calls:

`>>> print()`	● Function name: print ● Arguments: no arguments given ● Side effect: nothing is displayed in the output as no arguments were given ● Evaluates to the None object of type NoneType
`>>> len('apples')` `6`	● Function name: len ● Arguments: an object of type str representing the sequence of characters ‘apples’ ● Side effect: no side effect ● Evaluates an object of type int with the value *six*
`>>> print(1, 2 + 4, 3)` `1 6 3`	● Function name: print ● Arguments: three objects ○ An object of type int representing the number *one* ○ A binary expression evaluating to an object of type int representing the number *six* ○ An object of type int representing the number *three* ● Side effect: the three object’s values (*one*, *six*, and *three*) are displayed in the output ● Evaluates to the None object of type NoneType

2.2.1 Built-in Functions

Built-in functions are functions that are always available for use when coding. This means that they do not need to be imported into your code before they can be used.

`>>> type(3)`
`<class 'int'>`
	The built-in function `type()` returns the type of the given argument.

`>>> abs(4 - 10)`
`6`
	The built-in function `abs()` returns an int or float object.

Here's a list of some Python built-in functions:

`print()`	`input()`	`str()`
`type()`	`list()`	etc...

For a full list of built-in functions in Python, refer to the built-in functions section in the Python documentation.

2.2.2 Imported Functions

Imported functions are functions that exist in modules, and are only accessible after the module is imported into the code. Functions from different modules are made available by the import statement, for example:

`>>> import random`
`>>> my_list = ['a', 'b', 'c']`
`>>> random.choice(my_list)`
`'B'`
	The random module is imported at the beginning by using the import statement `import random`. This now gives us access to all the functions in the Python random module. Here, we are using the `choice()` function from the module, which chooses a random item from a sequence and returns it. In this case, the given sequence is a list of str objects, and the str object representing the character *b* is returned.

`>>> from random import randint`
`>>> randint(2, 6)`
`5`
	The `from` keyword can be used to specify which module to import from, then using the `import` keyword, we can import the specific function needed. Once the specific functions are imported, they can be referred to directly by their identifier. Here, the `randint()` function from the Python random module is imported, and we can directly refer to the function without using `random.` as we did in the above example.

For a full list of modules in Python, refer to the Python module index in the Python documentation.

2.3 Subscription Operation

Subscription allows us to select an item from a sequence such as list, string and tuple. It follows the format:

```
sequence[index]
```

String subscription:

`>>> my_string = "apple"`
`>>> my_string[3]`
`'l'`
	A str type object representing the sequence of characters *apple* is created and stored in the variable my_string. The line `my_string[3]` selects the 4th character in the string, which evaluates to a str object representing the character *l*. Remember that indices start at 0, thus the first character in the string has an index of 0.

List subscription:

`>>> my_list = ['a', 'b', 'c']`
`>>> my_list[2]`
`'c'`
	A list type object with 3 references to str objects is created and stored in the variable my_list. The line `my_list[2]` selects the 3rd item in the list, which evaluates to the str object representing the character *c*. Remember that indices start at 0, thus the first item has an index of 0.

Tuple subscription:

`>>> my_tuple = ('a', 'b', 'c')`
`>>> my_tuple[0]`
`'a'`
	A tuple type object with 3 references to str objects is created and stored in the variable my_tuple. The line `my_tuple[0]` selects the 1st item in the tuple, which evaluates to the str object representing the character *a*. Remember that indices start at 0, thus the first item has an index of 0.

A useful thing to note is that to obtain the last item in a sequence, we can put -1 within the square brackets [ ] rather than the last index in the sequence:

```
>>> my_list = ['bag', 'leaf', 'cup', 'bike']
>>> my_list [-1]
'bike'
```

2.4 Slicing

Slicing allows us to select a range of items or characters from a sequence such as a string, list or tuple. It follows the format:

```
sequence [start : stop : step]
```

String slicing:

`>>> my_string = "apple"`
`>>> my_string[0:3]`
`'app'`
	A str type object representing the sequence of characters *apple* is created and stored in the variable my_string. The line `my_string[0:3]` selects the characters from the 0th index up to and not including the character at the 3rd index. This then evaluates to a new str object representing the sequence of characters *app*. Remember that indices start at 0, thus the first character in the string has an index of 0.

List slicing:

`>>> my_list = ['a', 'b', 'c']`
`>>> my_list[1:3]`
`['b', 'c']`
	A list type object with 3 references to str objects is created and stored in the variable my_list. The line `my_list[1:3]` selects the items from index 1 up to and not including index 3. This evaluates to a list object with 2 references to str objects. Remember that indices start at 0, thus the first item has an index of 0.

Tuple subscription:

`>>> my_tuple = ('a', 'b', 'c')`
`>>> my_tuple[0:2]`
`('a', 'b')`
	A tuple type object with 4 references to str objects is created and stored in the variable my_tuple. The line `my_tuple[0:2]` selects the items from index 0 up to and not including index 2. This evaluates to a tuple object with 2 references to str objects. Remember that indices start at 0, thus the first item has an index of 0.

Here are some useful slicing tricks:

We'll use this list as an example:

```
>>> my_list = ['bag', 'leaf', 'cup', 'bike']
```

`>>> my_list[2:]`
`['cup', 'bike']`
	This selects all items starting from index 2 to the end of the sequence.

`>>> my_list[:2]`
`['bag', 'leaf']`
	This selects all items from the start of the sequence, and up to and not including the item at index 2.

`>>> my_list[::2]`
`['bag', 'cup']`
	This selects all items starting from index 0, and every second item afterwards.

Note that these operations (subscription and slicing) can be applied to any objects of the right type, and thus, the operations can be combined. For example:

```
>>> my_string = 'cassowary'
>>> my_string[5:9][2]
'r'
```

Slicing is applied first to the str object, where the characters at index 5 up to and not including index 9 is selected, resulting in a str object representing the sequence of characters *wary*. Then subscription is applied to select the character at index 2 from the result of the sliced str object, which gives us the result of the str object representing the character *r*.

2.5 Attribute Reference

Some objects in Python have so-called attributes. This can be seen in modules. Modules may contain functions and constants, which are considered attributes of that specific module. For instance, the Python math module has functions such as `ceil()` and `floor()`, and constants such as `e` and `pi`.

To reference an attribute we follow the format:

```
expression.name_of_attribute
```

Where `expression` evaluates to an object that contains the attributes we would like to access, and `name_of_attribute` is the name of the specific attribute we would like to access.

Let's take a look at the Python math module as an example:

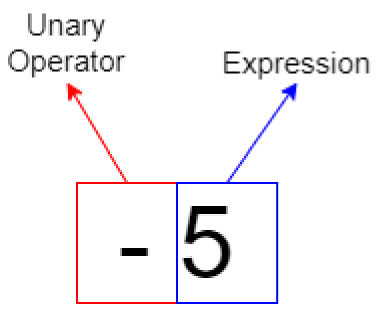
`>>> import math`
`>>> math.pi`
`3.141592653589793`
	Here we import the Python math module, so that we can access its attributes. `math` is used to reference the module type object that was just imported into the code, and `.pi` specifies that we would like to access the attribute `pi`, which is bound to an approximation of the mathematical value of π, from the Python math module. This line is then evaluated to an object of type float, displaying a representation of π.

Attribute references will be covered more in depth in the user-defined classes section.

2.6 Unary and Binary Expressions

Unary and binary expressions build upon other expressions with the addition of an operator.

A unary expression is made up of a unary operator followed by an expression. Unary operators include `+`, `-` and `not`.

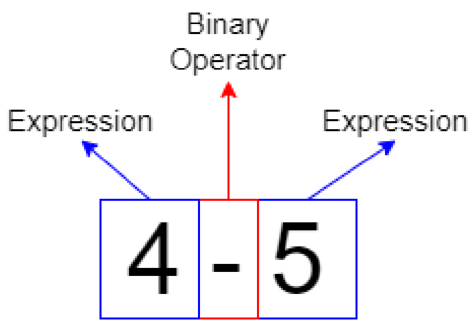


In this example, the unary expression is made up of the unary operator `-`, and the atomic expression literal int `5`.

A binary expression is made up of an expression followed by a binary operator, followed by another expression.

A binary operator can be:

- Any operator except for ~
- Individual keywords or, and, in, and is
- Keyword pairs not in, is not



In this example, the binary expression is made up of the atomic expression literal int `4`, the binary operator `-`, and the atomic expression literal int `5`.

Some more examples of unary and binary expressions:

The plus operator (+):
The plus operator is used for addition with integers, and concatenation when used with strings.

`>>> +5`	This unary expression combines the unary operator + with the atomic expression literal int `5`. The expression evaluates to an object of type int representing the number *five*. Here, the unary operator doesn't alter the value of the literal int `5`.

`>>> 27 + 4.3`
`31.3`
	This binary expression combines the binary operator +, and two atomic expressions: a literal int `27`, and a literal float `4.3`. The expression evaluates to an object of type float representing the number *thirty-one-point-three*.

`>>> 'apple' + 'pen'`	This binary expression combines two atomic expressions – a literal string `'apple'`, and a literal string `'pen'` – with the binary operator +. The expression evaluates to an object of type str which represents the sequence of characters *applepen*. This operation is also called string concatenation.

The minus operator (-):
The minus operator is used to make numbers (int or float) negative, and can also be used with int objects to perform subtraction.

`>>> -+3`
`-3`
	This unary expression combines the unary operator -, and another unary expression `+3`. The second unary expression can be further broken down into the unary operator +, and an atomic expression, the literal int `3`. The expression evaluates to an object of type int representing the number *minus three*.

`>>> 40 - 10`
`30`
	This binary expression combines the atomic expression literal int `40`, the binary operator `-`, and the atomic expression literal int `40`. The expression evaluates to an object of type int representing the number *thirty*.

Keywords (not, or, and, in, and is):
Specific keywords and keyword pairs can be combined with operators to create unary or binary expressions.

`>>> not True`
`False`
	This unary expression combines the unary operator keyword `not`, and the boolean atomic expression `True`. The expression evaluates to an object of bool type with the value *False*.

`>>> True or False`
`True`
	This binary expression combines the boolean atomic expression `True`, the binary operator keyword `or`, and the boolean atomic expression `False`. The expression evaluates to an object of bool type with the value *True*.

`>>> False and True`
`False`
	This binary expression combines the boolean atomic expression `False`, the binary operator keyword `and`, and the boolean atomic expression `True`. The expression evaluates to an object of bool type with the value *False*.

`>>> my_list = ['apple', 3,`
`'cat']`
`>>> 'apple' in my_list`
`True`
	This binary expression combines the atomic expression literal string `'apple'`, the binary operator keyword `in`, and the identifier `my_list` which holds an atomic expression of a list display. The list display evaluates to an object of list type,

	holding references to 3 items. The expression checks if the str object `'apple'` exists in the list object, and evaluates to an object of bool type with the value *True* because `'apple'` does exist within the list.

`>>> a = 300`
`>>> b = 300`
`>>> a is b`
`False`
`>>> id(a)`
`23019200`
`>>> id(b)`
`23019264`
	The identifiers a and b both hold an object of type int, representing the number *three-hundred*. The binary expression then combines the atomic expression identifier a, the binary operator keyword is, and the identifier b. The expression checks if the memory location or address of the objects are the same. Note that the is keyword differs from the == operator, which checks if the values of the two objects being compared are the same, or if they are identical to each other. In this case, the expression evaluates to an object of bool type with the value *False*. By using the `id()` function, we can see that these two int objects reside at different locations in the memory.

`>>> my_list = [1, 2, 3]`
`>>> 4 not in my_list`
`True`
	This binary expression combines the atomic expression literal int `'4'`, the binary operator keyword pair `not in`, and the identifier `my_list` which holds an atomic expression of a list display. The list display evaluates to an object of list type, holding references to 3 items. The expression checks if the int object 4 does not exist in the list object, and evaluates to an object of bool type with the value *True* because 4 does not exist within the list.

`>>> a = 300`
`>>> b = 300`
`>>> a is not b`
`True`
`>>> id(a)`
`23019200`
`>>> id(b)`
`23019264`
	The identifiers `a` and `b` both hold an object of type int, representing the number *three-hundred*. The binary expression then combines the atomic expression identifier `a`, the binary operator keyword pair is not, and the identifier `b`. The expression checks if the memory location or address of the objects are the same. Note that the is not keyword pair differs from the != operator, which checks if the values of the two objects being compared are not the same, or if they are not identical to each other. In this case, the expression evaluates to an object of bool type with the value *True*. By using the `id()` function, we can see that these two int objects reside at different locations in the memory.

Parenthesized Expressions:

Similar to how parentheses can be used to dictate the order of operations in math, parentheses can also be used in code to change how certain expressions are evaluated, and their results.

`>>> -4 - (3 * -4)`
`8`
	This binary expression combines the binary operator -, and a unary expression (`-4`) with a binary expression (`(3 * -4)`).
	● The unary expression `-4` evaluates to an object of type int representing the number *minus four*.
	● The binary expression `(3 * -4)` combines the atomic expression `3`, and the unary expression `-4` with the binary operator *. The expression evaluates to an object of type int representing the number *minus twelve*.

	Finally, the results from the two expressions are put together, resulting in binary expression `-4 - -12`. This then is evaluated to an object of type int representing the number *eight*.

`>>> 4 * 3 - 5 + 6`
`13`
	This expression will be evaluated in the following order:
	● The binary expression `4 * 3` is evaluated to an object of int type representing the number *twelve*
	● The binary expression `12 - 5` (`12` comes from the binary expression above) is evaluated to an object of type int representing the number *seven*
	● The binary expression `7 + 6` (`7` comes from the binary expression above) is evaluated to an object of type int representing the number *thirteen*

`>>> 4 * (3 - 5) + 6`
`-2`
	This expression will be evaluated in the following order:
	● The binary expression `3 - 5` is evaluated to an object of int type representing the number *minus two*
	● The binary expression `4 * -2` (`-2` comes from the binary expression above) is evaluated to an object of type int representing the number *minus eight*
	● The binary expression `-8 + 6` (`-8` comes from the binary expression above) is evaluated to an object of type int representing the number *minus two*

`>>> 4 * 3 - (5 + 6)`
`1`
	This expression will be evaluated in the following order:
	● The binary expression `5 + 6` is evaluated to an object of int type representing the number *eleven*
	● The binary expression `4 * 3` (by following order of operations) is evaluated to an object of type int

	representing the number *twelve* ● The binary expression `12 - 11` (`12` and `11` comes from the binary expressions above) is evaluated to an object of type int representing the number *one*

3 Assignment Statements

Assignment statements are how we give names to objects in memory. Suppose we run the following arithmetic expression in they Python shell:

`>>> 500 + 300`
`800`
	An int object representing the number *eight-hundred* has been created, but we have no way to refer to that object. There is no way to reuse that object in further statements because we have not saved the object.

Once we give a name to the object, in an assignment statement, we will be able to use it again later.

The basic form of an assignment statement is:

```
<identifier> = <expression that evaluates to an object>
```

For example:

`>>> size = 500 + 300`
`>>> size`
`800`
`>>> size + 100`
`900`
	The first line is an assignment statement that assigns the resulting value of the binary expression `500 + 300` to the identifier `size`. Now that the int object representing the number *eight-hundred* is saved to `size`, we can refer to the identifier `size` for that value.

4 Import Statements

The import statement allows us to gain access to different modules with specific functionalities. Let's take a look at the Python time module for example:

`>>> import time`
`>>> time.sleep(3)`
	Here, we use the import statement to gain access to the Python time module. This allows us to use the `sleep()` function. The `sleep()` function is called here with the given argument of the int object 3, which tells the program to pause for 3 seconds before resuming.

Sometimes we may only need one or a few functions from a module. If this is the case, we can specifically import those functions rather than the entire module.

`>>> from time import sleep`
`>>> sleep(3)`
	The `from` keyword can be used to specify which module to import from, then using the `import` keyword, we can import the specific functions needed. Once the specific functions are imported, they can be referred to directly by their identifier. Here, the `sleep()` function from the Python time module is imported, and we can directly refer to the function without using the `time.` as we did in the above example.

For a full list of all the modules in Python you can import, refer to the Python module index in the Python documentation.