**Python IDE**: an Integrated Development Environment.

**Boolean**

A boolean is the simplest data type; it’s either *True* or *False*.

**Python’s boolean operators**

| **Operator** | **Meaning** |
| --- | --- |
| > | greater than |
| < | smaller than |
| >= | greater than or equal to |
| <= | smaller than or equal to |
| == | is equal |
| != | is not equal |

As can be seen in the examples, these operators work on strings too. Strings are compared in the order of the alphabet, with these added rules:

Uppercase letters are ‘smaller’ than lowercase letters, e.g.: ‘M’ < ‘m’

Digits are smaller than letters: ‘1’ < ‘a’

**Python logical operators**

|  |  |  |
| --- | --- | --- |
| **Operator** | **What is does** | **Examples** |
| and | True if both statements are true | True and False == False False and False == False True and True == True |
| or | True if one of the statements is true | True or False == True True or True == True False or False == False |
| not | Negates the statement that follows | not True == False not False == True |

**Lists**: [1, “a”, “Hello”]

A Python list can contain zero or more objects. A list can contain all the types we’ve seen so far: **numbers**, **strings**, **booleans**, and even other **lists**.

*mylist = [1, 'a', 'Hello']*

*# Loop over the list*

*for item in mylist:*

*print(item)*

*# Access individual item*

*print(mylist[1])*

*>>> mylist[0] + mylist[1]*

*3*

*>>> mylist[2]*

*'Hello'*

*>>> mylist[3][0]*

*>>> mylist = [1, 2, 'Hello', ['a', 'b'] ]*

*'a'*

**Python For-loop**

There are two ways to create a loop in Python.

Let’s first look at Python’s **for-loop**. A for-loop iterates over the individual elements of the object you feed it.

Iterable: An iterable is an object in Python that can return its members one at a time.

On each iteration, an element from iterable is assigned to variable. This variable exists and can be used only inside the loop. Once there is nothing more left, the loop stops and the program continues with the next lines of code.

The general template for a for-loop in Python is:

*for <****variable****> in <****iterable****>:*

*... do something with variable*

**Python While-loop**

while <expression evaluates to True>:

do something

*i = 1*

*while i <= 4:*

*print(i)*

*i = i + 1*

**Infinite loops**

Sometimes you want your software to keep running. In such cases, an infinite loop can be of help.

**Python Function:**

**Function** - A Python function is a named section of a program that performs a specific task and, optionally, returns a value

**Advantages of using functions:**

* *Code reuse* - can be defined once and used many times
* *Parameters* - functions accept a parameter, and you’ll see how this works in a moment. The big advantage here, is that you can alter the behavior of the function by changing the parameter
* *Return values* - a function can return a value or multiple values. This value is often the result of some calculation or operation.

*>>> def say\_hi():*

*... print('Hi!')*

*...*

*>>> say\_hi()*

*Hi!*

keyword ***def***, which is Python’s keyword to define a function.

Next comes our function name, *say\_hi*.

Then we encounter two parentheses, (), which indicate that this function does not accept any parameters (unlike print and len).

We end the line with a colon (:)

And finally, we bump into a feature that sets Python apart from many other programming languages: indentation.

We can make this more interesting by allowing an argument to be passed to our function. Again we define a function with **def**, but we add a variable name between the parentheses:

*>>> def say\_hi(name):*

*... print('Hi', name)*

*...*

*>>> say\_hi('Erik')*

*Hi Erik*

Our function now accepts a value, which gets assigned to the variable *name*. We call such variables the **parameter**, while the actual value we provide (‘Erik’) is called the **argument**.

A Python function can have parameters. The values we pass through these parameters are called arguments.

A function runs until the end of that function, after which Python returns to where the function was called from.

* use the keyword **return** to return a value from our function.

If your function does not return anything, but you still want to return from the function, you can use an empty return statement. **return**

*def say\_hi():*

*... print("Hi", name)*

*... answer = "Hi"*

*...*

*>>> name = 'Erik'*

*>>> say\_hi()*

The variable ***answer*** only **exists inside our function**. We say that the scope of the variable name is limited to the function *say\_hi*, meaning it doesn’t exist outside of this function. If we define a variable at the so-called top-level of a program, it is visible in all places.

**Default values and named parameters** - A compelling Python feature is the ability to provide **default values** for the parameters:

*>>> def welcome(name='learner', location='this tutorial'):*

*... print("Hi", name, "welcome to", location)*

*...*

*>>> welcome()*

*Hi learner welcome to this tutorial*

*>>> welcome(name='John')*

*Hi John welcome to this tutorial*

*>>> welcome(location='this epic tutorial')*

*Hi learner welcome to this epic tutorial*

These parameters are called ***named parameters*** because we specify both the name and the value, instead of just the value. Thanks to these named parameters, the order in which we supply them doesn’t matter.

**Asking for input with Python**

the built-in function **input()**. It does exactly what you expect it to do: ask for input and assign that input to a variable. If you give input a string as an argument, it will print it as a prefix. In this case, it will print ‘*Your name:* ‘ and wait for you to enter your name.

When someone just hits enter when asked for input, the **input()** function returns an empty string. “(*space*)”

**Python objects:**

In Python, everything is an object. Strings, booleans, numbers, and even Python functions are objects.

**Object** - a collection of data (variables) and methods that operate on that data. Objects are defined by a Python class.

**Method** - when a function is part of an object or Python class, we call it a method.

There are **classes** called *str*, *int*, and *bool* - Python’s native classes, but, as I said, we can build our own types of classes too!

*class Car:  
 speed = 0  
 started = False  
 def start(self):  
 self.started = True  
 print("Car started, let's ride!")  
 def increase\_speed(self, delta):  
 if self.started:  
 self.speed = self.speed + delta  
 print('Vrooooom!')  
 else:  
 print("You need to start the car first")  
 def stop(self):  
 self.speed = 0  
 print('Halting')  
car = Car()  
car.increase\_speed(10)  
car.start()  
car.increase\_speed(40)  
car.stop()*

**You need to start the car first**

**Car started, let's ride!**

**Vrooooom!**

**Halting**

When we call a method on a Python object, Python automatically fills in the first variable, which we call **self** by convention.

This first variable is a reference to the object itself, hence its name. We can use this variable to reference other instance variables and functions of this object, like *self.speed* and *self.start(*).

*>>> car1 = Car()*

*>>> car2 = Car()*

*>>> id(car1)*

*139771129539104*

*>>> id(car2)*

*139771129539160*

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

­­­­­­­­­­­­­­­­­­­­\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

*>>> car1.start()*

*Car started, let's ride!*

*>>> car1.increase\_speed(10)*

*'Vrooom!'*

*>>> car1.speed*

*10*

*>>> car2.speed*

*0*

We just started car1 and increased its speed, while car2 is still halted. Inspection of the speeds confirms these are different cars with different states!

**Python Constructor**

A constructor is a function that is called automatically when an object is created. A constructor can optionally accept arguments as well, just like a regular function.

*car = Car()*

we are in fact calling a function! This method, which we did not have to define, is called the **constructor**. It constructs and initializes the object. Every class has one by default, called \_\_init\_\_, even if we don’t define it ourselves. This has to do with inheritance.

We can override the \_\_init\_\_ method, to give it extra abilities by accepting arguments. Let’s redefine the Car class using a custom constructor:

*class Car:*

*def \_\_init\_\_(self, started = False, speed = 0):*

*self.started = started*

*self.speed = speed*

*def start(self):*

*self.started = True*

*print("Car started, let's ride!")*

**Python Inheritance**

Classes can inherit properties and functions from other classes, so you don’t have to repeat yourself. Say, for example, we want our Car class to inherit some more generic functions and variables from a Vehicle class.

class Vehicle:

def \_\_init\_\_(self, started = False, speed = 0):

self.started = started

self.speed = speed

def start(self):

self.started = True

Sometimes you want to override the inherited **\_\_init\_\_** function. To demonstrate, we can create a Motorcycle class. Most motorcycles have a center stand. We’ll add the ability to either put it out or in on initialization:

*class Motorcycle(Vehicle):*

*def \_\_init\_\_(self, center\_stand\_out = False):*

*self.center\_stand\_out = center\_stand\_out*

*super().\_\_init\_\_()*

When you override the constructor, the constructor from the parent class that we inherited is not called at all. If you still want that functionality, you have to call it yourself. This is done with **super()**: it returns a reference to the parent class, so we can call the parent class’s constructor.

In this case, we added functionality for the center stand but removed the option to set the speed and started state in the constructor. If you want, you can add options for speed and started state too and pass those on to the Vehicle constructor.

**Python Modules:**

The Python import statement allows us to import a Python module. In turn, A Python module helps us organize and reuse our code.

So a module is a file that contains Python code, ending with the .py extension. In other words: any Python file is also a module. The name of a module is the same as the file name, excluding the extension. For example, if you create a file named mymodule.py, the name of your module is mymodule.

We can also import specific parts of a module, like a function, variable, or class. To do so, use the following syntax:

*from mymodule import my\_function*

The **my\_function** function is now available to the rest of the program under the name **my\_function**, as if it was defined inside the file itself. So we can use it like this:

*my\_function()*

In the same fashion, we can import multiple elements from a Python module. To do so, use the following syntax:

*from mymodule import my\_function, my\_variable*

To import everything from a module, we can use the following syntax:

*from mymodule import \**

You can use an alias to make the module name shorter:

*import mymodule as m*

**The If \_\_name\_\_ == '\_\_main\_\_' check**

You might have seen it before, and you might have wondered what it does. Why do Python programmers include this check in their script so often? It has everything to do with modules and using modules as scripts.

When we import a module, its name (stored in the variable \_\_name\_\_) is equal to the module name. When a script is executed, its name is always set to the string \_\_main\_\_. Hence, if we check for this name, we can detect if we are running as a script or if we’re being included somewhere as a module.