# Python Operator

## Python Arithmetic Operator

These Python arithmetic operators include Python operators for basic mathematical operations.

**Addition(+)**

|  |
| --- |
| >>> 3+4 # 7 |

**Subtraction(-)**

|  |
| --- |
| >>> 3-4 # -1 |

**Multiplication(\*)**

|  |
| --- |
| >>> 3\*4  # 12 |

**Division(/)**

|  |
| --- |
| >>> 3/4  # 0.75 |

**Exponentiation(\*\*)**

|  |
| --- |
| >>> 3\*\*4  # 81 |

**Floor Division(//)**

|  |
| --- |
| >>> 10//3  # 3 |

**Modulus(%)**

|  |
| --- |
| >>> 4%3  # 1 |

## Python Relational Operator

Relational Python Operator carries out the comparison between operands. They tell us whether an operand is greater than the other, lesser, equal, or a combination of those.

**Less than(<)**

|  |
| --- |
| >>> 3<4  # True |

**Greater than(>)**

|  |
| --- |
| >>> 3>4  # False |

**Less than or equal to(<=)**

|  |
| --- |
| >>> 7<=7  # True |

****Greater than or equal to(>=)****

|  |
| --- |
| >>> 0>=0  # True |

**Equal to(= =)**

|  |
| --- |
| >>> 3==3.0  >>> 1==True  >>> 0==False  # True |

**f. Not equal to(!=)**

|  |
| --- |
| >>> 1!=1.0  # False |

## Python Assignment Operator

An assignment operator assigns a value to a variable. It may manipulate the value by a factor before assigning it. We have 8 assignment operators- one plain, and seven for the 7 arithmetic python operators.

**Assign(=)**

|  |
| --- |
| >>> a=7  >>> print(a)  # 7 |

**Add and Assign(+=)**

|  |
| --- |
| >>> a+=2  >>> print(a)  # 9 |

**Subtract and Assign(-=)**

|  |
| --- |
| >>> a-=2  >>> print(a)  # 7 |

**Divide and Assign(/=)**

|  |
| --- |
| >>> a/=7  >>> print(a)  # 1.0 |

**Multiply and Assign(\*=)**

|  |
| --- |
| >>> a\*=8  >>> print(a)  # 8.0 |

**Modulus and Assign(%=)**

|  |
| --- |
| >>> a%=3  >>> print(a)  # 2.0 |

## Python Logical Operator

These are conjunctions that you can use to combine more than one condition. We have three Python logical operator – and, or, and not that come under python operators.

**and**

|  |
| --- |
| >>> a=7>7 and 2>-1  >>> print(a)  # False |

**or**

|  |
| --- |
| >>> a=7>7 or 2>-1  >>> print(a)  # True |

**not**

|  |
| --- |
| >>> a=not(0)  >>> print(a)  # True |

## Membership Python Operator

These operators test whether a value is a member of a sequence. The sequence may be a list, a string, or a tuple. We have two membership python operators- ‘in’ and ‘not in’.

**in**

|  |
| --- |
| >>> pets=[‘dog’,’cat’,’ferret’]  >>> ‘fox’ in pets  # False  >>> ‘cat’ in pets  # True |

**not in**

|  |
| --- |
| >>> ‘pot’ not in ‘disappointment’  # True |

## Python Identity Operator

Let us proceed towards identity Python Operator.

These operators test if the two operands share an identity. We have two identity operators- ‘is’ and ‘is not’.

**is**

|  |
| --- |
| >>> 2 is 20  # False |

**is not**

|  |
| --- |
| >>> 2 is not ‘2’  # True |

# Variables

Variables and data types in python as the name suggests are the values that vary. In a programming language, a variable is a memory location where you store a value. The value that you have stored may change in the future according to the specifications.

## Variable Definition & Declaration

Python has no additional commands to declare a variable. As soon as the value is assigned to it, the variable is declared.

|  |
| --- |
| >>> x = 10  # variable is declared as the value 10 is assigned to it. |

1. The variable name cannot start with a number. It can only start with a character or an underscore.
2. Variables in python are case sensitive.
3. They can only contain alpha-numeric characters and underscores.
4. No special characters are allowed.

# Data Types In Python

## Numerical Data Types

Numerical data type holds numerical value. In numerical data there are 4 sub types as well. Following are the sub-types of numerical data type:

* Integers
* Float
* Complex Numbers
* Boolean

## Integers

Integers are used to represent whole number values.

|  |
| --- |
| >>> x = 10  # it will be the integer as long as the value is a whole number. |

## Float

Float data type is used to represent decimal point values.

|  |
| --- |
| >>> x = 10.25  >>> y = 12.30 |

## Strings

Strings in python are used to represent unicode character values. Python does not have a character data type, a single character is also considered as a string.

We denote or declare the string values inside single quotes or double quotes. To access the values in a string, we use the indexes and square brackets.

|  |
| --- |
| >>> name = 'edureka'  >>> name[2]  # this will give you the output as 'u' |

****Operations using strings****

|  |
| --- |
| >>> name = 'edureka'  >>> name.upper()  # this will make the letters to uppercase  >>> name.lower()  # this will make the letters to lowercase  >>> name.replace('e', 'E')  # this will replace the letter 'e' with 'E'  >>> name[1: 4]  # this will return the strings starting at index 1 until the index 4. |

## Lists

List is one of the four collection data type that we have in python. When we are choosing a collection type, it is important to understand the functionality and limitations of the collection. Tuple, set and dictionary are the other collection data type is python.

A list is ordered and changeable, unlike strings. We can add duplicate values as well. To declare a list we use the square brackets.

|  |
| --- |
| >>> mylist = [10,20,30,40,20,30, 'edureka']  # this will give you the output as 'u' |

****Accessing values from a list****

|  |
| --- |
| >>> mylist[2:6]  # this will get the values from index 2 until index 6. |

****Adding/Replacing values in a list****

|  |
| --- |
| >>> mylist[6] = 'python'    # this will replace the value at the index 6.    >>> mylist.append('edureka')    # this will add the value at the end of the list.    >>> mylist.insert(5, 'data science')    # this will add the value at the index 5.  >>> len(mylist)  # 7 |

|  |  |
| --- | --- |
| **Method Name** | **Property** |
| clear() | removes all the elements from the list |
| copy() | returns a copy of the list |
| extend() | add the elements of the list to the end of the current list |
| count() | returns the number of elements of the specified value |
| index() | returns the index of the element |
| pop() | removes the element from the specified position |
| remove() | removes the item with the specified value |
| sort() | sorts the list |
| reverse() | returns the reversed list |

## **Tuples**

Tuple is a collection which is unchangeable or immutable. It is ordered and the values can be accessed using the index values. A tuple can have duplicate values as well. To declare a tuple we use the round brackets.

|  |
| --- |
| >>> mytuple = (10,10,20,30,40,50)  # to count the number of elements    >>> mytuple.count(10)  # the output will be 2  # to find the index    >>> mytuple.index(50)  # the output will be 5. since the index number at 50 is 5. |

## Sets

A set is a collection which is unordered, it does not have any indexes as well. To declare a set in python we use the curly brackets.

|  |
| --- |
| >>> myset = {10, 20 , 30 ,40, 50, 50} |

A set does not have any duplicate values, even though it will not show any errors while declaring the set, the output will only have the distinct values.

To access the values in a set we can either loop through the set, or use a membership operator to find a particular value.

|  |
| --- |
| >>> for x in myset:  >>> print(x)  #this will get all the values.  >>> 20 in myset  #this will return true if the value is in the set.  #to add a value in a set  >>> myset.add('edureka')  #to add multiple values in a list  >>> myset.update([ 10, 20, 30, 40, 50])  #to remove an item from a set  >>> myset.remove('edureka')  #we can use the discard or pop method to remove an item from a set as well.  >>> myset = {10, 20, 30}  >>> myset1 = {10,30,50}  >>> myset.issubset(myset1)  #this will return false  >>> myset.union(myset1)  #this will return a set with the union of the two sets. |

Other operations in a dictionary include the following.

|  |  |
| --- | --- |
| **Method Name** | **Property** |
| clear() | clears the items from a set |
| copy() | returns the copy of the set |
| difference() | returns a set with the difference of the two sets |
| isdisjoint() | returns if the sets have intersection |
| issubset() | returns if the set is a subset |
| symmetricdifference() | returns a set with the symmetric difference |
| update() | update the sets with union of the set |

## Dictionary

A dictionary is just like any other collection array in python. But they have key value pairs. A dictionary is unordered and changeable. We use the keys to access the items from a dictionary. To declare a dictionary, we use the curly brackets.

|  |
| --- |
| >>> mydictionary = { 'python': 'data science', 'machine learning' : 'tensorflow' , 'artificial intelligence': 'keras'}    >>> mydictionary['machine learning']    #this will give the output as 'tensorflow'    >>> mydictionary.get('python')    #this serves the same purpose to access the value. |

Since we are using the keys to access the items, they cannot be duplicate.The values can have duplicate items.

|  |
| --- |
| #adding a new value    >>> mydictionary['analysis'] = 'matplotlib'    #replacing a value    >>> mydictionary['analysis'] = 'pandas'    #deleting a value    >>> mydictionary.pop('analysis')    #remove() , del also serves the same purpose for deleting a value. |

Other operations in a dictionary include the following.

|  |  |
| --- | --- |
| **Method Name** | **Property** |
| copy() | returns a copy of the dictionary |
| clear() | clears the dictionary |
| items() | returns a list containing tuple of key value pairs |
| keys() | returns a list containing all the keys |
| update() | updates the dictionary with all the key-value pairs |
| values() | returns a list of all the values in a dictionary |
| setdefault() | returns the value of a specified key |

## **Range**

**Range is a data type which is mainly used when we are using a loop. Lets take an example to understand this.**

|  |
| --- |
| >>> for x in range(10):  >>> print(x)    #this will print the numbers from 0-10. Range will have the numbers from 0-10 |

# **Escape Sequences**

List of escape sequences available in Python 3.

****\newline****

|  |
| --- |
| >>> print("line1 \  >>> line2 \  >>> line3")  # line1 line2 line3 |

****\\****

|  |
| --- |
| >>> print("\\")  # \ |

****\'****

|  |
| --- |
| >>> print('\'')  # ' |

****\"****

|  |
| --- |
| >>> print("\"")  # " |

****\n****

|  |
| --- |
| >>> print("Hello \n World!")  # Hello  World! |

****\t****

|  |
| --- |
| >>> print("Hello \t World!")  # Hello World! |

# **String Interpolation**

**String Interpolation is the process of substituting values of variables into placeholders in a string, sounds like string concatenation right! But without using + or concatenation methods.**

**Let’s see how many ways string interpolation works in Python.**

* **%-formatting**
* **Str.format()**
* **f-strings**
* **Template Strings**

****%-formatting****

**It’s a feature provided by Python which can be accessed with a % operator. This is similar to printf style function in C.**

|  |
| --- |
| # Python program to demonstrate  # string interpolation    >>> n1 = 'Hello'  >>> n2 ='GeeksforGeeks'    # for single substitution  >>> print("Welcome to % s"% n2)    # for single and multiple substitutions () mandatory  >>> print("% s ! This is % s."%(n1, n2))  # Welcome to GeeksforGeeks  # Hello! This is GeeksforGeeks. |

****Str.format()****

str.format() is one of the string formatting methods in Python3, which allows multiple substitutions and value formatting. This method lets us concatenate elements within a string through positional formatting.

|  |
| --- |
| # Python program to demonstrate  # string interpolation    >>> n1 = 'Hello'  >>> n2 ='GeeksforGeeks'    # for single substitution  >>> print('Hello, {}'.format(n1))    # for single or multiple substitutions  # let's say b1 and b2 are formal parameters  # and n1 and n2 are actual parameters  >>> print("{b1}! This is {b2}.".format(b1 = n1, b2 = n2))    # else both can be same too  >>> print("{n1}! This is {n2}.".format(n2 = n2, n1 = n1))  # Hello, Hello  # Hello! This is GeeksforGeeks.  # Hello! This is GeeksforGeeks. |

****f-strings****

**PEP 498 introduced a new string formatting mechanism known as Literal String Interpolation or more commonly as F-strings (because of the leading f character preceding the string literal). The idea behind f-strings is to make string interpolation simpler.**

**To create an f-string, prefix the string with the letter “ f ”. The string itself can be formatted in much the same way that you would with str.format(). F-strings provide a concise and convenient way to embed python expressions inside string literals for formatting.**

|  |
| --- |
| # Python program to demonstrate  # string interpolation    >>> n1 = 'Hello'  >>> n2 ='GeeksforGeeks'    # f tells Python to restore the value of two  # string variable name and program inside braces {}  >>> print(f"{n1}! This is {n2}")    # inline arithmetic  >>> print(f"(2 \* 3)-10 = {(2 \* 3)-10}") |

# **Type Conversion**

Python defines type conversion functions to directly convert one data type to another which is useful in day to day and competitive programming. This article is aimed at providing the information about certain conversion functions.

## **int()**

This function converts any data type to integer

## **float()**

This function is used to convert any data type to a floating point number

|  |
| --- |
| >>> s = "26"  >>> c = int(s)  # string converting to int  >>> print(d)  # 26 #type int  >>> d = float(c)  # string converting to int  >>> print(d)  # 26.0 #type float |

## **ord()**

This function is used to convert a character to integer.

## **hex()**

This function is to convert integer to hexadecimal string.

## **oct()**

This function is to convert integer to octal string.

## tuple()

This function is used to convert to a tuple.

## set()

This function returns the type after converting to set.

## list()

This function is used to convert any data type to a list type.

## dict()

This function is used to convert a tuple of order (key,value) into a dictionary.

## str()

Used to convert integer into a string.

## complex(real,imag)

This function converts real numbers to complex(real,imag) number.

|  |
| --- |
| >>> a = 1  >>> b = 2  # initializing tuple  >>> tup = (('a', 1) ,('f', 2), ('g', 3))  # printing integer converting to complex number  >>> c = complex(1,2)  >>> print (c) # (1+2j)  # printing integer converting to string  >>> c = str(a)  >>> print (c) # 1  # printing tuple converting to expression dictionary  >>> c = dict(tup)  >>> print (c) # {'a': 1, 'f': 2, 'g': 3} |

# Control Flow Tools

## if Statements

Perhaps the most well-known statement type is the if statement.

|  |
| --- |
| >>> x = int(input("Please enter an integer: "))  # Please enter an integer: 42  >>> if x < 0:  ... x = 0  ... print('Negative changed to zero')  ... elif x == 0:  ... print('Zero')  ... elif x == 1:  ... print('Single')  ... else:  ... print('More')  ...  # More |

## for Statements

The for statement in Python differs a bit from what you may be used to in C or Pascal. Rather than always iterating over an arithmetic progression of numbers (like in Pascal), or giving the user the ability to define both the iteration step and halting condition (as C), Python’s for statement iterates over the items of any sequence (a list or a string), in the order that they appear in the sequence. For example (no pun intended):

|  |
| --- |
| # Measure some strings:  ... words = ['cat', 'window', 'defenestrate']  >>> for w in words:  ... print(w, len(w))  ...  # cat 3  # window 6  # defenestrate 12 |

## The range() Function

If you do need to iterate over a sequence of numbers, the built-in function range() comes in handy. It generates arithmetic progressions:

|  |
| --- |
| >>> for i in range(5):  ... print(i)  # 0  # 1  # 2  # 3  # 4 |

|  |
| --- |
| >>> for i in range(5, 10):  ... print(i)  # 5, 6, 7, 8, 9 |

|  |
| --- |
| >>> a = ['Mary', 'had', 'a', 'little', 'lamb']  >>> for i in range(len(a)):  ... print(i, a[i])  # 0 Mary  # 1 had  # 2 a  # 3 little  # 4 lamb |

## break and continue Statements, and else Clauses on Loops

The break statement, like in C, breaks out of the innermost enclosing for or while loop.

Loop statements may have an else clause; it is executed when the loop terminates through exhaustion of the iterable (with for) or when the condition becomes false (with while), but not when the loop is terminated by a break statement. This is exemplified by the following loop, which searches for prime numbers:

|  |
| --- |
| >>> for n in range(2, 10):  ... for x in range(2, n):  ... if n % x == 0:  ... print(n, 'equals', x, '\*', n//x)  ... break  ... else:  ... # loop fell through without finding a factor  ... print(n, 'is a prime number')  ...  # 2 is a prime number  # 3 is a prime number  # 4 equals 2 \* 2  # 5 is a prime number  # 6 equals 2 \* 3  # 7 is a prime number  # 8 equals 2 \* 4  # 9 equals 3 \* 3 |

|  |
| --- |
| >>> for n in range(1, 8):  ... if n % 2 == 0 :  ... continue  ... print(n)  # 1  # 3  # 5  # 7 |

## while Loop Statements

A while loop statement in Python programming language repeatedly executes a target statement as long as a given condition is true.

|  |
| --- |
| >>> count = 0  >>> while (count < 9):  ... print ('The count is:', count)  ... count += 1  >>> else:  ... print ("Good bye!")# The count is: 0  # The count is: 1  # The count is: 2  # The count is: 3  # The count is: 4  # The count is: 5  # The count is: 6  # The count is: 7  # The count is: 8  # Good bye! |

# Functions

A function is a block of code which only runs when it is called.

You can pass data, known as parameters, into a function.

A function can return data as a result.

|  |
| --- |
| >>> def my\_function():  ... print("Hello from a function") |

To call a function, use the function name followed by parenthesis:

|  |
| --- |
| >>> def my\_function():  ... print("Hello from a function")  >>> my\_function() |

## Arguments

Information can be passed into functions as arguments.

Arguments are specified after the function name, inside the parentheses. You can add as many arguments as you want, just separate them with a comma.

|  |
| --- |
| >>> def my\_function(fname):  ... print(fname + " Refsnes")  >>> my\_function("Emil")  >>> my\_function("Tobias")  >>> my\_function("Linus") |

## Arbitrary Arguments, \*args

If you do not know how many arguments that will be passed into your function, add a \* before the parameter name in the function definition.

This way the function will receive a tuple of arguments, and can access the items accordingly:

|  |
| --- |
| >>> def my\_function(\*kids):  ... print("The youngest child is " + kids[2])  >>> my\_function("Emil", "Tobias", "Linus") |

## Keyword Arguments

You can also send arguments with the key = value syntax.

This way the order of the arguments does not matter.

|  |
| --- |
| >>> def my\_function(child3, child2, child1):  ... print("The youngest child is " + child3)  >>> my\_function(child1 = "Emil", child2 = "Tobias", child3 = "Linus") |

## Arbitrary Keyword Arguments, \*\*kwargs

If you do not know how many keyword arguments that will be passed into your function, add two asterisk: \*\* before the parameter name in the function definition.

This way the function will receive a dictionary of arguments, and can access the items accordingly:

|  |
| --- |
| >>> def my\_function(\*\*kid):  ... print("His last name is " + kid["lname"])  >>> my\_function(fname = "Tobias", lname = "Refsnes") |

## Default Parameter Value

The following example shows how to use a default parameter value.

If we call the function without argument, it uses the default value:

|  |
| --- |
| >>> def my\_function(country = "Norway"):  ... print("I am from " + country)  >>> my\_function("Sweden")  >>> my\_function("India")  >>> my\_function()  >>> my\_function("Brazil") |

## Passing a List as an Argument

You can send any data types of argument to a function (string, number, list, dictionary etc.), and it will be treated as the same data type inside the function.

E.g. if you send a List as an argument, it will still be a List when it reaches the function:

|  |
| --- |
| >>> def my\_function(food):  ... for x in food:  ... print(x)  >>> fruits = ["apple", "banana", "cherry"]  >>> my\_function(fruits) |

## Return Values

To let a function return a value, use the return statement:

|  |
| --- |
| >>> def my\_function(x):  ... return 5 \* x  >>> print(my\_function(3))  >>> print(my\_function(5))  >>> print(my\_function(9)) |

# One-Liner

Swap Two Variables Python One-Liner

|  |
| --- |
| >>> a = 1  >>> b = 2  >>> a, b = b, a  >>> print(a, b)  # 2 1  >>> c = 1  >>> d = 2  >>> c = d  >>> d = c  >>> print(c, d)  # 2 2 |

# Modules

Consider a module to be the same as a code library.

A file containing a set of functions you want to include in your application.

## Create a Module

To create a module just save the code you want in a file with the file extension .py:

Save this code in a file named mymodule.py

|  |
| --- |
| >>> def greeting(name):  ... print("Hello, " + name) |

## Use a Module

Now we can use the module we just created, by using the import statement:

Import the module named mymodule, and call the greeting function:

|  |
| --- |
| >>> import mymodule  >>> mymodule.greeting("Jonathan") |

## Variables in Module

The module can contain functions, as already described, but also variables of all types (arrays, dictionaries, objects etc):

Save this code in the file mymodule.py

|  |
| --- |
| >>> person1 = { ...  "name": "John", ...  "age": 36, ...  "country": "Norway" ...} |

Import the module named mymodule, and access the person1 dictionary:

|  |
| --- |
| >>> import mymodule  >>> a = mymodule.person1["age"]  >>> print(a) |

## Naming a Module

You can name the module file whatever you like, but it must have the file extension .py

## Re-naming a Module

You can create an alias when you import a module, by using the as keyword:

Create an alias for mymodule called mx:

|  |
| --- |
| >>> import packages.mymodule as mx # packages is directory  >>> a = mx.person1["age"]  >>> print(a) |

## Import From Module

You can choose to import only parts from a module, by using the from keyword.

The module named mymodule has one function and one dictionary:

|  |
| --- |
| >>> def greeting(name):  ... print("Hello, " + name)  >>> person1 = {  ... "name": "John",  ... "age": 36,  ... "country": "Norway"  } |

Import only the person1 dictionary from the module:

|  |
| --- |
| >>> from mymodule import person1  >>> print (person1["age"]) |

# List Comprehensions

With this elegant approach, you could rewrite the for loop from the first example in just a single line of code:

|  |
| --- |
| >>> squares = [i \* i for i in range(10)]  >>> squares  # [0, 1, 4, 9, 16, 25, 36, 49, 64, 81] |

|  |
| --- |
| >>> list = [i for i in range(10) if i % 2 == 0]  >>> print(list)  # [0, 2, 4, 6, 8] |

# Python Datetime

## Python Dates

A date in Python is not a data type of its own, but we can import a module named datetime to work with dates as date objects.

|  |
| --- |
| >>> import datetime  >>> x = datetime.datetime.now()  >>> print(x)  # 2020-05-30 10:34:56.697776 |

The date contains year, month, day, hour, minute, second, and microsecond.

The datetime module has many methods to return information about the date object.

Here are a few examples, you will learn more about them later in this chapter:

|  |
| --- |
| >>> import datetime  >>> x = datetime.datetime.now()  >>> print(x.year)  # 2020  >>> print(x.strftime("%A"))  # Saturday |

## Creating Date Objects

To create a date, we can use the datetime() class (constructor) of the datetime module.

The datetime() class requires three parameters to create a date: year, month, day.

|  |
| --- |
| >>> import datetime  >>> x = datetime.datetime(2020, 5, 17)  >>> print(x)  # 2020-05-17 00:00:00 |

## The strftime() Method

The datetime object has a method for formatting date objects into readable strings.

The method is called strftime(), and takes one parameter, format, to specify the format of the returned string:

|  |
| --- |
| >>> import datetime  >>> x = datetime.datetime(2018, 6, 1)  >>> print(x.strftime("%B"))  # June |

A reference of all the legal format codes:

|  |  |  |
| --- | --- | --- |
| **Directive** | **Description** | **Example** |
| %a | Weekday, short version | Wed |
| %A | Weekday, full version | Wednesday |
| %w | Weekday as a number 0-6, 0 is Sunday | 3 |
| %d | Day of month 01-31 | 31 |
| %b | Month name, short version | Dec |
| %B | Month name, full version | December |
| %m | Month as a number 01-12 | 12 |
| %y | Year, short version, without century | 18 |
| %Y | Year, full version | 2018 |
| %H | Hour 00-23 | 17 |
| %I | Hour 00-12 | 05 |
| %p | AM/PM | PM |
| %M | Minute 00-59 | 41 |
| %S | Second 00-59 | 08 |
| %f | Microsecond 000000-999999 | 548513 |
| %z | UTC offset | +0100 |
| %Z | Timezone | CST |
| %j | Day number of year 001-366 | 365 |
| %U | Week number of year, Sunday as the first day of week, 00-53 | 52 |
| %W | Week number of year, Monday as the first day of week, 00-53 | 52 |
| %c | Local version of date and time | Mon Dec 31 17:41:00 2018 |
| %x | Local version of date | 12/31/18 |
| %X | Local version of time | 17:41:00 |
| %% | A % character | % |  |

# Try Except

The try block lets you test a block of code for errors.

The except block lets you handle the error.

The finally block lets you execute code, regardless of the result of the try- and except blocks.

## Exception Handling

When an error occurs, or exception as we call it, Python will normally stop and generate an error message.

These exceptions can be handled using the try statement:

The try block will generate an exception, because x is not defined:

|  |
| --- |
| >>> try:  ... print(x)  >>> except:  ... print("An exception occurred")  # The try block will generate an error, because x is not defined |

## Many Exceptions

You can define as many exception blocks as you want, e.g. if you want to execute a special block of code for a special kind of error:

|  |
| --- |
| >>> try:  ... print(x)  >>> except NameError:  ... print("Variable x is not defined")  >>> except:  ... print("Something else went wrong")  # The try block will generate a NameError, because x is not defined |

## Else

You can use the else keyword to define a block of code to be executed if no errors were raised.

In this example, the try block does not generate any error:

|  |
| --- |
| >>> try:  ... print("Hello")  >>> except:  ... print("Something went wrong")  >>> else:  ... print("Nothing went wrong")  # Hello  # Nothing went wrong |

## Finally

The finally block, if specified, will be executed regardless if the try block raises an error or not.

|  |
| --- |
| >>> try:  ... print(x)  >>> except:  ... print("Something went wrong")  >>> finally:  ... print("The 'try except' is finished")  # Something went wrong  # The 'try except' is finished |

This can be useful to close objects and clean up resources

Try to open and write to a file that is not writable:

|  |
| --- |
| >>> try:  ... f = open("demofile.txt")  ... f.write("Lorum Ipsum")  >>> except:  ... print("Something went wrong when writing to the file")  >>> finally:  ... f.close() |

## Raise an exception

As a Python developer you can choose to throw an exception if a condition occurs.

To throw (or raise) an exception, use the raise keyword.

Raise an error and stop the program if x is lower than 0:

|  |
| --- |
| >>> x = -1  >>> if x < 0:  ... raise Exception("Sorry, no numbers below zero") |

The raise keyword is used to raise an exception.

You can define what kind of error to raise, and the text to print to the user.

|  |
| --- |
| >>> x = "hello"  >>> if not type(x) is int:  ... raise TypeError("Only integers are allowed") |

# Classes and Objects

Python is an object oriented programming language.

Almost everything in Python is an object, with its properties and methods.

A Class is like an object constructor, or a "blueprint" for creating objects.

## Create a Class

To create a class, use the keyword class:

|  |
| --- |
| >>> class MyClass:  ... x = 5 |

## Create Object

Now we can use the class named MyClass to create objects:

|  |
| --- |
| >>> class MyClass:  ... x = 5  >>> p1 = MyClass()  >>> print(p1.x) |

## The \_\_init\_\_() Function

The examples above are classes and objects in their simplest form, and are not really useful in real life applications.

To understand the meaning of classes we have to understand the built-in \_\_init\_\_() function.

All classes have a function called \_\_init\_\_(), which is always executed when the class is being initiated.

Use the \_\_init\_\_() function to assign values to object properties, or other operations that are necessary to do when the object is being created:

|  |
| --- |
| >>> class Person:  ... def \_\_init\_\_(self, name, age):  ... self.name = name  ... self.age = age  >>> p1 = Person("John", 36)  >>> print(p1.name)  >>> print(p1.age)  # John  # 36 |

## Object Methods

Objects can also contain methods. Methods in objects are functions that belong to the object.

Let us create a method in the Person class:

|  |
| --- |
| >>> lass Person:  ... def \_\_init\_\_(self, name, age):  ... self.name = name  ... self.age = age  ... def myfunc(self):  ... print("Hello my name is " + self.name)  >>> p1 = Person("John", 36)  >>> p1.myfunc()  # Hello my name is John |

## The self Parameter

The self parameter is a reference to the current instance of the class, and is used to access variables that belongs to the class.

It does not have to be named self , you can call it whatever you like, but it has to be the first parameter of any function in the class:

|  |
| --- |
| >>> class Person:  ... def \_\_init\_\_(mysillyobject, name, age):  ... mysillyobject.name = name  ... mysillyobject.age = age  ... def myfunc(abc):  ... print("Hello my name is " + abc.name)  >>> p1 = Person("John", 36)  >>> p1.myfunc()  # Hello my name is John |

## Modify Object Properties

You can modify properties on objects like this:

|  |
| --- |
| >>> p1.age = 40 |

## Delete Object Properties

You can delete properties on objects by using the del keyword:

|  |
| --- |
| >>> del p1.age |

## Delete Objects

You can delete objects by using the del keyword:

|  |
| --- |
| >>> del p1 |

# Python Inheritance

Inheritance allows us to define a class that inherits all the methods and properties from another class.

Parent class is the class being inherited from, also called base class.

Child class is the class that inherits from another class, also called derived class.

## Create a Parent Class

Any class can be a parent class, so the syntax is the same as creating any other class:

|  |
| --- |
| >>> class Person:  ... def \_\_init\_\_(self, fname, lname):  ... self.firstname = fname  ... self.lastname = lname  ... def printname(self):  ... print(self.firstname, self.lastname)  # Use the Person class to create an object, and then execute the printname method:  >>> x = Person("John", "Doe")  >>> x.printname() |

## Create a Child Class

To create a class that inherits the functionality from another class, send the parent class as a parameter when creating the child class:

|  |
| --- |
| >>> class Student(Person):  ... pass |

## Use the super() Function

Python also has a super() function that will make the child class inherit all the methods and properties from its parent:

|  |
| --- |
| >>> class Student(Person):  ... def \_\_init\_\_(self, fname, lname):  ... super().\_\_init\_\_(fname, lname) |

## Add Properties

Add a year parameter, and pass the correct year when creating objects:

|  |
| --- |
| >>> class Student(Person):  ... def \_\_init\_\_(self, fname, lname, year):  ... super().\_\_init\_\_(fname, lname)  ... self.graduationyear = year  >>> x = Student("Mike", "Olsen", 2019) |

## Add Methods

Add a method called welcome to the Student class:

|  |
| --- |
| >>> class Student(Person):  ... def \_\_init\_\_(self, fname, lname, year):  ... super().\_\_init\_\_(fname, lname)  ... self.graduationyear = year  ... def welcome(self):  ... print("Welcome", self.firstname, self.lastname, "to the class of", >>> self.graduationyear) |

# Encapsulation

An objects variables should not always be directly accessible.

To prevent accidental change, an objects variables can sometimes only be changed with an objects methods. Those type of variables are private variables.

The methods can ensure the correct values are set. If an incorrect value is set, the method can return an error.

|  |
| --- |
| >>> class Student(Person):  ... def \_\_init\_\_(self, fname, lname, year):  ... super().\_\_init\_\_(fname, lname)  ... self.graduationyear = year  ... def welcome(self):  ... print("Welcome", self.firstname, self.lastname, "to the class of", >>> self.graduationyear) |

Python does not have the private keyword, unlike some other object oriented languages, but encapsulation can be done.

Instead, it relies on the convention: a class variable that should not directly be accessed should be prefixed with an underscore.

|  |
| --- |
| >>> class Robot(object):  ... def \_\_init\_\_(self):  ... self.a = 123  ... self.\_b = 123  ... self.\_\_c = 123  >>> obj = Robot()  >>> print(obj.a)  >>> print(obj.\_b)  >>> print(obj.\_\_c) |

If you run the program you see:

|  |
| --- |
| # 123  # 123  # Traceback (most recent call last):  # File "test.py", line 10, in &lt;module&gt;  # print(obj.\_\_c)  # AttributeError: 'Robot' object has no attribute '\_\_c' |

So what’s with the underscores and error?

A single underscore: Private variable, it should not be accessed directly. But nothing stops you from doing that (except convention).

A double underscore: Private variable, harder to access but still possible.

Both are still accessible: Python has private variables by convention.

## Getters and setters

Private variables are intended to be changed using getter and setter methods. These provide indirect access to them:

|  |
| --- |
| >>> class Robot(object):  ... def \_\_init\_\_(self):  ... self.\_\_version = 22  ... def getVersion(self):  ... print(self.\_\_version)  ... def setVersion(self, version):  ... self.\_\_version = version  >>> obj = Robot()  >>> obj.getVersion()  >>> obj.setVersion(23)  >>> obj.getVersion()  >>> print(obj.\_\_version)  # 22  # 23 |

# Special Methods

In this section, we will learn about a variety of instance methods that are reserved by Python, which affect an object’s high level behavior and its interactions with operators. These are known as special methods. \_\_init\_\_ is an example of a special method; recall that it controls the process of creating instances of a class. Similarly, we will see that \_\_add\_\_ controls the behavior of an object when it is operated on by the + symbol, for example. In general, the names of special methods take the form of \_\_<name>\_\_, where the two underscores preceed and succeed the name. Accordingly, special methods can also be referred to as “dunder” (double-underscore) methods. Learning to leverage special methods will enable us to design elegant and powerful classes of objects.

These methods give us complete control over the various high-level interfaces that we use to interact with objects. Let’s make a simple class with nonsensical behavior to demonstrate our ability to shape how our class behaves:

|  |
| --- |
| # Demonstrating (mis)use of special methods  >>> class SillyClass:  ... def \_\_getitem\_\_(self, key):  ... """ Determines behavior of `self[key]` """  ... return [True, False, True, False]  ... def \_\_pow\_\_(self, other):  ... """ Determines behavior of `self \*\* other` """  ... return "Python Like You Mean It"  >>> silly = SillyClass()  >>> silly[None]  # [True, False, True, False]  >>> silly \*\* 2  # 'Python Like You Mean It' |

This section is not meant to be a comprehensive treatment of special methods, which would require us to reach beyond our desired level of sophistication. The official Python documentation provides a rigorous but somewhat inaccessible treatment of special methods. Dive into Python 3 has an excellent appendix on special methods. It is strongly recommended that readers consult this resource.

## Mapping Operators to Functions

This table shows how abstract operations correspond to operator symbols in the Python syntax and the functions in the operator module.

|  |  |  |
| --- | --- | --- |
| Operation | Syntax | Function |
| Addition | a + b | add(a, b) |
| Concatenation | seq1 + seq2 | concat(seq1, seq2) |
| Containment Test | obj in seq | contains(seq, obj) |
| Division | a / b | truediv(a, b) |
| Division | a // b | floordiv(a, b) |
| Bitwise And | a & b | and\_(a, b) |
| Bitwise Exclusive Or | a ^ b | xor(a, b) |
| Bitwise Inversion | ~ a | invert(a) |
| Bitwise Or | a | b | or\_(a, b) |
| Exponentiation | a \*\* b | pow(a, b) |
| Identity | a is b | is\_(a, b) |
| Identity | a is not b | is\_not(a, b) |
| Indexed Assignment | obj[k] = v | setitem(obj, k, v) |
| Indexed Deletion | del obj[k] | delitem(obj, k) |
| Indexing | obj[k] | getitem(obj, k) |
| Left Shift | a << b | lshift(a, b) |
| Modulo | a % b | mod(a, b) |
| Multiplication | a \* b | mul(a, b) |
| Matrix Multiplication | a @ b | matmul(a, b) |
| Negation (Arithmetic) | - a | neg(a) |
| Negation (Logical) | not a | not\_(a) |
| Positive | + a | pos(a) |
| Right Shift | a >> b | rshift(a, b) |
| Slice Assignment | seq[i:j] = values | setitem(seq, slice(i, j), values) |
| Slice Deletion | del seq[i:j] | delitem(seq, slice(i, j)) |
| Slicing | seq[i:j] | getitem(seq, slice(i, j)) |
| String Formatting | s % obj | mod(s, obj) |
| Subtraction | a - b | sub(a, b) |
| Truth Test | obj | truth(obj) |
| Ordering | a < b | lt(a, b) |
| Ordering | a <= b | le(a, b) |
| Equality | a == b | eq(a, b) |
| Difference | a != b | ne(a, b) |
| Ordering | a >= b | ge(a, b) |
| Ordering | a > b | gt(a, b) |

Example:

|  |
| --- |
| >>> class spm:  ... def \_\_init\_\_(self, a, b, c):  ... self.a = a  ... self.b = b  ... self.c = c  ... def \_\_add\_\_(self, other):  ... return self.a + other.a, self.b + other.b, self.c + other.c    >>> spm1 = spm(1, 5, 6)  >>> spm2 = spm(2, 8, 9)  >>> print(spm1 + spm2)  # (3, 13, 15) |

# Decorators

In Python, functions are the first class objects, which means that –

Functions are objects; they can be referenced to, passed to a variable and returned from other functions as well.

Functions can be defined inside another function and can also be passed as argument to another function.

Decorators are very powerful and useful tool in Python since it allows programmers to modify the behavior of function or class. Decorators allow us to wrap another function in order to extend the behavior of wrapped function, without permanently modifying it.

In Decorators, functions are taken as the argument into another function and then called inside the wrapper function.

|  |
| --- |
| >>> @gfg\_decorator  ... def hello\_decorator():  ... print("Gfg") |

In the above code, gfg\_decorator is a callable function, will add some code on the top of some another callable function, hello\_decorator function and return the wrapper function.

|  |
| --- |
| >>> def myDecoretor(func):  ... def wrapper():  ... print('#'\*30)  ... func()  ... print('#'\*30)  ... return wrapper  >>> @myDecoretor  >>> def message():  ... print("hello, welcome to my app")  >>> message()  ##############################  hello, welcome to my app  ############################## |

## Decorators with Arguments

Example: Decorators with parameters in Python

|  |
| --- |
| >>> def myDecoretor(func):  ... def wrapper(\*args, \*\*kwargs):  ... print('#'\*30)  ... func(\*args, \*\*kwargs)  ... print('#'\*30)  ... return wrapper  >>> @myDecoretor  >>> def message(name):  ... print(f"hello, welcome {name}")  >>> message("Down Jon") |

|  |
| --- |
| >>> def myDecoretor1(func):  ... def wrapper(\*args, \*\*kwargs):  ... print('#'\*30)  ... func(\*args, \*\*kwargs)  ... print('#'\*30)  ... return wrapper  >>> def myDecoretor2(func):  ... def wrapper(\*args, \*\*kwargs):  ... print('\*'\*30)  ... func(\*args, \*\*kwargs)  ... print('\*'\*30)  ... return wrapper  >>> @myDecoretor2  >>> @myDecoretor1  >>> def message(name):  ... print(f"hello, welcome {name}")  >>> message("Down Jon") |

|  |
| --- |
| >>> def decorator(msg):  ... print(msg)  ... def inner(func):  ... def wrapper(\*args, \*\*kwargs):  ... func(\*args, \*\*kwargs)  ... return wrapper  ... return inner  >>> @decorator("welcome")  >>> def message(name):  ... print(f"Hello' {name}")  >>> message("Jon") |