Contents

[Installation 6](#_Toc73617950)

[Configure and run the debugger 6](#_Toc73617951)

[Install and use packages# 7](#_Toc73617952)

[Python REPL 9](#_Toc73617953)

[Python Identifiers and Keywords 11](#_Toc73617954)

[Keywords 11](#_Toc73617955)

[Identifiers 13](#_Toc73617956)

[Python Simple and Compound Statements 17](#_Toc73617957)

[Simple Statements 17](#_Toc73617958)

[Compound Statements 17](#_Toc73617959)

[Python Values, Types and Variables 20](#_Toc73617960)

[Values and types 20](#_Toc73617961)

[Python Built-in Data Types 21](#_Toc73617962)

[Getting the Data Type 21](#_Toc73617963)

[Setting the Data Type 21](#_Toc73617964)

[Python Statements 23](#_Toc73617965)

[Multi-line statement 23](#_Toc73617966)

[Python : Indentation 24](#_Toc73617967)

[Python Operators and Expressions 25](#_Toc73617968)

[Arithmetic Operators 25](#_Toc73617969)

[Relational Operators 25](#_Toc73617970)

[Bitwise Operators 26](#_Toc73617971)

[Assignment operators 27](#_Toc73617972)

[Identity operators 27](#_Toc73617973)

[Membership Operators 27](#_Toc73617974)

[Expressions 28](#_Toc73617975)

[Order of operations 28](#_Toc73617976)

[String operations: 28](#_Toc73617977)

[Take Input from User in Python 29](#_Toc73617978)

[Take an integer as an Input 29](#_Toc73617979)

[Python Comments 30](#_Toc73617980)

[Multi-line comments 30](#_Toc73617981)

[Docstring in Python 30](#_Toc73617982)

[Declaration of docstrings 30](#_Toc73617983)

[Python Conditional Execution 32](#_Toc73617984)

[if Statement 32](#_Toc73617985)

[Alternative execution: if-else Statement 32](#_Toc73617986)

[Chained conditionals: if…elif…else 32](#_Toc73617987)

[Nested conditionals 32](#_Toc73617988)

[Catching exceptions using try and except 33](#_Toc73617989)

[Python Functions 34](#_Toc73617990)

[Define function and function call 34](#_Toc73617991)

[Abstraction and Reusability 34](#_Toc73617992)

[Modularity 34](#_Toc73617993)

[Namespace Separation 36](#_Toc73617994)

[Parameters and Arguments 36](#_Toc73617995)

[Built-in Functions in Python 36](#_Toc73617996)

[Type Conversion Functions 36](#_Toc73617997)

[Random Numbers 37](#_Toc73617998)

[randint() Function in Python 37](#_Toc73617999)

[choice() Function in Python 37](#_Toc73618000)

[Math functions in Python 37](#_Toc73618001)

[Positional Arguments 37](#_Toc73618002)

[Keyword Arguments 39](#_Toc73618003)

[Default Parameters 40](#_Toc73618004)

[Mutable Default Parameter Values 40](#_Toc73618005)

[Pass-By-Value vs Pass-By-Reference 42](#_Toc73618006)

[Pass-By-Value vs Pass-By-Reference in Python 43](#_Toc73618007)

[Python Lambda Functions 47](#_Toc73618008)

[Lambda Calculus 47](#_Toc73618009)

[History 47](#_Toc73618010)

[First Example 47](#_Toc73618011)

[Anonymous Functions 49](#_Toc73618012)

[Arguments 50](#_Toc73618013)

[Closure 51](#_Toc73618014)

[Python Loops 53](#_Toc73618015)

[Python While Loop 53](#_Toc73618016)

[Python “Infinite loops” and break 53](#_Toc73618017)

[Finishing iterations with continue in Python 54](#_Toc73618018)

[Python for Loop 54](#_Toc73618019)

[The range() function in Python 55](#_Toc73618020)

[Bonus Example: Counting and summing loops 55](#_Toc73618021)

[Bonus Example: Maximum and minimum loops 55](#_Toc73618022)

[Python Strings 56](#_Toc73618023)

[Getting the length of a string using len() function 56](#_Toc73618024)

[String Methods 56](#_Toc73618025)

[String % 56](#_Toc73618026)

[Single, Double and Triple Quotes 57](#_Toc73618027)

[Single Quotes 57](#_Toc73618028)

[Double Quotes 57](#_Toc73618029)

[Triple Quotes 58](#_Toc73618030)

[Traversing a  string with a loop 58](#_Toc73618031)

[String slices 58](#_Toc73618032)

[Strings are immutable 58](#_Toc73618033)

[Looping and counting 59](#_Toc73618034)

[The in operator in Python 59](#_Toc73618035)

[String comparison 59](#_Toc73618036)

[Python Lists 60](#_Toc73618037)

[How to create a list? 60](#_Toc73618038)

[Lists are mutable 60](#_Toc73618039)

[How to access elements from a list? 60](#_Toc73618040)

[List Index 61](#_Toc73618041)

[Negative indexing 61](#_Toc73618042)

[Traversing a list 61](#_Toc73618043)

[List operations 61](#_Toc73618044)

[How to slice lists in Python? 61](#_Toc73618045)

[List methods 62](#_Toc73618046)

[How to delete or remove elements from a list? 62](#_Toc73618047)

[***pop*** *operator* 62](#_Toc73618048)

[***del*** operator 62](#_Toc73618049)

[remove() Function 63](#_Toc73618050)

[Lists and functions 63](#_Toc73618051)

[Lists and strings 64](#_Toc73618052)

[List arguments 64](#_Toc73618053)

[Python Dictionaries 66](#_Toc73618054)

[How to create a dictionary? 66](#_Toc73618055)

[How to access elements from a dictionary? 66](#_Toc73618056)

[How to change or add elements in a dictionary? 67](#_Toc73618057)

[Dictionary Membership Test 67](#_Toc73618058)

[How to delete or remove elements from a dictionary? 67](#_Toc73618059)

[Python Dictionary Methods 68](#_Toc73618060)

[Object Oriented Programming in Python 3 70](#_Toc73618061)

[Define a Class in Python 70](#_Toc73618062)

[Classes vs Instances 71](#_Toc73618063)

[How to Define a Class 71](#_Toc73618064)

[Instantiate an Object in Python 72](#_Toc73618065)

[Class and Instance Attributes 73](#_Toc73618066)

[Instance Methods 75](#_Toc73618067)

[Inherit From Other Classes in Python 76](#_Toc73618068)

[Dog Park Example 77](#_Toc73618069)

[Parent Classes vs Child Classes 78](#_Toc73618070)

[Extend the Functionality of a Parent Class 79](#_Toc73618071)

[Namespaces 82](#_Toc73618072)

[Namespaces in Python 82](#_Toc73618073)

[The Built-In Namespace 82](#_Toc73618074)

[The Global Namespace 83](#_Toc73618075)

[The Local and Enclosing Namespaces 83](#_Toc73618076)

[Variable Scope 84](#_Toc73618077)

[Python Namespace Dictionaries 87](#_Toc73618078)

[The globals() function 87](#_Toc73618079)

[The locals() function 89](#_Toc73618080)

[Modify Variables Out of Scope 90](#_Toc73618081)

[The global Declaration 92](#_Toc73618082)

[The nonlocal Declaration 93](#_Toc73618083)

[Referecnces 96](#_Toc73618084)

# Installation

* Install VS Code (<https://code.visualstudio.com/>)
* Install Python (<https://www.python.org/downloads>)
  + abcd
* Install Python Extension for VS Code
  + Open VS Code and install the extension from:

<https://marketplace.visualstudio.com/items?itemName=ms-python.python>



* + Once you have a version of Python installed, activate it using the **Python: Select Interpreter** command.
  + If VS Code doesn't automatically locate the interpreter you're looking for, refer to [Environments - Manually specify an interpreter](https://code.visualstudio.com/docs/python/environments#_manually-specify-an-interpreter).
  + You can configure the Python extension through settings. Learn more in the [Python Settings reference](https://code.visualstudio.com/docs/python/settings-reference).
* Verify Python installation
  + python –version
  + py –version
* From the terminal window, create a folder and navigate to that folder
* Open that folder in VS Code
* Create a file named hello.py and add the following code to it and save the file:

msg = "Hello World"

print(msg)

* The Python extension then provides shortcuts to run Python code in the currently selected interpreter (**Python: Select Interpreter** in the Command Palette):
  + In the text editor: right-click anywhere in the editor and select **Run Python File in Terminal**. If invoked on a selection, only that selection is run.
  + In Explorer: right-click a Python file and select **Run Python File in Terminal**.

## Configure and run the debugger

First, set a breakpoint on line 2 of hello.py by placing the cursor on the print call and pressing F9. Alternately, just click in the editor's left gutter, next to the line numbers. When you set a breakpoint, a red circle appears in the gutter.

Next, to initialize the debugger, press F5. Since this is your first time debugging this file, a configuration menu will open from the Command Palette allowing you to select the type of debug configuration you would like for the opened file.

Just select **Python File**, which is the configuration that runs the current file shown in the editor using the currently selected Python interpreter.

**Note**: VS Code uses JSON files for all of its various configurations; launch.json is the standard name for a file containing debugging configurations.

Select the Debug menu from the left panel, from the dropdown at the top, select Add Configuration. This will create a launch,json in the .vscode folder. Once this file is created, you will not be required to select the Python from the dropdown to run the code.

You can also work with variables in the **Debug Console** (*If you don't see it, select****Debug Console****in the lower right area of VS Code, or select it from the****...****menu*). Then try entering the following lines, one by one, at the **>** prompt at the bottom of the console:

msg

msg.capitalize()

msg.split()

## Install and use packages[#](https://code.visualstudio.com/docs/python/python-tutorial#_install-and-use-packages)

Let's now run an example that's a little more interesting. In Python, packages are how you obtain any number of useful code libraries, typically from [PyPI](https://pypi.org/). For this example, you use the matplotlib and numpy packages to create a graphical plot as is commonly done with data science. (Note that matplotlib cannot show graphs when running in the [Windows Subsystem for Linux](https://docs.microsoft.com/windows/wsl/about) as it lacks the necessary UI support.)

Return to the **Explorer** view (the top-most icon on the left side, which shows files), create a new file called standardplot.py, and paste in the following source code:

import matplotlib.pyplot as plt

import numpy as np

x = np.linspace(0, 20, 100) # Create a list of evenly-spaced numbers over the range

plt.plot(x, np.sin(x)) # Plot the sine of each x point

plt.show() # Display the plot

Next, try running the file in the debugger using the "Python: Current file" configuration as described in the last section.

Unless you're using an Anaconda distribution or have previously installed the matplotlib package, you should see the message, **"ModuleNotFoundError: No module named 'matplotlib'"**. Such a message indicates that the required package isn't available in your system.

To install the matplotlib package (which also installs numpy as a dependency), stop the debugger and use the Command Palette to run **Terminal: Create New Integrated Terminal** (Ctrl+Shift+`). This command opens a command prompt for your selected interpreter.

A best practice among Python developers is to avoid installing packages into a global interpreter environment. You instead use a project-specific virtual environment that contains a copy of a global interpreter. Once you activate that environment, any packages you then install are isolated from other environments. Such isolation reduces many complications that can arise from conflicting package versions. To create a virtual environment and install the required packages, enter the following commands as appropriate for your operating system:

1. Create and activate the virtual environment

**Note**: When you create a new virtual environment, you should be prompted by VS Code to set it as the default for your workspace folder. If selected, the environment will automatically be activated when you open a new terminal.



**For Windows**

py -3 -m venv .venv

.venv\scripts\activate

If the activate command generates the message "Activate.ps1 is not digitally signed. You cannot run this script on the current system.", then you need to temporarily change the PowerShell execution policy to allow scripts to run (see [About Execution Policies](https://go.microsoft.com/fwlink/?LinkID=135170) in the PowerShell documentation):

Set-ExecutionPolicy -ExecutionPolicy RemoteSigned -Scope Process

**For macOS/Linux**

python3 -m venv .venv

source .venv/bin/activate

1. Select your new environment by using the **Python: Select Interpreter** command from the **Command Palette**.
2. Install the packages

# Don't use with Anaconda distributions because they include matplotlib already.

# macOS

python3 -m pip install matplotlib

# Windows (may require elevation)

python -m pip install matplotlib

# Linux (Debian)

apt-get install python3-tk

python3 -m pip install matplotlib

1. Rerun the program now (with or without the debugger) and after a few moments a plot window appears with the output:



1. Once you are finished, type deactivate in the terminal window to deactivate the virtual environment.

## Python REPL

REPL = Red-Evaluate-Print Loop

The process is:

1. **Read:** take user input.
2. **Eval:** evaluate the input.
3. **Print:** shows the output to the user.
4. **Loop:** repeat.

REPL is an interactive read-evaluate-print loop (REPL) window for each of your Python environments, which improves upon the REPL you get with python.exe on the command line.

Just open a command prompt and run “python” to open the REPL interactive environment:



In the window, start entering python code, which will be executed one line at a time, like an interpreter:



Some more examples:

>>> "hello world"  
'hello world'  
>>>

>>> 128 / 8  
16.0  
>>> 256 \* 4  
1024  
>>>

>>> width = 10  
>>> height = 20  
>>> size = width\*height  
>>> print(size)  
200  
>>>

To quit:

>>> exit()

# Python Identifiers and Keywords

## Keywords

* The keywords are some predefined and reserved words in python that have special meaning. Keywords are used to define the syntax of the coding.
* The keyword cannot be used as an identifier, function, and variable name.
* All the keywords in python are written in lower case expect True and False.
* There are 33 keywords in Python 3.7, let’s go through all of them one by one.

| **No.** | **Keywords** | **Description** |
| --- | --- | --- |
| 1 | **and** | This is a logical operator it returns true if both the operands are true else return false. |
| 2 | **Or** | This is also a logical operator it returns true if anyone operand is true else return false. |
| 3 | **not** | This is again a logical operator it returns True if the operand is false else return false. |
| 4 | **if** | This is used to make a conditional statement. |
| 5 | **elif** | Elif is a condition statement used with if statement the elif statement is executed if the previous conditions were not true |
| 6 | **else** | Else is used with if and elif conditional statement the else block is executed if the given condition is not true. |
| 7 | **for** | This is created for a loop. |
| 8 | **while** | This keyword is used to create a while loop. |
| 9 | **break** | This is used to terminate the loop. |
| 10 | **as** | This is used to create an alternative. |
| 11 | **def** | It helps us to define functions. |
| 12 | **lambda** | It used to define the anonymous function. |
| 13 | **pass** | This is a null statement that means it will do nothing. |
| 14 | **return** | It will return a value and exit the function. |
| 15 | **True** | This is a boolean value. |
| 16 | **False** | This is also a boolean value. |
| 17 | **try** | It makes a try-except statement. |
| 18 | **with** | The with keyword is used to simplify exception handling. |
| 19 | **assert** | This function is used for debugging purposes. Usually used to check the correctness of code |
| 20 | **class** | It helps us to define a class. |
| 21 | **continue** | It continues to the next iteration of a loop |
| 22 | **del** | It deletes a reference to an object. |
| 23 | **except** | Used with exceptions, what to do when an exception occurs |
| 24 | **finally** | Finally is use with exceptions, a block of code that will be executed no matter if there is an exception or not. |
| 25 | **from** | The form is used to import specific parts of any module. |
| 26 | **global** | This declares a global variable. |
| 27 | **import** | This is used to import a module. |
| 28 | **in** | It’s used to check if a value is present in a list, tuple, etc, or not. |
| 29 | **is** | This is used to check if the two variables are equal or not. |
| 30 | **None** | This is a special constant used to denote a null value or avoid. It’s important to remember, 0, any empty container(e.g empty list) do not compute to None |
| 31 | **nonlocal** | It’s declared a non-local variable. |
| 32 | **raise** | This raises an exception |
| 33 | **yield** | It’s ends a function and returns a generator. |

## Identifiers

An identifier is a name used to identify a variable, function, class, module, etc. The identifier is a combination of character digits and underscore. The identifier should start with a character or Underscore then use digit. The characters are A-Z or a-z,a UnderScore ( \_ ) and digit (0-9). we should not use special characters ( #, @, $, %, ! ) in identifiers.

**Examples of valid identifiers:**

var1

\_var1

\_1\_var

var\_1

**Examples of invalid identifiers:**

!var1

1var

1\_var

var#1

**Example of and, or, not, True, False keywords:**

print("example of True, False, and, or not keywords")

#  compare two operands using and operator

print(True and True)

# compare two operands using or operator

print(True or False)

# use of not operator

print(not False)

**Example of a break, continue.**

# execute for loop

for i in range(1, 11):

    # print the value of i

    print(i)

    # check the value of i is less then 5

    # if i lessthen 5 then continue loop

    if i < 5:

        continue

    # if i greather then 5 then break loop

    else:

        break

**Example of for, in, if, elif and else keyword:**

# run for loop

for t in range(1, 5):

  # print one of t ==1

    if t == 1:

        print('One')

   # print two if t ==2

    elif t == 2:

        print('Two')

    else:

        print('else block execute')

**Example of def, if and else keywords:**

# define GFG() function using def keyword

def GFG():

    i=20

    # check i is odd or not

    # using if and else keyword

    if(i % 2 == 0):

        print("given number is even")

    else:

        print("given number is odd")

# call GFG() function

GFG()

**Example try, except, raise:**

def fun(num):

    try:

        r = 1/num

    except:

        print('Exception raised.')

        return

    return r

print(fun(10))

print(fun(0))

**Example of a lambda keyword:**

# define a anonymous using lambda keyword

# this labda function increment the value of b

a = lambda b: b+1

# run a for loop

for i in range(1, 6):

    print(a(i))

**Use of return keyword:**

# define a function

def fun():

  # declare a variable

    a = 5

    # return the value of a

    return a

# call fun method and store

# it's return value in a variable

t = fun()

# print the value of t

print(t)

**Use of a del keyword:**

# create a list

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

# print list before using del keyword

print(l)

del l[2]

# print list after using del keyword

print(l)

**Use of global keyword:**

# declare a variable

gvar = 10

# create a function

def fun1():

  # print the value of gvar

    print(gvar)

# declare fun2()

def fun2():

  # declare global value gvar

    global gvar

    gvar = 100

# call fun1()

fun1()

# call fun2()

fun2()

**Example of yield keyword:**

def Generator():

    for i in range(6):

        yield i+1

t = Generator()

for i in t:

    print(i)

**Example of assert keyword:**

def sumOfMoney(money):

    assert len(money) != 0,"List is empty."

    return sum(money)

money = []

print("sum of money:",sumOfMoney(money))

# Python Simple and Compound Statements

We write code blocks in Python and each code block contains sequence of statements. We classified these statements as simple and compound statements. Python program contains collection of these statements; assignments, expressions, computations, functions, loops etc.

## Simple Statements

The statements which are meant for simple operations and mostly written in a single logical line of code.

**For example**, assignment statements are simple statements.

x = 10

which means, we are assigning a value “10” to the variable “x”. This we call as simple statement.

The computation statements (expression statements) also we call simple statements; these statements will compute or calculate some expressions and return the results.

**For example**, x = (10 + 15) is an expression statement.

Other than Assignment and Expression statements; the statements below also we called as Simple Statements: These are the statements formed with Python keyword(s); some of them are break, continue, return and import.

* **break** Statement – We use ***break*** statement, to bypass the execution of the statements which are defined after the break statement. The execution control will go to end of the Compound Statement. Usually we use this statement, within the Compound Statements.
* **continue** statement – **continue** statement is used to skip the statements execution which are defined after this statement. The execution control will go to the beginning of the Compound Statement. These statements also usually use with the Compound Statements.
* Have you noticed the difference between break & continue statements? Control execution will go to the beginning of the Compound Statement when we use continue; where as for break, the control execution will go to end of the Compound Statement.
* **return** statement -We use **return** statements within the function to return from the function with or without a value.
* **import** statement – To import code modules to current namespace, we use **import** statement. Usually, we write these statements at the beginning of the Program code.

## Compound Statements

A compound statement is a statement comprise of group of statements. The compound statements are usually executed, when a condition satisfies or a code block is called directly or through a function call. Compound Statements are spread into multiple logical lines; but aligned them into a particular group.

Class definitions and Function definitions are Compound Statements

Other Compound Statements are:

* The conditional statement – The if statement
* The statements which are grouped with in the Conditional Compound Statement (**The if statement**) are going to execute when the particular condition is satisfied.
* Condition Loop Statements – The for statement AND the while statement
* **for** statement is used to iterate through the elements of a sequence; whereas the statements within the **while** statement are going to execute when the condition is satisfied.
* Using **while** statement also we can iterate through the elements of a sequence; but we need to write additional code to do this; whereas **for**statement syntax by default supports this.
* An Exception Handler – The try statement
* The group of statements with-in **try** are block are going to execute when an exception occurs.

Putting all together the statements; the complete code looks like below:

|  |
| --- |
| #stmts\_example.py |
|  |
| # import statement |
| import math |
|  |
| x = 100 |
| index = 1 |
|  |
| # Display PI value |
| print("PI Value:\n", math.pi) |
|  |
| # conditional statement - The if statement |
| if ( x == 100 ): |
| x = x / 4 |
| print("\nThe result of (100/4) is:\n", x) |
|  |
|  |
| # The for statement |
| print("\n-- The for statement --\n") |
| print("Elements in the sequence are:") |
| sequence = [1, 2, 3, 4, 5] |
| for element in sequence: |
| print(element) |
|  |
|  |
| # The while statement |
| print("\n-- The while statement --\n") |
| print("Print only EVEN numbers:") |
| while(index < x): |
| if( ( index % 2 ) == 0 ): |
| print(index) |
|  |
| index = index + 1 |
|  |
| # The break & continue statements |
| print("\n-- The break & continue statements --\n") |
| print("Enter any value (0 - exit):") |
| while(1): |
| n = int(input()) |
| if ( n == 0 ): |
| break |
|  |
| # skip EVEN numbers to print |
| if( ( n % 2 ) == 0 ): |
| continue |
|  |
| print("You ENTERED the NUMBER : ", n) |
|  |
|  |
| # The try statement |
| print("-- The try statement --") |
| try: |
| div\_by\_0 = (1 / 0) |
| except: |
| print("Hurray!!! we caught, Divide / 0 Error") |

# Python Values, Types and Variables

## Values and types

* A value is one of the most basic things in any program works with.
* A value may be characters i.e., ‘Hello, World!’ or a number like 1,2.2 ,3.5 etc.
* Values belong to different types: 1 is an integer, 2 is a float and ‘Hello, World!’ is a string etc.

**Numbers:**

Python supports 3 types of numbers: integers, float and complex number. If you want to know  what type a value has you can use type() function. Paste the following code and click the run button to check the output.

print(type(1))  
print(type(2.2))  
print(type(complex(2,3)))

**Strings:**

Strings are defined either with a single quote or a double quotes. The difference between the two is that using double quotes makes it easy to include apostrophes.

print(type('Hello World'))  
print(type("Today's News Paper"))

**Variables:**

A variable is nothing but a name that refers to a value. An assignment statement creates new variables and gives them values.

name="Mr. XYZ"  
id=123  
height=165.5

print(name)  
print(id)  
print(height)

The type of a variable means the type of the value it refers to.

print(type(name))  
print(type(id))  
print(type(height))

You can do assignments on more than one variable “simultaneously” on the same line like the following code.

a, b, c = "Make", "Me", "Analyst"  
d=a+b+c  
print(d)

## Python Built-in Data Types

Python has the following data types built-in by default, in these categories:

|  |  |
| --- | --- |
| **Text Type:** | str |
| **Numeric Types:** | int, float, complex |
| **Sequence Types:** | list, tuple, range |
| **Mapping Type:** | dict |
| **Set Types:** | set, frozenset |
| **Boolean Type:** | bool |
| **Binary Types:** | bytes, bytearray, memoryview |

### Getting the Data Type

You can get the data type of any object by using the type() function:

Example: Print the data type of the variable x:

x = 5

print(type(x))

### Setting the Data Type

In Python, the data type is set when you assign a value to a variable:

|  |  |
| --- | --- |
| **Example** | **Data Type** |
| x = "Hello World" | str |
| x = 20 | int |
| x = 20.5 | float |
| x = 1j | complex |
| x = ["apple", "banana", "cherry"] | list |
| x = ("apple", "banana", "cherry") | tuple |
| x = range(6) | range |
| x = {"name" : "John", "age" : 36} | dict |
| x = {"apple", "banana", "cherry"} | set |
| x = frozenset({"apple", "banana", "cherry"}) | frozenset |
| x = True | bool |
| x = b"Hello" | bytes |
| x = bytearray(5) | bytearray |
| x = memoryview(bytes(5)) | memoryview |

# Python Statements

Statements are instructions or piece of codes that Python interpreter can execute. We have already seen two kinds of statements: print and assignment. There are other kinds of statements like if statement, for statement, while statement etc.

When you type a statement, the interpreter executes it and displays the result, if something is there. If you write a script it usually contains a sequence of statements. If there is more than one statement, the results appear one at a time as the statements execute one by one.

print(100)  
x = 200  
y=400  
z=x+y  
print(z)

## Multi-line statement

In Python, end of a statement is marked by a newline character. But You can write a statement with multiple lines using character (\). Check the following example.

st = "I " + "am" + " Mr." + \  
" X."+" I live in " \  
"city Y."

print(st)

Line continuation is implied inside parentheses ( ), brackets [ ] and braces { } in Python. This is called explicit line continuation. For example, you can write the above multi-line statement as the following code.

st = ("I " + "am" + " Mr." +  
" X."+" I live in "  
"city Y.")

print(st)

In Python, end of a statement is marked by a newline character. But You can write a statement with multiple lines using character (\). Check the following example. You can use [ ] and { } for the same purpose described above.

st = ["I " + "am " + "Mr. " +  
" X."+" I live in "+  
"city Y"]  
print(st)

You can write multiple statements in a single line using semicolons, as following example.

x= 100; y = 200; c = x\*y

print(c)

## Python : Indentation

One of the most distinctive features of Python is its use of certain indentation style to mark blocks of code. Once you are wrting python code just be careful of few things:

* In Python white spaces are important!
* The indentation is important!
* If you write program that is not correctly indented, it shows either errors or does not give result what you want!
* Python is case sensitive!
* You can’t safely mix tabs and spaces in Python

Normally, we use tabs or four whitespaces for indentation.

smallest\_so\_far = 50

for the\_num in [9, 41, 12, 3, 74, 15] :

if the\_num < smallest\_so\_far :

smallest\_so\_far = the\_num

print (smallest\_so\_far)

# Python Operators and Expressions

Operators are special symbols that are useful for doing computations like addition, subtraction, multiplication, division, and exponentiation etc. The operators are always applied to some values which are called operands.

Python has so many built-in operators to perform different arithmetic and logical operations. There are main 7 types of operators in Python.

1. Arithmetic Operators
2. Relational Operators
3. Logical Operators
4. Bitwise Operators
5. Assignment operators
6. Identity operators
7. Membership operators

## Arithmetic Operators



**Examples:**

print(35/6)

print(3.14\*10)

print(10+41)

print(10%4)

print(5\*\*2)

(5+9)\*(15-7)

## Relational Operators

Below table shows the relational operators in Python.These operators are used to compare values.



**Examples:**

a=10

b=10

print(a<b)

print(a>b)

print(a==b)

print(a<=b)

print(a>=b)

## Bitwise Operators

Bitwise operators act on operands bit by bit as if they are string of binary digits.



**Examples:**

a=1

b=2

print(a&b)

print(a|b)

print(a^b)

print(~a)

print(a<<b)

print(a>>b)

## Assignment operators

In Python, we use Assignment operators to assign values to variables. Following table covers all assignment operators available in Python.



## Identity operators

There two identity operators in Python are is and is not. we use Identity Operators to compare the memory location of two objects.



**Examples:**

a = 2

b=7

print(a is not b)

print(a is b)

x=[1,2,3]

y=[1,2,3]

print(x is y)

## Membership Operators

In Python, there are operators that are mainly useful to test for membership in a sequence such as lists, strings or tuples.operators test for membership in a sequence such as lists, strings, tuples, set and dictionary.



**Examples:**

a=[1,2,3,4]

b=3

print(b in a)

x="Make Me Analyst"

y="Analyst"

print(y in x)

print(y not in x)

## Expressions

An expression is a combination of values, variables, and operators. A value all by itself is considered an expression, and so is a variable, so the following are all legal expressions:

## Order of operations

If more than one operator appears in an expression, the order of evaluation depends on the rules of precedence. For mathematical operators, Python follows mathematical convention. The acronym **PEMDAS** is a useful way to remember the rules:

1. Parentheses have the highest precedence. It can be used to force an expression to evaluate in the order you want. Since expressions in parentheses are evaluated first, 2 \* (3-1) is 4, and (1+1)\*\*(5-2) is 8. You can also use parentheses to make an expression easier to read, as in (minute \* 100) / 60, even if it doesn’t change the result.
2. Exponentiation has the next highest precedence, so 2\*\*1+1 is 3, not 4, and 3\*1\*\*3 is 3, not 27.
3. Multiplication and Division have the same precedence, which is higher than Addition and Subtraction, which also have the same precence. So 2\*3-1 is 5, not 4, and 6+4/2 is 8, not 5.
4. Operators with the same precedence are evaluated from left to right. So the expression 5-3-1 is 1, not 3, because the 5-3 happens first and thened 1 is subtracted from 2.

When you have doubt, always put parentheses in your expressions to make sure the computations are performed in the order you intend.

## String operations:

The + operator works perfectly with strings, but keep in mind that it is not addition in the mathematical sense.Actuallly, it performs concatenation, which means joining the strings by linking them end to end. For example:

name="Mr. X"

age="30"

s="I am "+ name + "."+ "My age is "+ age

print(s)

# Take Input from User in Python

Sometimes you would like to take input for a particular variable from the user via keyboard. In python, **input()** function is a built-in function for taking input from user. When this function is called, the program stops and waits for receiving a input. When the user presses Return or Enter, the program resumes and input returns what the user typed as a string.

i = input()  
print(i)

**Output:**

>>> input = input()  
MakeMeAnalyst  
>>> print(input)  
MakeMeAnalyst  
>>>

It is a better to print a prompt telling the user what is the input they should enter . You can pass a string to input to be displayed to the user before pausing for input:

>>> name = input('Enter your name?\n')  
Enter your name?  
Mr. X  
>>> print(name)  
Mr. X  
>>>

The sequence \n at the end of the prompt represents a newline, which is a special character that causes a line break. That’s why the user’s input appears below the prompt.

## Take an integer as an Input

If you expect the user to type an integer, you can try to convert the return value to int using the int() function:

prompt = 'What is your age?\n'  
i=input(prompt)  
print(i)  
print(type(i)) #It returns a string  
i=int(i) #Convert it to integer.  
print(i)

**Output:**

>>>  
What is your age?  
30  
30  
<class ‘str’>  
30  
>>>

# Python Comments

In Python, comments start with the # symbol.

x = 10 # assign 10 to x

print("Value of x is:",x) #print value of x

## Multi-line comments

If you want comments that extend multiple lines, one way of doing it is to put hash (#) in the beginning of each line. For example:

#Here is an example of  
#multi-lines comments  
#in python.

Other way of doing the same multi-line comments just putting triple quotes, either ''' or """.

"""Here is an example of  
multi-lines comments  
in python."""

OR

'''Here is an example of  
multi-lines comments  
in python.'''

## Docstring in Python

Python documentation strings (or docstrings) provide a convenient way of associating documentation with Python modules, functions, classes, and methods. An object’s docstring is defined by including a string constant as the first statement in the object’s definition. For example, the following function defines a docstring:

def my\_fun():

"""Take two numbers as input

and print the sum of the two numbers.

"""

a=10

b=20

c=a+b

print(c)

print(my\_fun.\_\_doc\_\_)

my\_fun()

## Declaration of docstrings

The following Python file shows the declaration of docstrings within a python source file:

"""

Assuming this is file called test.py, then this string is

first statement in the file. This will become the "test" module's

docstring when the file is imported.

"""

class TestClass(object):

"""The test class's docstring"""

def test\_method(self):

"""The test method's docstring"""

def test\_function():

"""The test function's docstring"""

import test  
help(test)  
help(test.TestClass)  
help(test.TestClass.test\_method)  
help(test.test\_function)

**Output:**

>>> import test  
>>> help(test)  
Help on module test:

NAME  
test

DESCRIPTION  
Assuming this is file called test.py, then this string is  
first statement in the file. This will become the “test” module’s  
docstring when the file is imported.

>>> help(test.TestClass.test\_method)  
Help on class TestClass in module test:

>>> help(test.test\_function)  
Help on function test\_method in module test:

# Python Conditional Execution

## if Statement

When we write programs, we almost always need the ability to check conditions and change the behavior of the program accordingly. The simplest form is the if statement. The boolean expression after the if statement is called the condition. We end the if statement with a colon character (:) and the line(s) after the if statement are indented. Check the following example.

x=10  
if x > 0 :

print('x is positive')

## Alternative execution: if-else Statement

A second form of the if statement is alternative execution.,In this case there are two possibilities and the condition determines which one gets executed. The syntax looks like this:

x=11

if x%2 == 0 :

    print('x is even')

else :

    print('x is odd')

## Chained conditionals: if…elif…else

Sometimes there are more than two possibilities. Therefore you need more than two branches. One way to express a computation like that is a chained conditional like below:

a=10

b=20

if a < b:

print('a is less than b')

elif a > b:

print('a is greater than b')

else:

print('a and b are equal')

There is no limit on the number of elif statements. If there is an else clause, it has to be at the end, but there doesn’t have to be one.

a=10

b=20

if a < b:

print('a is less than b')

elif a > b:

print('a is greater than b')

elif a==b:

print('a and b are equal')

## Nested conditionals

One conditional can also be nested within another. You could have written the three-branch example like this:

a=20  
b=10  
if a == b:

print('a and b are equal')

else:

if a < b:

print('a is less than b')

else:

print('a is greater than b')

## Catching exceptions using try and except

There is a conditional execution structure built into Python to handle certain types of expected and unexpected errors called “try / except”. The idea of try and except is that you know that some sequence of instruction(s) may have a problem and you want to add some statements to be executed if an error occurs. These extra statements (the except block) are ignored if there is no error. Lets consider the following the example:

i=int(input("Enter a number\n"))  
print(i)

For the above code if you don’t enter any number just hit enter without giving any input then you will get an error like this:

Traceback (most recent call last):  
File “<pyshell#4>”, line 1, in <module>  
i=int(input(“Enter a number\n”))  
ValueError: invalid literal for int() with base 10: ”

Python starts by executing the sequence of statements in the try block. If all goes well, it skips the except block and proceeds. If an exception occurs in the try block, Python jumps out of the try block and executes the sequence of statements in the except block.

try:

i=int(input("Enter a number\n"))

print(i)

except:

print("Please enter a number")

**Output:**

>>>

Enter a number

Please enter a number  
>>>

# Python Functions

Functions can reduce the program smaller by eliminating repetitive code. Any point of time, if you make a change, you just change it in one place. SO, creating function allows to name a group of statements, which makes your program easier to read, understand, and debug.  Basically, when you define a function, you specify the name and the sequence of statements. Later, you can “call” the function by name. We have already seen one example of a function call:

>> type(10)  
<class 'int'>

Here the name of the function is type. The expression in parentheses is called the argument of the function. The argument is a value or variable that we are passing into the function as input to the function. The result, for the type function, is the type of the argument. A function may “returns” a result. The result is called the return value.

## Define function and function call

def is a keyword that indicates that this is a function definition. A function definition specifies the name of a new function and the sequence of statements that execute when the function is called. Here is an example:

def print\_myname():

print("I'm Mr. K.")

print('I am from city Y.')

Here is the syntax for calling them:

print(print\_myname)  
print(type(print\_myname))  
print\_myname()

### Abstraction and Reusability

The **abstraction of functionality** into a function definition is an example of the Don’t Repeat Yourself (DRY) Principle of software development. This is arguably the strongest motivation for using functions.

### Modularity

Functions allow **complex processes** to be broken up into smaller steps. Imagine, for example, that you have a program that reads in a file, processes the file contents, and then writes an output file. Your code could look like this:

# Main program

# Code to read file in

<statement>

<statement>

<statement>

<statement>

# Code to process file

<statement>

<statement>

<statement>

<statement>

# Code to write file out

<statement>

<statement>

<statement>

<statement>

In this example, the main program is a bunch of code strung together in a long sequence, with whitespace and comments to help organize it. However, if the code were to get much lengthier and more complex, then you’d have an increasingly difficult time wrapping your head around it.

Alternatively, you could structure the code more like the following:

def read\_file():

# Code to read file in

<statement>

<statement>

<statement>

<statement>

def process\_file():

# Code to process file

<statement>

<statement>

<statement>

<statement>

def write\_file():

# Code to write file out

<statement>

<statement>

<statement>

<statement>

# Main program

read\_file()

process\_file()

write\_file()

This example is **modularized**. Instead of all the code being strung together, it’s broken out into separate functions, each of which focuses on a specific task. Those tasks are read, process, and write. The main program now simply needs to call each of these in turn.

### Namespace Separation

A **namespace** is a region of a program in which **identifiers** have meaning (*more about namespaces a bit later*). As you’ll see below, when a Python function is called, a new namespace is created for that function, one that is distinct from all other namespaces that already exist.

The practical upshot of this is that variables can be defined and used within a Python function even if they have the same name as variables defined in other functions or in the main program. In these cases, there will be no confusion or interference because they’re kept in separate namespaces.

This means that when you write code within a function, you can use variable names and identifiers without worrying about whether they’re already used elsewhere outside the function. This helps minimize errors in code considerably.

## Parameters and Arguments

Inside the function, the arguments are assigned to variables called parameters. Here is an example of a user-defined function that takes an argument:

def add(a, b):

add1 = a + b

return add1

x = add(3, 5)

print(x)

## Built-in Functions in Python

Python provides a number of important built-in functions. Those built-in functions can be used without providing the function definition. Few examples are given below.

print(max(1,2,3,4,5))  
print(min(1,2,3,4,5))  
print(len("Hi! I am Mr. K"))

## Type Conversion Functions

Sometimes you need to convert values from one type to another. Python also provides built-in functions for that. For example, The int function takes any value and converts it to an integer, if it can, or give errors otherwise:

print(int('10'))

print(int("Hello! I am Mr. K")) # You will get ValueError for this.

print(int(1.99999))

float converts integers and strings to floating-point numbers:

print(float(12))

print(float('2.190'))

Similarly, str converts its argument to a string:

print(str(12))

print(str(2.1))

## Random Numbers

To create random numbers in python you can use random() function which returns a random float between 0.0 and 1.0 (including 0.0 but not 1.0). Each time you call random, you get the next number in a long series.

Check the following example to produce 5 random numbers.

import random

for i in range(5):

x = random.random()

print(x)

## randint() Function in Python

There is another function called randint which takes the parameters low and high, and returns an integer between low and high (including both).

random.randint(5, 10)

random.randint(15, 20)

## choice() Function in Python

To choose an element from a sequence at random, you can use choice:

t = [1, 2, 3,4,5]

random.choice(t)

## Math functions in Python

Python provides math module for mathematical functions. Before you can use the module, you have to import it:

import math  
print(math.pi)  
print(math.sqrt(16) / 4.0)  
print(math.sin(90))

## Positional Arguments

The most straightforward way to pass arguments to a Python function is with **positional arguments** (also called **required arguments**). In the function definition, you specify a comma-separated list of parameters inside the parentheses:

>>> def f(qty, item, price):

... print(f'{qty} {item} cost ${price:.2f}')

...

When the function is called, you specify a corresponding list of arguments:

>>> f(6, 'bananas', 1.74)

6 bananas cost $1.74

The parameters (qty, item, and price) behave like **variables** that are defined locally to the function. When the function is called, the arguments that are passed (6, 'bananas', and 1.74) are **bound** to the parameters in order, as though by variable assignment.

In some programming texts, the parameters given in the function definition are referred to as **formal parameters**, and the arguments in the function call are referred to as **actual parameters:**

[](https://files.realpython.com/media/t.4eefe0ad45c8.png)

Although positional arguments are the most straightforward way to pass data to a function, they also afford the least flexibility. For starters, the **order** of the arguments in the call must match the order of the parameters in the definition. There’s nothing to stop you from specifying positional arguments out of order, of course:

>>> f('bananas', 1.74, 6)

bananas 1.74 cost $6.00

The function may even still run, as it did in the example above, but it’s very unlikely to produce the correct results. It’s the responsibility of the programmer who defines the function to [document](https://realpython.com/documenting-python-code/) what the **appropriate arguments** should be, and it’s the responsibility of the user of the function to be aware of that information and abide by it.

With positional arguments, the arguments in the call and the parameters in the definition must agree not only in order but in **number** as well. That’s the reason positional arguments are also referred to as required arguments. You can’t leave any out when calling the function:

>>> # Too few arguments

>>> f(6, 'bananas')

Traceback (most recent call last):

File "<pyshell#6>", line 1, in <module>

f(6, 'bananas')

TypeError: f() missing 1 required positional argument: 'price'

Nor can you specify extra ones:

>>> # Too many arguments

>>> f(6, 'bananas', 1.74, 'kumquats')

Traceback (most recent call last):

File "<pyshell#5>", line 1, in <module>

f(6, 'bananas', 1.74, 'kumquats')

TypeError: f() takes 3 positional arguments but 4 were given

Positional arguments are conceptually straightforward to use, but they’re not very forgiving. You must specify the same number of arguments in the function call as there are parameters in the definition, and in exactly the same order. In the sections that follow, you’ll see some argument-passing techniques that relax these restrictions.

## Keyword Arguments

When you’re calling a function, you can specify arguments in the form <keyword>=<value>. In that case, each <keyword> must match a parameter in the Python function definition. For example, the previously defined function f() may be called with **keyword arguments** as follows:

>>> f(qty=6, item='bananas', price=1.74)

6 bananas cost $1.74

Referencing a keyword that doesn’t match any of the declared parameters generates an exception:

>>> f(qty=6, item='bananas', cost=1.74)

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

TypeError: f() got an unexpected keyword argument 'cost'

Using keyword arguments lifts the restriction on argument order. Each keyword argument explicitly designates a specific parameter by name, so you can specify them in any order and Python will still know which argument goes with which parameter:

>>> f(item='bananas', price=1.74, qty=6)

6 bananas cost $1.74

Like with positional arguments, though, the number of arguments and parameters must still match:

>>> # Still too few arguments

>>> f(qty=6, item='bananas')

Traceback (most recent call last):

File "<pyshell#16>", line 1, in <module>

f(qty=6, item='bananas')

TypeError: f() missing 1 required positional argument: 'price'

So, keyword arguments allow flexibility in the order that function arguments are specified, but the number of arguments is still rigid.

You can call a function using both positional and keyword arguments:

>>> f(6, price=1.74, item='bananas')

6 bananas cost $1.74

>>> f(6, 'bananas', price=1.74)

6 bananas cost $1.74

When positional and keyword arguments are both present, all the positional arguments must come first:

>>> f(6, item='bananas', 1.74)

SyntaxError: positional argument follows keyword argument

Once you’ve specified a keyword argument, there can’t be any positional arguments to the right of it.

## Default Parameters

If a parameter specified in a Python function definition has the form <name>=<value>, then <value> becomes a default value for that parameter. Parameters defined this way are referred to as **default or optional parameters**. An example of a function definition with default parameters is shown below:

>>> def f(qty=6, item='bananas', price=1.74):

... print(f'{qty} {item} cost ${price:.2f}')

...

When this version of f() is called, any argument that’s left out assumes its default value:

>>> f(4, 'apples', 2.24)

4 apples cost $2.24

>>> f(4, 'apples')

4 apples cost $1.74

>>> f(4)

4 bananas cost $1.74

>>> f()

6 bananas cost $1.74

>>> f(item='kumquats', qty=9)

9 kumquats cost $1.74

>>> f(price=2.29)

6 bananas cost $2.29

In summary:

* **Positional arguments** must agree in order and number with the parameters declared in the function definition.
* **Keyword arguments** must agree with declared parameters in number, but they may be specified in arbitrary order.
* **Default parameters** allow some arguments to be omitted when the function is called.

## Mutable Default Parameter Values

Things can get weird if you specify a default parameter value that is a **mutable object**. Consider this Python function definition:

>>> def f(my\_list=[]):

... my\_list.append('###')

... return my\_list

...

f() takes a single list parameter, appends the string '###' to the end of the list, and returns the result:

>>> f(['foo', 'bar', 'baz'])

['foo', 'bar', 'baz', '###']

>>> f([1, 2, 3, 4, 5])

[1, 2, 3, 4, 5, '###']

The default value for parameter my\_list is the empty list, so if f() is called without any arguments, then the return value is a list with the single element '###':

>>> f()

['###']

Everything makes sense so far. Now, what would you expect to happen if f() is called without any parameters a second and a third time? Let’s see:

>>> f()

['###', '###']

>>> f()

['###', '###', '###']

Oops! You might have expected each subsequent call to also return the singleton list ['###'], just like the first. Instead, the return value keeps growing. What happened?

In Python, default parameter values are **defined only once** when the function is defined (that is, when the def statement is executed). The default value isn’t re-defined each time the function is called. Thus, each time you call f() without a parameter, you’re performing [.append()](https://realpython.com/python-append/) on the same list.

You can demonstrate this with id():

>>> def f(my\_list=[]):

... print(id(my\_list))

... my\_list.append('###')

... return my\_list

...

>>> f()

140095566958408

['###']

>>> f()

140095566958408

['###', '###']

>>> f()

140095566958408

['###', '###', '###']

The **object identifier** displayed confirms that, when my\_list is allowed to default, the value is the same object with each call. Since lists are mutable, each subsequent .append() call causes the list to get longer. This is a common and pretty well-documented pitfall when you’re using a mutable object as a parameter’s default value. It potentially leads to confusing code behavior, and is probably best avoided.

As a workaround, consider using a default argument value that signals **no argument has been specified**. Most any value would work, but [None](https://realpython.com/null-in-python/) is a common choice. When the sentinel value indicates no argument is given, create a new empty list inside the function:

>>> def f(my\_list=None):

... if my\_list is None:

... my\_list = []

... my\_list.append('###')

... return my\_list

...

>>> f()

['###']

>>> f()

['###']

>>> f()

['###']

>>> f(['foo', 'bar', 'baz'])

['foo', 'bar', 'baz', '###']

>>> f([1, 2, 3, 4, 5])

[1, 2, 3, 4, 5, '###']

Note how this ensures that my\_list now truly defaults to an empty list whenever f() is called without an argument.

## Pass-By-Value vs Pass-By-Reference

In programming language design, there are two common paradigms for passing an argument to a function:

1. **Pass-by-value:** A copy of the argument is passed to the function.
2. **Pass-by-reference:** A reference to the argument is passed to the function.

Other mechanisms exist, but they are essentially variations on these two.

In many programming languages, the following are essentially the distinction between pass-by-value and pass-by-reference:

* **If a variable is passed by value,** then the function has a copy to work on, but it can’t modify the original value in the calling environment.
* **If a variable is passed by reference,** then any changes the function makes to the corresponding parameter will affect the value in the calling environment.

The reason why comes from what a **reference** means in these languages. Variable values are stored in memory. In Pascal and similar languages, a reference is essentially the address of that memory location, as demonstrated below:



In the diagram on the left, x has memory allocated in the main program’s namespace. When f() is called, x is **passed by value**, so memory for the corresponding parameter fx is allocated in the namespace of f(), and the value of x is copied there. When f() modifies fx, it’s this local copy that is changed. The value of x in the calling environment remains unaffected.

In the diagram on the right, x is **passed by reference**. The corresponding parameter fx points to the actual address in the main program’s namespace where the value of x is stored. When f() modifies fx, it’s modifying the value in that location, just the same as if the main program were modifying x itself.

### Pass-By-Value vs Pass-By-Reference in Python

Are parameters in Python pass-by-value or pass-by-reference? The answer is they’re neither, exactly. That’s because a reference doesn’t mean quite the same thing in Python as it does in Pascal.

Recall that in Python, every piece of data is an **object**. A reference points to an object, not a specific memory location. That means assignment isn’t interpreted the same way in Python as it is in Pascal. Consider the following pair of statements in Pascal:

x := 5

x := 10

These are interpreted this way:

* **The variable x** references a specific memory location.
* **The first statement** puts the value 5 in that location.
* **The next statement** overwrites the 5 and puts 10 there instead.

By contrast, in Python, the analogous assignment statements are as follows:

x = 5

x = 10

These assignment statements have the following meaning:

* **The first statement** causes x to point to an object whose value is 5.
* **The next statement** reassigns x as a new reference to a different object whose value is 10. Stated another way, the second assignment rebinds x to a different object with value 10.

In Python, when you pass an argument to a function, a similar **rebinding** occurs. Consider this example:

1>>> def f(fx):

2... fx = 10

3...

4>>> x = 5

5>>> f(x)

6>>> x

75

In the main program, the statement x = 5 on line 5 creates a reference named x bound to an object whose value is 5. f() is then called on line 7, with x as an argument. When f() first starts, a new reference called fx is created, which initially points to the same 5 object as x does:



However, when the statement fx = 10 on line 2 is executed, f() **rebinds** fx to a new object whose value is 10. The two references, x and fx, are **uncoupled** from one another. Nothing else that f() does will affect x, and when f() terminates, x will still point to the object 5, as it did prior to the function call:

[](https://files.realpython.com/media/t.c246f52a6217.png)

You can confirm all this using id(). Here’s a slightly augmented version of the above example that displays the numeric identifiers of the objects involved:

>>> def f(fx):

2... print('fx =', fx, '/ id(fx) = ', id(fx))

3... fx = 10

4... print('fx =', fx, '/ id(fx) = ', id(fx))

5...

6

7>>> x = 5

8>>> print('x =', x, '/ id(x) = ', id(x))

9x = 5 / id(x) = 1357924048

10

11>>> f(x)

12fx = 5 / id(fx) = 1357924048

13fx = 10 / id(fx) = 1357924128

14

15>>> print('x =', x, '/ id(x) = ', id(x))

16x = 5 / id(x) = 1357924048

When f() first starts, fx and x both point to the same object, whose id() is 1357924048. After f() executes the statement fx = 10 on line 3, fx points to a different object whose id() is 1357924128. The connection to the original object in the calling environment is lost.

Argument passing in Python is somewhat of a hybrid between pass-by-value and pass-by-reference. What gets passed to the function is a reference to an object, but the reference is passed by value.

**Note:** Python’s argument-passing mechanism has been called **pass-by-assignment**. This is because parameter names are bound to objects on function entry in Python, and assignment is also the process of binding a name to an object. You may also see the terms pass-by-object, pass-by-object-reference, or pass-by-sharing.

The key takeaway here is that a Python function can’t change the value of an argument by reassigning the corresponding parameter to something else. The following example demonstrates this:

>>> def f(x):

... x = 'foo'

...

>>> for i in (

... 40,

... dict(foo=1, bar=2),

... {1, 2, 3},

... 'bar',

... ['foo', 'bar', 'baz']):

... f(i)

... print(i)

...

40

{'foo': 1, 'bar': 2}

{1, 2, 3}

bar

['foo', 'bar', 'baz']

Here, objects of type int, dict, set, str, and list are passed to f() as arguments. f() tries to assign each to the string object 'foo', but as you can see, once back in the calling environment, they are all unchanged. As soon as f() executes the assignment x = 'foo', the reference is **rebound**, and the connection to the original object is lost.

Does that mean a Python function can never modify its arguments at all? Actually, no, that isn’t the case! Watch what happens here:

>>> def f(x):

... x[0] = '---'

...

>>> my\_list = ['foo', 'bar', 'baz', 'qux']

>>> f(my\_list)

>>> my\_list

['---', 'bar', 'baz', 'qux']

In this case, the argument to f() is a [list](https://realpython.com/python-lists-tuples/#python-lists). When f() is called, a reference to my\_list is passed. You’ve already seen that f() can’t reassign my\_list wholesale. If x were assigned to something else, then it would be bound to a different object, and the connection to my\_list would be lost.

However, f() can use the reference to make modifications inside my\_list. Here, f() has modified the first element. You can see that once the function returns, my\_list has, in fact, been changed in the calling environment. The same concept applies to a dictionary:

>>> def f(x):

... x['bar'] = 22

...

>>> my\_dict = {'foo': 1, 'bar': 2, 'baz': 3}

>>> f(my\_dict)

>>> my\_dict

{'foo': 1, 'bar': 22, 'baz': 3}

Here, f() uses x as a reference to make a change inside my\_dict. That change is reflected in the calling environment after f() returns.

# Python Lambda Functions

Python lambdas are little, anonymous functions, subject to a more restrictive but more concise syntax than regular Python functions.

## Lambda Calculus

Lambda expressions in Python and other programming languages have their roots in lambda calculus, a model of computation invented by Alonzo Church. You’ll uncover when lambda calculus was introduced and why it’s a fundamental concept that ended up in the Python ecosystem.

### History

[Alonzo Church](https://en.wikipedia.org/wiki/Alonzo_Church) formalized [lambda calculus](https://en.wikipedia.org/wiki/Lambda_calculus), a language based on pure abstraction, in the 1930s. Lambda functions are also referred to as lambda abstractions, a direct reference to the abstraction model of Alonzo Church’s original creation.

Lambda calculus can encode any computation. It is [Turing complete](https://simple.wikipedia.org/wiki/Turing_complete), but contrary to the concept of a [Turing machine](https://en.wikipedia.org/wiki/Turing_machine), it is pure and does not keep any state.

[Functional](https://realpython.com/python-functional-programming/) languages get their origin in mathematical logic and lambda calculus, while imperative programming languages embrace the state-based model of computation invented by Alan Turing. The two models of computation, lambda calculus and [Turing machines](https://en.wikipedia.org/wiki/Turing_machine), can be translated into each another. This equivalence is known as the [Church-Turing hypothesis](https://en.wikipedia.org/wiki/Church%E2%80%93Turing_thesis).

Functional languages directly inherit the lambda calculus philosophy, adopting a declarative approach of programming that emphasizes abstraction, data transformation, composition, and purity (no state and no side effects). Examples of functional languages include [Haskell](https://www.haskell.org/), [Lisp](https://en.wikipedia.org/wiki/Lisp_%28programming_language%29), or [Erlang](https://www.erlang.org/).

By contrast, the Turing Machine led to imperative programming found in languages like [Fortran](https://en.wikipedia.org/wiki/Fortran), [C](https://en.wikipedia.org/wiki/C_%28programming_language%29), or [Python](https://www.python.org/).

The imperative style consists of programming with statements, driving the flow of the program step by step with detailed instructions. This approach promotes mutation and requires managing state.

The separation in both families presents some nuances, as some functional languages incorporate imperative features, like [OCaml](http://www.ocaml.org/), while functional features have been permeating the imperative family of languages in particular with the introduction of lambda functions in [Java](https://en.wikipedia.org/wiki/Java_%28programming_language%29), or Python.

Python is not inherently a functional language, but it adopted some functional concepts early on. In January 1994, [map()](https://realpython.com/python-map-function/), filter(), reduce(), and the lambda operator were added to the language.

## First Example

Here are a few examples to give you an appetite for some Python code, functional style.

The identity function, a function that returns its argument, is expressed with a standard Python function definition using the keyword def as follows:

>>> def identity(x):

... return x

identity() takes an argument x and returns it upon invocation.

In contrast, if you use a Python lambda construction, you get the following:

>>> lambda x: x

In the example above, the expression is composed of:

* **The keyword**: lambda
* **A bound variable**: x
* **A body**: x

**Note**: In the context of this article, a **bound variable** is an argument to a lambda function.

In contrast, a **free variable** is not bound and may be referenced in the body of the expression. A free variable can be a constant or a variable defined in the enclosing scope of the function.

You can write a slightly more elaborated example, a function that adds 1 to an argument, as follows:

>>> lambda x: x + 1

You can apply the function above to an argument by surrounding the function and its argument with parentheses:

>>> (lambda x: x + 1)(2)

3

Reduction is a lambda calculus strategy to compute the value of the expression. In the current example, it consists of replacing the bound variable x with the argument 2:

(lambda x: x + 1)(2) = lambda 2: 2 + 1

= 2 + 1

= 3

Because a lambda function is an expression, it can be named. Therefore you could write the previous code as follows:

>>> add\_one = lambda x: x + 1

>>> add\_one(2)

3

The above lambda function is equivalent to writing this:

def add\_one(x):

return x + 1

These functions all take a single argument. You may have noticed that, in the definition of the lambdas, the arguments don’t have parentheses around them. Multi-argument functions (functions that take more than one argument) are expressed in Python lambdas by listing arguments and separating them with a comma (,) but without surrounding them with parentheses:

>>> full\_name = lambda first, last: f'Full name: {first.title()} {last.title()}'

>>> full\_name('guido', 'van rossum')

'Full name: Guido Van Rossum'

The lambda function assigned to full\_name takes two arguments and returns a string interpolating the two parameters first and last. As expected, the definition of the lambda lists the arguments with no parentheses, whereas calling the function is done exactly like a normal Python function, with parentheses surrounding the arguments.

## Anonymous Functions

The following terms may be used interchangeably depending on the programming language type and culture:

* Anonymous functions
* Lambda functions
* Lambda expressions
* Lambda abstractions
* Lambda form
* Function literals

For the rest of this article after this section, you’ll mostly see the term **lambda function**.

Taken literally, an anonymous function is a function without a name. In Python, an anonymous function is created with the lambda keyword. More loosely, it may or not be assigned a name. Consider a two-argument anonymous function defined with lambda but not bound to a variable. The lambda is not given a name:

>>> lambda x, y: x + y

The function above defines a lambda expression that takes two arguments and returns their sum.

Other than providing you with the feedback that Python is perfectly fine with this form, it doesn’t lead to any practical use. You could invoke the function in the Python interpreter:

>>> \_(1, 2)

3

The example above is taking advantage of the interactive interpreter-only feature provided via the underscore (\_). See the note below for more details.

You could not write similar code in a Python module. Consider the \_ in the interpreter as a side effect that you took advantage of. In a Python module, you would assign a name to the lambda, or you would pass the lambda to a function. You’ll use those two approaches later in this article.

**Note**: In the interactive interpreter, the single underscore (\_) is bound to the last expression evaluated. In the example above, the \_ points to the lambda function.

Another pattern used in other languages like JavaScript is to immediately execute a Python lambda function. This is known as an **Immediately Invoked Function Expression** (IIFE, pronounce “iffy”). Here’s an example:

>>> (lambda x, y: x + y)(2, 3)

5

The lambda function above is defined and then immediately called with two arguments (2 and 3). It returns the value 5, which is the sum of the arguments.

Lambda functions are frequently used with higher-order functions, which take one or more functions as arguments or return one or more functions.

A ***higher-order function*** is a function that does at least one of the following:

* takes one or more functions as arguments,
* returns a function as its result.

All other functions are *first-order functions*.

A lambda function can be a higher-order function by taking a function (normal or lambda) as an argument like in the following contrived example:

>>> high\_ord\_func = lambda x, func: x + func(x)

>>> high\_ord\_func(2, lambda x: x \* x)

6

>>> high\_ord\_func(2, lambda x: x + 3)

7

### Arguments

Like a normal function object defined with def, Python lambda expressions support all the different ways of passing arguments. This includes:

* Positional arguments
* Named arguments (sometimes called keyword arguments)
* Variable list of arguments (often referred to as **varargs**)
* Variable list of keyword arguments
* Keyword-only arguments

The following examples illustrate options open to you in order to pass arguments to lambda expressions:

>>> (lambda x, y, z: x + y + z)(1, 2, 3)

6

>>> (lambda x, y, z=3: x + y + z)(1, 2)

6

>>> (lambda x, y, z=3: x + y + z)(1, y=2)

6

>>> (lambda \*args: sum(args))(1,2,3)

6

>>> (lambda \*\*kwargs: sum(kwargs.values()))(one=1, two=2, three=3)

6

>>> (lambda x, \*, y=0, z=0: x + y + z)(1, y=2, z=3)

6

### Closure

A closure is a function where every free variable, everything except parameters, used in that function is bound to a specific value defined in the enclosing scope of that function. In effect, closures define the environment in which they run, and so can be called from anywhere.

The concepts of lambdas and closures are not necessarily related, although lambda functions can be closures in the same way that normal functions can also be closures. Some languages have special constructs for closure or lambda (for example, Groovy with an anonymous block of code as Closure object), or a lambda expression (for example, Java Lambda expression with a limited option for closure).

Here’s a closure constructed with a normal Python function:

1def outer\_func(x):

2 y = 4

3 def inner\_func(z):

4 print(f"x = {x}, y = {y}, z = {z}")

5 return x + y + z

6 return inner\_func

7

8for i in range(3):

9 closure = outer\_func(i)

10 print(f"closure({i+5}) = {closure(i+5)}")

outer\_func() returns inner\_func(), a [nested function](https://realpython.com/inner-functions-what-are-they-good-for/) that computes the sum of three arguments:

* **x** is passed as an argument to outer\_func().
* **y** is a variable local to outer\_func().
* **z** is an argument passed to inner\_func().

To test the behavior of outer\_func() and inner\_func(), outer\_func() is invoked three times in a [for loop](https://realpython.com/python-for-loop/) that prints the following:

x = 0, y = 4, z = 5

closure(5) = 9

x = 1, y = 4, z = 6

closure(6) = 11

x = 2, y = 4, z = 7

closure(7) = 13

On line 9 of the code, inner\_func() returned by the invocation of outer\_func() is bound to the name closure. On line 5, inner\_func() captures x and y because it has access to its embedding environment, such that upon invocation of the closure, it is able to operate on the two free variables x and y.

Similarly, a lambda can also be a closure. Here’s the same example with a Python lambda function:

1def outer\_func(x):

2 y = 4

3 return lambda z: x + y + z

4

5for i in range(3):

6 closure = outer\_func(i)

7 print(f"closure({i+5}) = {closure(i+5)}")

When you execute the code above, you obtain the following output:

closure(5) = 9

closure(6) = 11

closure(7) = 13

On line 6, outer\_func() returns a lambda and assigns it to to the variable closure. On line 3, the body of the lambda function references x and y. The variable y is available at definition time, whereas x is defined at runtime when outer\_func() is invoked.

# Python Loops

## Python While Loop

Iteration is very common in any programming language. Python provides several features to make it easier. One form of iteration in Python is the while statement.  
Flow of execution for a while statement:

* 1. Evaluate the condition is True or False.
  2. If the condition is false, exit the while statement and continue execution atthe next statement.
  3. If the condition is true, execute the body and then go back to step 1.

Here is a simple program:

# To take input from the user.

# n = int(input("Enter n: "))

n = 10

while n <15 :

print(n)

n = n + 1

print('STOP!!!')

For the above loop, we would say, “It had five iterations”, which means that the body of the loop was executed five times.

The body of the loop should change the value of one or more variables so that the condition becomes false and the loop terminates. The variable which helps to finish the loop is called iteration variable. If there is no iteration variable, the loop will repeat forever, resulting in an infinite loop.

## Python “Infinite loops” and break

You can write an infinite loop on purpose and then use the break statement to jump out of the loop.

n = 10

while True :

print(n)

n = n + 1

print('STOP!!!')

If you mistakenly run the above code then you will see that it will run forever. While this is a dysfunctional infinite loop, we can still use this pattern to build useful loops as long as we carefully add code to the body of the loop to explicitly exit the loop using **break** when we have reached the exit condition. For example, suppose you want to take input from the user until they type done.  
You could write:

while True:

line = input('Enter "STOP" to stop the loop\n')

if line == 'STOP':

break

print(line)

print('STOP!')

Here, the loop condition is True, which is always true, so the loop runs repeatedly until it hits the break statement.

## Finishing iterations with continue in Python

Sometimes you are in an iteration of a loop and want to finish the current iteration and immediately jump to the next iteration. In that case you can use the continue statement to skip to the next iteration without finishing the body of the loop for the current iteration.

Here is an example of a loop that copies its input until the user types “STOP”, but treats lines that start with the hash character as lines not to be printed (kind of like Python comments).

while True:

line = input('> ')

if line[0] == '#':

continue

if line == 'done':

break

print(line)

print('Done!')

**Example 2:**

for i in "Make Me Analyst":

if i == "M":

continue

print(i)

print("STOP")

## Python for Loop

Sometimes You want to loop through a set of things such as a list of words, the lines in a file, or a list of numbers. When you have a list of things to loop through, you can construct a definite loop using a for statement. You call the while statement an indefinite loop because it simply loops until some condition becomes False, whereas the for loop is looping through a known set of items so it runs through as many iterations as there are items in the set. The syntax of a for loop is similar to the while loop in that there is a for statement and a loop body:

emp = ['Seba', 'Kattula', 'Mohan']

for e in emp:

print('Hello:', e)

print('Done!')

**Example 2:**

arr=[1,2,3,4,5]

for i in arr:

print(i)

## The range() function in Python

You can generate a sequence of numbers using range() function. range(5) will generate numbers from 0 to 4 (5 numbers). You can also define the start, stop and step size as **range(start,stop,step size)**. step size defaults to 1 if not provided. You can use this function in a list() to output all the items in it.

# Program to iterate through a list using indexing

arr = [1,2,3,4,5]

# iterate over the list using index

for i in range(len(arr)):

print(arr[i])

**Example 2:**

# Program to iterate through a list using indexing

arr = ["A","B","C","D"]

# iterate over the list using index

for i in range(len(arr)):

print(arr[i])

## Bonus Example: Counting and summing loops

count = 0

for i in [1,2,3,4,5]:

count = count + 1

print('Count: ', count)

## Bonus Example: Maximum and minimum loops

largest = None

print('Before:', largest)

for i in [3, 4, 12, 90, 44, 150]:

if largest is None or i > largest :

largest = i

print('Loop:', i, largest)

print('Largest:', largest)

smallest = None

print('Before:', smallest)

for i in [3, 4, 12, 90, 44, 150]:

if smallest is None or i < smallest :

smallest = i

print('Loop:', i, smallest)

print(Smallest:', smallest)

# Python Strings

A string is a sequence of characters. You can access the characters one at a time with the bracket operator. The expression in brackets is called an index. The index indicates which character in the sequence you want to print.

name="Mr. X"  
l = name[0]  
print(l)

## Getting the length of a string using len() function

print(len(name))

To get the last letter of a string, you might try this:

print(l[len(l)-1])

Alternatively, you can use negative indices, which count backward from the end of the string. The expression l[-1] yields the last letter, l[-2] yields the second to last, and so on.

print(name[-1])  
print(name[-2])

## String Methods

Here are some of the most common string methods. A method is like a function, but it runs "on" an object. If the variable s is a string, then the code s.lower() runs the lower() method on that string object and returns the result (this idea of a method running on an object is one of the basic ideas that make up Object Oriented Programming, OOP). Here are some of the most common string methods:

* s.lower(), s.upper() -- returns the lowercase or uppercase version of the string
* s.strip() -- returns a string with whitespace removed from the start and end
* s.isalpha()/s.isdigit()/s.isspace()... -- tests if all the string chars are in the various character classes
* s.startswith('other'), s.endswith('other') -- tests if the string starts or ends with the given other string
* s.find('other') -- searches for the given other string (not a regular expression) within s, and returns the first index where it begins or -1 if not found
* s.replace('old', 'new') -- returns a string where all occurrences of 'old' have been replaced by 'new'
* s.split('delim') -- returns a list of substrings separated by the given delimiter. The delimiter is not a regular expression, it's just text. 'aaa,bbb,ccc'.split(',') -> ['aaa', 'bbb', 'ccc']. As a convenient special case s.split() (with no arguments) splits on all whitespace chars.
* s.join(list) -- opposite of split(), joins the elements in the given list together using the string as the delimiter. e.g. '---'.join(['aaa', 'bbb', 'ccc']) -> aaa---bbb---ccc

### String %

Python has a printf()-like facility to put together a string. The % operator takes a printf-type format string on the left (%d int, %s string, %f/%g floating point), and the matching values in a tuple on the right (a tuple is made of values separated by commas, typically grouped inside parentheses):

# % operator  
text = "%d little pigs come out, or I'll %s, and I'll %s, and I'll blow your %s down." % (3, 'huff', 'puff', 'house')

# Add parentheses to make the long line work:  
text = (  
  "%d little pigs come out, or I'll %s, and I'll %s, and I'll blow your %s down."  
  % (3, 'huff', 'puff', 'house'))

# Split the line into chunks, which are concatenated automatically by Python  
  text = (  
    "%d little pigs come out, "  
    "or I'll %s, and I'll %s, "  
    "and I'll blow your %s down."  
    % (3, 'huff', 'puff', 'house'))

## Single, Double and Triple Quotes

### Single Quotes

Generally use single quotes for string literals.

word = 'Ask?'

print(word)

sentence = 'Python Programming'

print(sentence)

name = '"Hi" ABC'

print(name)

congrat = 'We congrat's you.'

print(congrat)

**Output**

|  |
| --- |
| Ask?  Python Programming  Hi ABC  Invalid Syntax |

### Double Quotes

Use Double Quotes for string representation.

wish = "Hello World!"

print(wish)

hey = "AskPython says "Hi""

print(hey)

famous ="'Taj Mahal' is in Agra."

print(famous)

**Output**

|  |
| --- |
| Hello World!  Invalid Syntax  'Taj Mahal' is in Agra. |

### Triple Quotes

What if you have to use strings that may include both single and double quotes? For this, Python allows you to use triple quotes. A simple example for the same is shown below. Triple quotes also allow you to add multi-line strings to Python variables instead of being limited to single lines.

**Example of triple quotes**

|  |
| --- |
| sentence1 = '''He asked, "did you speak with him?"'''  print(sentence1)  sentence2 = '''"That's great", she said.'''  print(sentence2) |

Output:

|  |
| --- |
| He asked, "did you speak with him?"  "That's great", she said. |

## Traversing a  string with a loop

One way to write a traversal is with a **while loop**:

i = 0

while i < len(name):

letter = name[i]

print(letter)

i = i + 1

One way to write a traversal is with a **for loop**:

for char in name:

print(char)

## String slices

A segment of a string is called a slice. Selecting a slice is similar to selecting a character:

s = 'Make Me Analyst'

print(s[0:4])

print(s[8:len(s)])

print(s[:4])

print(s[:len(s)])

## Strings are immutable

Strings are immutable in Python. It means you can’t change an existing string. Let’s try the below example:

str = 'Make Me Analyst'  
str[0]='T'

If you run the above code, you will get an error like this: TypeError: ‘str’ object does not support item assignment  
The reason for the error is that strings are immutable. The best you can do is create a new string that is a variation on the original:

str = 'Make Me Analyst'  
new\_str='Hi! '+ str[8:len(str)]

This example concatenates a new first word onto a slice of the string and it has no effect on the original string.

## Looping and counting

The following program counts the number of times the letter “M” appears in a string:

str = 'Make Me Analyst'

count = 0

for letter in str:

if letter == 'M':

count = count + 1

print(count)

## The in operator in Python

str = 'Make Me Analyst'

a='Analyst' in str

print(a)

b='x' in str

print(b)

## String comparison

The comparison operators work on strings.  Following code checks if two strings are equal:

word='Analyst'

if word=='Analyst':

print('Both are same!')

Some comparison operations are useful for putting words in alphabetical order:

word='Orange'

if word < 'Apple':

print('Your word, ' + word + ', comes before Apple')

elif word > 'Apple':

print('Your word, ' + word + ', comes after Apple.')

else:

print('All right, Orange!!!')

**Note:**Python does not handle uppercase and lowercase letters the same way that people do. All the uppercase letters come before all the lowercase letters

# Python Lists

Like a [**string**](http://makemeanalyst.com/python-programming/strings/), a list is a sequence of values. In a string, the values are characters; in a list, they can be any type. The values in list are called elements or sometimes items.

## How to create a list?

There are several ways to create a new list; the simplest is to enclose the elements in square brackets ([ and ]):

This is an example of a list of five integers.

numbers = [10, 20, 30, 40, 50]  
print(numbers)

Here is an empty list.

empty = []

Below example is a list of three strings.

food = ['Hot dog','Sandwich', 'Hamburger']  
print(food)

You can also create a list with mixed datatypes

mixed\_list = [1, "Python", 1.5]  
print(mixed\_list)

The following list contains a string, a float, an integer, and another list:

nested\_list = ['Python', 2.0, 5, [10, 20]]  
print(nested\_list)

## Lists are mutable

The syntax for accessing the elements of a list is the same as for accessing the characters of a [**string**](http://makemeanalyst.com/python-programming/strings/): the bracket operator. The expression inside the brackets specifies the index. Remember that the indices start at 0:

food = ['Hot dog','Sandwich', 'Hamburger']  
print(food[0])  
print(food[1])

Unlike strings, lists are mutable because you can change the order of items in a list or reassign an item in a list. When the bracket operator appears on the left side of an assignment, it identifies the element of the list that will be assigned.

numbers = [10, 20]  
numbers[0] = 100  
numbers[1] = 200  
print(numbers)

The in operator also works on lists.

food = ['Hot dog','Sandwich', 'Hamburger']  
print('Hot dog' in food)  
print('French fries' in food)

## How to access elements from a list?

You have already seen in the above example that we can use the index operator [] to access an item in a list. Index starts from 0.  So, a list having 3 elements will have index from 0 to 2.

## List Index

food=['Hot dog','Sandwich', 'Hamburger']  
print(food[0])  
print(food[1])

If you try to read or write an element that does not exist, you get an IndexError

print(food[3])

## Negative indexing

If an index has a negative value, it counts backward from the end of the list.

food=['Hot dog','Sandwich', 'Hamburger']  
print(food[-1])  
print(food[-2])

## Traversing a list

The most common way to traverse the elements of a list is with a for loop.

food=['Hot dog','Sandwich', 'Hamburger']

for i in food:

print(i)

Above method works well if you only need to read the elements of the list. But if you want to write or update the elements, you need the indices. A common way to traverse the list is to combine the functions range and len:

food=['Hot dog','Sandwich', 'Hamburger']

for i in range(len(food)):

print(food[i])

## List operations

The + operator concatenates lists:

a = [1, 2, 3]  
b = [4, 5, 6]  
c = a + b  
print(c)

Similarly, the operator repeats a list a given number of times:

a=[0]\*4  
print(a)  
b=[1,2,3]\*3  
print(b)

The first example repeats four times. The second example repeats the list three times.

## How to slice lists in Python?

The slice operator also works on lists. You can access a range of items in a list by using the slicing operator (colon).

l = ['make','me', 'analyst']  
# get elements 2nd to 3rd  
print(l[1:3])  
# get elements beginning to 2nd  
print(l[:-1])  
# get elements 2nd to end  
print(l[1:])  
# elements beginning to end  
print(l[:])

Since lists are mutable, it is often useful to make a copy before performing operations  
that fold, spindle, or mutilate lists.

A slice operator on the left side of an assignment can update multiple elements:

t = ['a', 'b', 'c', 'd', 'e', 'f']  
t[1:3] = ['x', 'y']  
print(t)

## List methods

Python provides methods that operate on lists. For example, **append** adds a new element to the end of a list:

x = ['a', 'b', 'c']  
x.append('d')  
print(x)

**extend** takes a list as an argument and appends all of the elements:

x1 = ['a', 'b', 'c']  
x2 = ['d', 'e']  
x1.extend(x2)  
print(x1)

This example leaves x2 unmodified.

**sort** arranges the elements of the list from low to high:

t = ['d', 'c', 'e', 'b', 'a']  
t.sort()  
print(t)

Most list methods are void; they modify the list and return None. If you accidentally write t = t.sort(), you will be disappointed with the result.

## How to delete or remove elements from a list?

### ***pop*** *operator*

There are several ways to delete elements from a list. If you know the index of the element you want, you can use **pop**:

t = ['a', 'b', 'c']  
x = t.pop(1)  
print(t)  
print(x)

### ***del*** operator

pop modifies the list and returns the element that was removed. If you don’t provide an index, it deletes and returns the last element.

If you don’t need the removed value, you can use the **del** operator:

t = ['a', 'b', 'c']  
del t[1]  
print(t)

### remove() Function

If you know the element you want to remove (but not the index), you can use **remove**:

t = ['a', 'b', 'c']  
t.remove('b')  
print(t)

The return value from remove is None. To remove more than one element, you can use del with a slice index:

t = ['a', 'b', 'c', 'd', 'e', 'f']  
del t[1:5]  
print(t)

As usual, the slice selects all the elements up to, but not including, the second index.

## Lists and functions

There are a number of built-in functions that can be used on lists that allow you to quickly look through a list without writing your own loops:

nums = [3, 4, 5, 6, 7, 8]  
print(len(nums))  
print(max(nums))  
print(min(nums))  
print(sum(nums))  
print(sum(nums)/len(nums))

The ***sum()*** function only works when the list elements are numbers. The other functions (***max()***, **len()**, etc.) work with lists of strings and other types that can be comparable.

You could rewrite an earlier program that computed the average of a list of numbers entered by the user using a list. First, the program to compute an average without a list:

total = 0

count = 0

while (True):

inp = input('Enter a number: ')

if inp == 'done': break

value = float(inp)

total = total + value

count = count + 1

average = total / count

print('Average:', average)

In this program, you have count and total variables to keep the number and running total of the user’s numbers as we repeatedly prompt the user for a number. You could simply remember each number as the user entered it and use built-in functions to compute the sum and count at the end.

numlist = list()

while (True):

inp = input('Enter a number: ')

if inp == 'done': break

value = float(inp)

numlist.append(value)

average = sum(numlist) / len(numlist)

print('Average:', average)

We make an empty list before the loop starts, and then each time we have a number, we append it to the list. At the end of the program, we simply compute the sum of the numbers in the list and divide it by the count of the numbers in the list to come up with the average.

## Lists and strings

A string is a sequence of characters and a list is a sequence of values, but a list of characters is not the same as a string. To convert from a string to a list of characters, you can use list:

l="Make Me Aanlyst"  
t = list(l)  
print(t)

The list function breaks a string into individual letters. If you want to break a string into words, you can use the split method:

s = 'Make Me Aanlyst'  
t = s.split()  
print(t)

Once you have used split to break the string into a list of words, you can use the index operator (square bracket) to look at a particular word in the list. You can call split with an optional argument called a delimiter that specifies which characters to use as word boundaries. The following example uses a hyphen.

s = 'make-me-analyst'  
delimiter = '-'  
s.split(delimiter)

***join*** is the inverse of split. It takes a list of strings and concatenates the elements. join is a string method, so you have to invoke it on the delimiter and pass the list as a parameter:

t = ['Make', 'Me', 'Analyst']  
delimiter = ' '  
delimiter.join(t)

## List arguments

When you pass a list to a function, the function gets a reference to the list. If the function modifies a list parameter, the caller sees the change. For example, delete\_head removes the first element from a list:

def delete\_head(t):

del t[0]

Here’s how it is used:

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

delete\_head(letters)

print(letters)

The parameter t and the variable letters are aliases for the same object. It is important to distinguish between operations that modify lists and operations that create new lists. For example, the append method modifies a list, but the + operator creates a new list:

t1 = [1, 2]

t2 = t1.append(3)

print(t1)

print(t2)

t3 = t1 + [3]

print(t3)

t2 is t3

# Python Dictionaries

A dictionary is like a list, but more general. In a list, the index positions have to be integers; in a dictionary, the indices can be (almost) any type. Dictionary as a mapping between a set of indices (which are called keys) and a set of values. Each key maps to a value.

The association of a key and a value is called a key-value pair or sometimes an item. As an example, we’ll build a dictionary that maps from English to German words, so the keys and the values are all strings.

## How to create a dictionary?

The function dict creates a new dictionary with no items. Because dict is the name of a built-in function, you should avoid using it as a variable name.

eng2gr = dict()

print(eng2gr)

The curly brackets, {}, represent an empty dictionary. To add items to the dictionary, you can use square brackets:

eng2gr['one'] = 'eins'

This line creates an item that maps from the key ’one’ to the value “eins”. If you print the dictionary again, you see a key-value pair with a colon between the key and value:

print(eng2gr)

This output format is also an input format. For example, you can create a new dictionary with three items. But if you print eng2gr, you might be surprised:

eng2gr = {'one': 'eins', 'two': 'zwei', 'three': 'drei'}

print(eng2gr)

## How to access elements from a dictionary?

The order of the key-value pairs is not the same. In fact, if you type the same example on your computer, you might get a different result. In general, the order of items in a dictionary is unpredictable. But that’s not a problem because the elements of a dictionary are never indexed with integer indices. Instead, you use the keys to look up the corresponding values:

eng2gr = {'one': 'eins', 'two': 'zwei', 'three': 'drei'}

print(eng2gr['two'])

The key ’two’ always maps to the value “zwei” so the order of the items doesn’t matter.If the key isn’t in the dictionary, you get an exception:

>>> print(eng2gr['four'])  
KeyError: 'four'

While indexing is used with other container types to access values, dictionary uses keys. Key can be used either inside square brackets or with the get() method.

The difference while using get() is that it returns None instead of KeyError, if the key is not found.

print(eng2gr.get('two'))

print(eng2gr.get('three'))

The len function works on dictionaries; it returns the number of key-value pairs:

len(eng2gr)

## How to change or add elements in a dictionary?

Dictionaries are mutable. We can add new items or change the value of existing items using assignment operator. If the key is already present, value gets updated, else a new key: value pair is added to the dictionary.

eng2gr = {'one': 'eins', 'two': 'zwei', 'three': 'drei'}

eng2gr['four'] = 'four' #Add Element

print(eng2gr)

eng2gr['four'] = 'vier'  #Update Element

print(eng2gr)

## Dictionary Membership Test

The in operator works on dictionaries; it tells you whether something appears as a key in the dictionary.

>>> 'one' in eng2gr  
True  
>>> 'eins' in eng2gr  
False

To see whether something appears as a value in a dictionary, you can use the method values, which returns the values as a list, and then use the**in** operator:

vals = list(eng2gr.values())

>>>'eins' in vals

True

The in operator uses different algorithms for lists and dictionaries. For lists, it uses a linear search algorithm. As the list gets longer, the search time gets longer in direct proportion to the length of the list. For dictionaries, Python uses an algorithm called a hash table that has a remarkable property: the ***in*** operator takes about the same amount of time no matter how many items there are in a  
dictionary. I won’t explain why hash functions are so magical, but you can read more about it at [wikipedia.org/wiki/Hash\_table](http://wikipedia.org/wiki/Hash_table).

## How to delete or remove elements from a dictionary?

You can remove a particular item in a dictionary by using the method pop(). This method removes as item with the provided key and returns the value.

The method, popitem() can be used to remove and return an arbitrary item (key, value) form the dictionary. All the items can be removed at once using the clear() method.

You can also use the del keyword to remove individual items or the entire dictionary itself.

eng2gr = {'one': 'eins', 'two': 'zwei', 'three': 'drei', 'four':'vier'}

# remove a particular item

print(eng2gr.pop('four'))

print(eng2gr)

# remove an arbitrary item

print(eng2gr.popitem())

print(eng2gr)

# delete a particular item

del eng2gr['one']

print(eng2gr)

# remove all items

eng2gr.clear()

## Python Dictionary Methods

Methods that are available with dictionary are tabulated below. Some of them have already been used in the above examples.

|  |  |
| --- | --- |
| Python Dictionary Methods | |
| **Method** | **Description** |
| clear() | Remove all items form the dictionary. |
| copy() | Return a shallow copy of the dictionary. |
| fromkeys(seq[, v]) | Return a new dictionary with keys from seq and value equal to v(defaults to None). |
| get(key[,d]) | Return the value of key. If key doesnot exit, return d (defaults to None). |
| items() | Return a new view of the dictionary’s items (key, value). |
| keys() | Return a new view of the dictionary’s keys. |
| pop(key[,d]) | Remove the item with key and return its value or d if key is not found. If d is not provided and key is not found, raises KeyError. |
| popitem() | Remove and return an arbitary item (key, value). Raises KeyError if the dictionary is empty. |
| setdefault(key[,d]) | If key is in the dictionary, return its value. If not, insert key with a value of d and return d (defaults to None). |
| update([other]) | Update the dictionary with the key/value pairs from other, overwriting existing keys. |
| values() | Return a new view of the dictionary’s values |

Here are a few example use of these methods.

fruits = {}.fromkeys(['Orange','Apple','Banana'], 0)

print(fruits)

for item in fruits.items():

print(item)

list(sorted(fruits.keys()))

# Object Oriented Programming in Python 3

**Object-oriented programming** (OOP) is a method of structuring a program by bundling related properties and behaviors into individual **objects**.

An object contains data, like the raw or preprocessed materials at each step on an assembly line, and behavior, like the action each assembly line component performs.

Object-oriented programming is a programming paradigm that provides a means of structuring programs so that properties and behaviors are bundled into individual **objects**.

For instance, an object could represent a person with **properties** like a name, age, and address and **behaviors** such as walking, talking, breathing, and running. Or it could represent an email with properties like a recipient list, subject, and body and behaviors like adding attachments and sending.

Another common programming paradigm is **procedural programming**, which structures a program like a recipe in that it provides a set of steps, in the form of functions and code blocks, that flow sequentially in order to complete a task.

The key takeaway is that objects are at the center of object-oriented programming in Python, not only representing the data, as in procedural programming, but in the overall structure of the program as well.

## Define a Class in Python

Primitive data structures—like numbers, strings, and lists—are designed to represent simple pieces of information, such as the cost of an apple, the name of a poem, or your favorite colors, respectively. What if you want to represent something more complex?

For example, let’s say you want to track employees in an organization. You need to store some basic information about each employee, such as their name, age, position, and the year they started working.

One way to do this is to represent each employee as a list:

kirk = ["James Kirk", 34, "Captain", 2265]

spock = ["Spock", 35, "Science Officer", 2254]

mccoy = ["Leonard McCoy", "Chief Medical Officer", 2266]

There are a number of issues with this approach.

First, it can make larger code files more difficult to manage. If you reference kirk[0] several lines away from where the kirk list is declared, will you remember that the element with index 0 is the employee’s name?

Second, it can introduce errors if not every employee has the same number of elements in the list. In the mccoy list above, the age is missing, so mccoy[1] will return "Chief Medical Officer" instead of Dr. McCoy’s age.

A great way to make this type of code more manageable and more maintainable is to use **classes**.

### Classes vs Instances

Classes are used to create user-defined data structures. Classes define functions called **methods**, which identify the behaviors and actions that an object created from the class can perform with its data.

A class is a blueprint for how something should be defined. It doesn’t actually contain any data. The Dog class specifies that a name and an age are necessary for defining a dog, but it doesn’t contain the name or age of any specific dog.

While the class is the blueprint, an **instance** is an object that is built from a class and contains real data. An instance of the Dog class is not a blueprint anymore. It’s an actual dog with a name, like Miles, who’s four years old.

### How to Define a Class

All class definitions start with the class keyword, which is followed by the name of the class and a colon. Any code that is indented below the class definition is considered part of the class’s body.

Here’s an example of a Dog class:

class Dog:

pass

The body of the Dog class consists of a single statement: the pass keyword. pass is often used as a placeholder indicating where code will eventually go. It allows you to run this code without Python throwing an error.

**Note:** Python class names are written in CapitalizedWords notation by convention. For example, a class for a specific breed of dog like the Jack Russell Terrier would be written as JackRussellTerrier.

The Dog class isn’t very interesting right now, so let’s spruce it up a bit by defining some properties that all Dog objects should have. There are a number of properties that we can choose from, including name, age, coat color, and breed. To keep things simple, we’ll just use name and age.

The properties that all Dog objects must have are defined in a method called .\_\_init\_\_(). Every time a new Dog object is created, .\_\_init\_\_() sets the initial **state** of the object by assigning the values of the object’s properties. That is, .\_\_init\_\_() initializes each new instance of the class.

You can give .\_\_init\_\_() any number of parameters, but the first parameter will always be a [variable](https://realpython.com/python-variables/) called self. When a new class instance is created, the instance is automatically passed to the self parameter in .\_\_init\_\_() so that new **attributes** can be defined on the object.

Let’s update the Dog class with an .\_\_init\_\_() method that creates .name and .age attributes:

class Dog:

def \_\_init\_\_(self, name, age):

self.name = name

self.age = age

Notice that the .\_\_init\_\_() method’s signature is indented four spaces. The body of the method is indented by eight spaces. This indentation is vitally important. It tells Python that the .\_\_init\_\_() method belongs to the Dog class.

In the body of .\_\_init\_\_(), there are two statements using the self variable:

1. **self.name = name** creates an attribute called name and assigns to it the value of the name parameter.
2. **self.age = age** creates an attribute called age and assigns to it the value of the age parameter.

Attributes created in .\_\_init\_\_() are called **instance attributes**. An instance attribute’s value is specific to a particular instance of the class. All Dog objects have a name and an age, but the values for the name and age attributes will vary depending on the Dog instance.

On the other hand, **class attributes** are attributes that have the same value for all class instances. You can define a class attribute by assigning a value to a [variable](https://realpython.com/python-variables/) name outside of .\_\_init\_\_().

For example, the following Dog class has a class attribute called species with the value "Canis familiaris":

class Dog:

# Class attribute

species = "Canis familiaris"

def \_\_init\_\_(self, name, age):

self.name = name

self.age = age

Class attributes are defined directly beneath the first line of the class name and are indented by four spaces. They must always be assigned an initial value. When an instance of the class is created, class attributes are automatically created and assigned to their initial values.

Use class attributes to define properties that should have the same value for every class instance. Use instance attributes for properties that vary from one instance to another.

Now that we have a Dog class, let’s create some dogs!

## Instantiate an Object in Python

Open Python’s interactive window and type the following:

>>> class Dog:

... pass

This creates a new Dog class with no attributes or methods.

Creating a new object from a class is called **instantiating** an object. You can instantiate a new Dog object by typing the name of the class, followed by opening and closing parentheses:

>>> Dog()

<\_\_main\_\_.Dog object at 0x106702d30>

You now have a new Dog object at 0x106702d30. This funny-looking string of letters and numbers is a **memory address** that indicates where the Dog object is stored in your computer’s memory. Note that the address you see on your screen will be different.

Now instantiate a second Dog object:

>>> Dog()

<\_\_main\_\_.Dog object at 0x0004ccc90>

The new Dog instance is located at a different memory address. That’s because it’s an entirely new instance and is completely unique from the first Dog object that you instantiated.

To see this another way, type the following:

>>> a = Dog()

>>> b = Dog()

>>> a == b

False

In this code, you create two new Dog objects and assign them to the variables a and b. When you compare a and b using the == operator, the result is False. Even though a and b are both instances of the Dog class, they represent two distinct objects in memory.

### Class and Instance Attributes

Now create a new Dog class with a class attribute called .species and two instance attributes called .name and .age:

>>> class Dog:

... species = "Canis familiaris"

... def \_\_init\_\_(self, name, age):

... self.name = name

... self.age = age

To instantiate objects of this Dog class, you need to provide values for the name and age. If you don’t, then Python raises a TypeError:

>>> Dog()

Traceback (most recent call last):

File "<pyshell#6>", line 1, in <module>

Dog()

TypeError: \_\_init\_\_() missing 2 required positional arguments: 'name' and 'age'

To pass arguments to the name and age parameters, put values into the parentheses after the class name:

>>> buddy = Dog("Buddy", 9)

>>> miles = Dog("Miles", 4)

This creates two new Dog instances—one for a nine-year-old dog named Buddy and one for a four-year-old dog named Miles.

The Dog class’s .\_\_init\_\_() method has three parameters, so why are only two arguments passed to it in the example?

When you instantiate a Dog object, Python creates a new instance and passes it to the first parameter of .\_\_init\_\_(). This essentially removes the self parameter, so you only need to worry about the name and age parameters.

After you create the Dog instances, you can access their instance attributes using **dot notation**:

>>> buddy.name

'Buddy'

>>> buddy.age

9

>>> miles.name

'Miles'

>>> miles.age

4

You can access class attributes the same way:

>>> buddy.species

'Canis familiaris'

One of the biggest advantages of using classes to organize data is that instances are guaranteed to have the attributes you expect. All Dog instances have .species, .name, and .age attributes, so you can use those attributes with confidence knowing that they will always return a value.

Although the attributes are guaranteed to exist, their values can be changed dynamically:

>>> buddy.age = 10

>>> buddy.age

10

>>> miles.species = "Felis silvestris"

>>> miles.species

'Felis silvestris'

In this example, you change the .age attribute of the buddy object to 10. Then you change the .species attribute of the miles object to "Felis silvestris", which is a species of cat. That makes Miles a pretty strange dog, but it is valid Python!

The key takeaway here is that custom objects are mutable by default. An object is mutable if it can be altered dynamically. For example, lists and dictionaries are mutable, but strings and tuples are immutable.

## Instance Methods

**Instance methods** are functions that are defined inside a class and can only be called from an instance of that class. Just like .\_\_init\_\_(), an instance method’s first parameter is always self.

Open a new Python interactive window and type in the following Dog class:

class Dog:

species = "Canis familiaris"

def \_\_init\_\_(self, name, age):

self.name = name

self.age = age

# Instance method

def description(self):

return f"{self.name} is {self.age} years old"

# Another instance method

def speak(self, sound):

return f"{self.name} says {sound}"

This Dog class has two instance methods:

1. **.description()** returns a string displaying the name and age of the dog.
2. **.speak()** has one parameter called sound and returns a string containing the dog’s name and the sound the dog makes.

Save the modified Dog class to a file called dog.py and run it in VS Code. Then open the interactive window and type the following to see your instance methods in action:

>>> miles = Dog("Miles", 4)

>>> miles.description()

'Miles is 4 years old'

>>> miles.speak("Woof Woof")

'Miles says Woof Woof'

>>> miles.speak("Bow Wow")

'Miles says Bow Wow'

In the above Dog class, .description() returns a string containing information about the Dog instance miles. When writing your own classes, it’s a good idea to have a method that returns a string containing useful information about an instance of the class. However, .description() isn’t the most Pythonic way of doing this.

When you create a list object, you can use print() to display a string that looks like the list:

>>> names = ["Fletcher", "David", "Dan"]

>>> print(names)

['Fletcher', 'David', 'Dan']

Let’s see what happens when you print() the miles object:

>>> print(miles)

<\_\_main\_\_.Dog object at 0x00aeff70>

When you print(miles), you get a cryptic looking message telling you that miles is a Dog object at the memory address 0x00aeff70. This message isn’t very helpful. You can change what gets printed by defining a special instance method called .\_\_str\_\_().

In the editor window, change the name of the Dog class’ .description() method to .\_\_str\_\_():

class Dog:

# Leave other parts of Dog class as-is

# Replace .description() with \_\_str\_\_()

def \_\_str\_\_(self):

return f"{self.name} is {self.age} years old"

Save the file and run it. Now, when you print(miles), you get a much friendlier output:

>>> miles = Dog("Miles", 4)

>>> print(miles)

'Miles is 4 years old'

Methods like .\_\_init\_\_() and .\_\_str\_\_() are called **dunder methods** because they begin and end with double underscores. There are many dunder methods that you can use to customize classes in Python. Although too advanced a topic for a beginning Python book, understanding dunder methods is an important part of mastering object-oriented programming in Python.

## Inherit From Other Classes in Python

Inheritance is the process by which one class takes on the attributes and methods of another. Newly formed classes are called **child classes**, and the classes that child classes are derived from are called **parent classes**.

Child classes can override or extend the attributes and methods of parent classes. In other words, child classes inherit all of the parent’s attributes and methods but can also specify attributes and methods that are unique to themselves.

Although the analogy isn’t perfect, you can think of object inheritance sort of like genetic inheritance.

You may have inherited your hair color from your mother. It’s an attribute you were born with. Let’s say you decide to color your hair purple. Assuming your mother doesn’t have purple hair, you’ve just **overridden** the hair color attribute that you inherited from your mom.

You also inherit, in a sense, your language from your parents. If your parents speak English, then you’ll also speak English. Now imagine you decide to learn a second language, like German. In this case you’ve **extended** your attributes because you’ve added an attribute that your parents don’t have.

### Dog Park Example

Pretend for a moment that you’re at a dog park. There are many dogs of different breeds at the park, all engaging in various dog behaviors.

Suppose now that you want to model the dog park with Python classes. The Dog class that you wrote in the previous section can distinguish dogs by name and age but not by breed.

You could modify the Dog class in the editor window by adding a .breed attribute:

class Dog:

species = "Canis familiaris"

def \_\_init\_\_(self, name, age, breed):

self.name = name

self.age = age

self.breed = breed

The instance methods defined earlier are omitted here because they aren’t important for this discussion.

Press F5 to save the file. Now you can model the dog park by instantiating a bunch of different dogs in the interactive window:

>>> miles = Dog("Miles", 4, "Jack Russell Terrier")

>>> buddy = Dog("Buddy", 9, "Dachshund")

>>> jack = Dog("Jack", 3, "Bulldog")

>>> jim = Dog("Jim", 5, "Bulldog")

Each breed of dog has slightly different behaviors. For example, bulldogs have a low bark that sounds like *woof*, but dachshunds have a higher-pitched bark that sounds more like *yap*.

Using just the Dog class, you must supply a string for the sound argument of .speak() every time you call it on a Dog instance:

>>> buddy.speak("Yap")

'Buddy says Yap'

>>> jim.speak("Woof")

'Jim says Woof'

>>> jack.speak("Woof")

'Jack says Woof'

Passing a string to every call to .speak() is repetitive and inconvenient. Moreover, the string representing the sound that each Dog instance makes should be determined by its .breed attribute, but here you have to manually pass the correct string to .speak() every time it’s called.

You can simplify the experience of working with the Dog class by creating a child class for each breed of dog. This allows you to extend the functionality that each child class inherits, including specifying a default argument for .speak().

### Parent Classes vs Child Classes

Let’s create a child class for each of the three breeds mentioned above: Jack Russell Terrier, Dachshund, and Bulldog.

For reference, here’s the full definition of the Dog class:

class Dog:

species = "Canis familiaris"

def \_\_init\_\_(self, name, age):

self.name = name

self.age = age

def \_\_str\_\_(self):

return f"{self.name} is {self.age} years old"

def speak(self, sound):

return f"{self.name} says {sound}"

Remember, to create a child class, you create new class with its own name and then put the name of the parent class in parentheses. Add the following to the dog.py file to create three new child classes of the Dog class:

class JackRussellTerrier(Dog):

pass

class Dachshund(Dog):

pass

class Bulldog(Dog):

pass

Press F5 to save and run the file. With the child classes defined, you can now instantiate some dogs of specific breeds in the interactive window:

>>> miles = JackRussellTerrier("Miles", 4)

>>> buddy = Dachshund("Buddy", 9)

>>> jack = Bulldog("Jack", 3)

>>> jim = Bulldog("Jim", 5)

Instances of child classes inherit all of the attributes and methods of the parent class:

>>> miles.species

'Canis familiaris'

>>> buddy.name

'Buddy'

>>> print(jack)

Jack is 3 years old

>>> jim.speak("Woof")

'Jim says Woof'

To determine which class a given object belongs to, you can use the built-in type():

>>> type(miles)

<class '\_\_main\_\_.JackRussellTerrier'>

What if you want to determine if miles is also an instance of the Dog class? You can do this with the built-in isinstance():

>>> isinstance(miles, Dog)

True

Notice that isinstance() takes two arguments, an object and a class. In the example above, isinstance() checks if miles is an instance of the Dog class and returns True.

The miles, buddy, jack, and jim objects are all Dog instances, but miles is not a Bulldog instance, and jack is not a Dachshund instance:

>>> isinstance(miles, Bulldog)

False

>>> isinstance(jack, Dachshund)

False

More generally, all objects created from a child class are instances of the parent class, although they may not be instances of other child classes.

Now that you’ve created child classes for some different breeds of dogs, let’s give each breed its own sound.

### Extend the Functionality of a Parent Class

Since different breeds of dogs have slightly different barks, you want to provide a default value for the sound argument of their respective .speak() methods. To do this, you need to override .speak() in the class definition for each breed.

To override a method defined on the parent class, you define a method with the same name on the child class. Here’s what that looks like for the JackRussellTerrier class:

class JackRussellTerrier(Dog):

def speak(self, sound="Arf"):

return f"{self.name} says {sound}"

Now .speak() is defined on the JackRussellTerrier class with the default argument for sound set to "Arf".

Update dog.py with the new JackRussellTerrier class and press F5 to save and run the file. You can now call .speak() on a JackRussellTerrier instance without passing an argument to sound:

>>> miles = JackRussellTerrier("Miles", 4)

>>> miles.speak()

'Miles says Arf'

Sometimes dogs make different barks, so if Miles gets angry and growls, you can still call .speak() with a different sound:

>>> miles.speak("Grrr")

'Miles says Grrr'

One thing to keep in mind about class inheritance is that changes to the parent class automatically propagate to child classes. This occurs as long as the attribute or method being changed isn’t overridden in the child class.

For example, in the editor window, change the string returned by .speak() in the Dog class:

class Dog:

# Leave other attributes and methods as they are

# Change the string returned by .speak()

def speak(self, sound):

return f"{self.name} barks: {sound}"

Save the file and press F5. Now, when you create a new Bulldog instance named jim, jim.speak() returns the new string:

>>>

>>> jim = Bulldog("Jim", 5)

>>> jim.speak("Woof")

'Jim barks: Woof'

However, calling .speak() on a JackRussellTerrier instance won’t show the new style of output:

>>> miles = JackRussellTerrier("Miles", 4)

>>> miles.speak()

'Miles says Arf'

Sometimes it makes sense to completely override a method from a parent class. But in this instance, we don’t want the JackRussellTerrier class to lose any changes that might be made to the formatting of the output string of Dog.speak().

To do this, you still need to define a .speak() method on the child JackRussellTerrier class. But instead of explicitly defining the output string, you need to call the Dog class’s .speak() *inside* of the child class’s .speak() using the same arguments that you passed to JackRussellTerrier.speak().

You can access the parent class from inside a method of a child class by using super():

class JackRussellTerrier(Dog):

def speak(self, sound="Arf"):

return super().speak(sound)

When you call super().speak(sound) inside JackRussellTerrier, Python searches the parent class, Dog, for a .speak() method and calls it with the variable sound.

Update dog.py with the new JackRussellTerrier class. Save the file and press F5 so you can test it in the interactive window:

>>> miles = JackRussellTerrier("Miles", 4)

>>> miles.speak()

'Miles barks: Arf'

Now when you call miles.speak(), you’ll see output reflecting the new formatting in the Dog class.

# Namespaces

We have seen the importance of **objects** in Python. Objects are everywhere! Virtually everything that your Python program creates or acts on is an object.

An **assignment statement** creates a **symbolic name** that you can use to reference an object. The statement x = 'foo' creates a symbolic name x that refers to the string object 'foo'.

In a program of any complexity, you’ll create hundreds or thousands of such names, each pointing to a specific object. How does Python keep track of all these names so that they don’t interfere with one another?

## Namespaces in Python

A namespace is a collection of currently defined symbolic names along with information about the object that each name references. You can think of a namespace as a dictionary in which the keys are the object names and the values are the objects themselves. Each key-value pair maps a name to its corresponding object.

Python uses them extensively. In a Python program, there are four types of namespaces:

1. Built-In
2. Global
3. Enclosing
4. Local

These have differing lifetimes. As Python executes a program, it creates namespaces as necessary and deletes them when they’re no longer needed. Typically, many namespaces will exist at any given time.

### The Built-In Namespace

The **built-in namespace** contains the names of all of Python’s built-in objects. These are available at all times when Python is running. You can list the objects in the built-in namespace with the following command:

>>> dir(\_\_builtins\_\_)

['ArithmeticError', 'AssertionError', 'AttributeError',

'BaseException','BlockingIOError', 'BrokenPipeError', 'BufferError',

'BytesWarning', 'ChildProcessError', 'ConnectionAbortedError',

'ConnectionError', 'ConnectionRefusedError', 'ConnectionResetError',

'DeprecationWarning', 'EOFError', 'Ellipsis', 'EnvironmentError',

'Exception', 'False', 'FileExistsError', 'FileNotFoundError',

'FloatingPointError', 'FutureWarning', 'GeneratorExit', 'IOError',

'ImportError', 'ImportWarning', 'IndentationError', 'IndexError',

'InterruptedError', 'IsADirectoryError', 'KeyError', 'KeyboardInterrupt',

'LookupError', 'MemoryError', 'ModuleNotFoundError', 'NameError', 'None',

'NotADirectoryError', 'NotImplemented', 'NotImplementedError', 'OSError',

'OverflowError', 'PendingDeprecationWarning', 'PermissionError',

'ProcessLookupError', 'RecursionError', 'ReferenceError', 'ResourceWarning',

'RuntimeError', 'RuntimeWarning', 'StopAsyncIteration', 'StopIteration',

'SyntaxError', 'SyntaxWarning', 'SystemError', 'SystemExit', 'TabError',

'TimeoutError', 'True', 'TypeError', 'UnboundLocalError',

'UnicodeDecodeError', 'UnicodeEncodeError', 'UnicodeError',

'UnicodeTranslateError', 'UnicodeWarning', 'UserWarning', 'ValueError',

'Warning', 'ZeroDivisionError', '\_', '\_\_build\_class\_\_', '\_\_debug\_\_',

'\_\_doc\_\_', '\_\_import\_\_', '\_\_loader\_\_', '\_\_name\_\_', '\_\_package\_\_',

'\_\_spec\_\_', 'abs', 'all', 'any', 'ascii', 'bin', 'bool', 'bytearray',

'bytes', 'callable', 'chr', 'classmethod', 'compile', 'complex',

'copyright', 'credits', 'delattr', 'dict', 'dir', 'divmod', 'enumerate',

'eval', 'exec', 'exit', 'filter', 'float', 'format', 'frozenset',

'getattr', 'globals', 'hasattr', 'hash', 'help', 'hex', 'id', 'input',

'int', 'isinstance', 'issubclass', 'iter', 'len', 'license', 'list',

'locals', 'map', 'max', 'memoryview', 'min', 'next', 'object', 'oct',

'open', 'ord', 'pow', 'print', 'property', 'quit', 'range', 'repr',

'reversed', 'round', 'set', 'setattr', 'slice', 'sorted', 'staticmethod',

'str', 'sum', 'super', 'tuple', 'type', 'vars', 'zip']

You’ll see some objects here that you may recognize from previous tutorials—for example, the StopIteration exception, built-in functions like max() and len(), and object types like int and str.

The Python interpreter creates the built-in namespace when it starts up. This namespace remains in existence until the interpreter terminates.

### The Global Namespace

The **global namespace** contains any names defined at the level of the main program. Python creates the global namespace when the main program body starts, and it remains in existence until the interpreter terminates.

Strictly speaking, this may not be the only global namespace that exists. The interpreter also creates a global namespace for any **module** that your program loads with the import statement.

When you see the term global namespace, think of the one belonging to the main program.

### The Local and Enclosing Namespaces

As you learned in functions, the interpreter creates a new namespace whenever a function executes. That namespace is local to the function and remains in existence until the function terminates.

Functions don’t exist independently from one another only at the level of the main program. You can also define one function inside another:

>>> def f():

... print('Start f()')

... def g():

... print('Start g()')

... print('End g()')

... return

... g()

... print('End f()')

... return

...

>>> f()

Start f()

Start g()

End g()

End f()

In this example, function g() is defined within the body of f(). Here’s what’s happening in this code:

* **Lines 1 to 12** define f(), the **enclosing** function.
* **Lines 4 to 7** define g(), the **enclosed** function.
* On **line 15**, the main program calls f().
* On **line 9**, f() calls g().

When the main program calls f(), Python creates a new namespace for f(). Similarly, when f() calls g(), g() gets its own separate namespace. The namespace created for g() is the **local namespace**, and the namespace created for f() is the **enclosing namespace**.

Each of these namespaces remains in existence until its respective function terminates. Python might not immediately reclaim the memory allocated for those namespaces when their functions terminate, but all references to the objects they contain cease to be valid.

## Variable Scope

The existence of multiple, distinct namespaces means several different instances of a particular name can exist simultaneously while a Python program runs. As long as each instance is in a different namespace, they’re all maintained separately and won’t interfere with one another.

But that raises a question: Suppose you refer to the name x in your code, and x exists in several namespaces. How does Python know which one you mean?

The answer lies in the concept of **scope**. The scope of a name is the region of a program in which that name has meaning. The interpreter determines this at runtime based on where the name definition occurs and where in the code the name is referenced.

To return to the above question, if your code refers to the name x, then Python searches for x in the following namespaces in the order shown:

1. **Local**: If you refer to x inside a function, then the interpreter first searches for it in the innermost scope that’s local to that function.
2. **Enclosing**: If x isn’t in the local scope but appears in a function that resides inside another function, then the interpreter searches in the enclosing function’s scope.
3. **Global**: If neither of the above searches is fruitful, then the interpreter looks in the global scope next.
4. **Built-in**: If it can’t find x anywhere else, then the interpreter tries the built-in scope.

This is the **LEGB rule** as it’s commonly called in Python literature (although the term doesn’t actually appear in the Python documentation). The interpreter searches for a name from the inside out, looking in the **l**ocal, **e**nclosing, **g**lobal, and finally the **b**uilt-in scope:



If the interpreter doesn’t find the name in any of these locations, then Python raises a NameError exception.

**Examples**

Several examples of the LEGB rule appear below. In each case, the innermost enclosed function g() attempts to display the value of a variable named x to the console. Notice how each example prints a different value for x depending on its scope.

**Example 1: Single Definition**

In the first example, x is defined in only one location. It’s outside both f() and g(), so it resides in the global scope:

>>> x = 'global'

>>> def f():

... def g():

... print(x)

... g()

...

>>> f()

global

The print() statement on **line 6** can refer to only one possible x. It displays the x object defined in the global namespace, which is the string 'global'.

**Example 2: Double Definition**

In the next example, the definition of x appears in two places, one outside f() and one inside f() but outside g():

>>> x = 'global'

>>> def f():

... x = 'enclosing'

... def g():

... print(x)

... g()

...

>>> f()

enclosing

As in the previous example, g() refers to x. But this time, it has two definitions to choose from:

* **Line 1** defines x in the global scope.
* **Line 4** defines x again in the enclosing scope.

According to the LEGB rule, the interpreter finds the value from the enclosing scope before looking in the global scope. So the print() statement on **line 7** displays 'enclosing' instead of 'global'.

**Example 3: Triple Definition**

Next is a situation in which x is defined here, there, and everywhere. One definition is outside f(), another one is inside f() but outside g(), and a third is inside g():

>>> x = 'global'

>>> def f():

... x = 'enclosing'

... def g():

... x = 'local'

... print(x)

... g()

...

>>> f()

local

Now the print() statement on **line 8** has to distinguish between three different possibilities:

* **Line 1** defines x in the global scope.
* **Line 4** defines x again in the enclosing scope.
* **Line 7** defines x a third time in the scope that’s local to g().

Here, the LEGB rule dictates that g() sees its own locally defined value of x first. So the print() statement displays 'local'.

**Example 4: No Definition**

Last, we have a case in which g() tries to print the value of x, but x isn’t defined anywhere. That won’t work at all:

>>> def f():

... def g():

... print(x)

... g()

...

>>> f()

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

File "<stdin>", line 6, in f

File "<stdin>", line 4, in g

NameError: name 'x' is not defined

This time, Python doesn’t find x in any of the namespaces, so the print() statement on **line 4** generates a NameError exception.

## Python Namespace Dictionaries

Earlier, when namespaces were first introduced, you were encouraged to think of a namespace as a dictionary in which the keys are the object names and the values are the objects themselves. In fact, for global and local namespaces, that’s precisely what they are! Python really does implement these namespaces as dictionaries.

**Note:** The built-in namespace doesn’t behave like a dictionary. Python implements it as a module.

Python provides built-in functions called globals() and locals() that allow you to access global and local namespace dictionaries.

### The globals() function

The built-in function globals() returns a reference to the current global namespace dictionary. You can use it to access the objects in the global namespace. Here’s an example of what it looks like when the main program starts:

>>> type(globals())

<class 'dict'>

>>> globals()

{'\_\_name\_\_': '\_\_main\_\_', '\_\_doc\_\_': None, '\_\_package\_\_': None,

'\_\_loader\_\_': <class '\_frozen\_importlib.BuiltinImporter'>, '\_\_spec\_\_': None,

'\_\_annotations\_\_': {}, '\_\_builtins\_\_': <module 'builtins' (built-in)>}

As you can see, the interpreter has put several entries in globals() already. Depending on your Python version and operating system, it may look a little different in your environment. But it should be similar.

Now watch what happens when you define a variable in the global scope:

>>> x = 'foo'

>>> globals()

{'\_\_name\_\_': '\_\_main\_\_', '\_\_doc\_\_': None, '\_\_package\_\_': None,

'\_\_loader\_\_': <class '\_frozen\_importlib.BuiltinImporter'>, '\_\_spec\_\_': None,

'\_\_annotations\_\_': {}, '\_\_builtins\_\_': <module 'builtins' (built-in)>,

'x': 'foo'}

After the assignment statement x = 'foo', a new item appears in the global namespace dictionary. The dictionary key is the object’s name, x, and the dictionary value is the object’s value, 'foo'.

You would typically access this object in the usual way, by referring to its symbolic name, x. But you can also access it indirectly through the global namespace dictionary:

>>> x

'foo'

>>> globals()['x']

'foo'

>>> x is globals()['x']

True

The is comparison on **line 6** confirms that these are in fact the same object.

You can create and modify entries in the global namespace using the globals() function as well:

>>> globals()['y'] = 100

>>> globals()

{'\_\_name\_\_': '\_\_main\_\_', '\_\_doc\_\_': None, '\_\_package\_\_': None,

'\_\_loader\_\_': <class '\_frozen\_importlib.BuiltinImporter'>, '\_\_spec\_\_': None,

'\_\_annotations\_\_': {}, '\_\_builtins\_\_': <module 'builtins' (built-in)>,

'x': 'foo', 'y': 100}

>>> y

100

>>> globals()['y'] = 3.14159

>>> y

153.14159

The statement on **line 1** has the same effect as the assignment statement y = 100. The statement on **line 12** is equivalent to y = 3.14159.

It’s a little off the beaten path to create and modify objects in the global scope this way when simple assignment statements will do. But it works, and it illustrates the concept nicely.

### The locals() function

Python also provides a corresponding built-in function called locals(). It’s similar to globals() but accesses objects in the local namespace instead:

>>> def f(x, y):

... s = 'foo'

... print(locals())

...

>>> f(10, 0.5)

{'s': 'foo', 'y': 0.5, 'x': 10}

When called within f(), locals() returns a dictionary representing the function’s local namespace. Notice that, in addition to the locally defined variable s, the local namespace includes the function parameters x and y since these are local to f() as well.

If you call locals() outside a function in the main program, then it behaves the same as globals().

**Deep Dive: A Subtle Difference Between globals() and locals()**

There’s one small difference between globals() and locals() that’s useful to know about.

globals() returns an actual reference to the dictionary that contains the global namespace. That means if you call globals(), save the return value, and subsequently define additional variables, then those new variables will show up in the dictionary that the saved return value points to:

1>>> g = globals()

2>>> g

3{'\_\_name\_\_': '\_\_main\_\_', '\_\_doc\_\_': None, '\_\_package\_\_': None,

4'\_\_loader\_\_': <class '\_frozen\_importlib.BuiltinImporter'>, '\_\_spec\_\_': None,

5'\_\_annotations\_\_': {}, '\_\_builtins\_\_': <module 'builtins' (built-in)>,

6'g': {...}}

7

8>>> x = 'foo'

9>>> y = 29

10>>> g

11{'\_\_name\_\_': '\_\_main\_\_', '\_\_doc\_\_': None, '\_\_package\_\_': None,

12'\_\_loader\_\_': <class '\_frozen\_importlib.BuiltinImporter'>, '\_\_spec\_\_': None,

13'\_\_annotations\_\_': {}, '\_\_builtins\_\_': <module 'builtins' (built-in)>,

14'g': {...}, 'x': 'foo', 'y': 29}

Here, g is a reference to the global namespace dictionary. After the assignment statements on **lines 8 and 9**, x and y appear in the dictionary that g points to.

locals(), on the other hand, returns a dictionary that is a current copy of the local namespace, not a reference to it. Further additions to the local namespace won’t affect a previous return value from locals() until you call it again. Also, you can’t modify objects in the actual local namespace using the return value from locals():

1>>> def f():

2... s = 'foo'

3... loc = locals()

4... print(loc)

5...

6... x = 20

7... print(loc)

8...

9... loc['s'] = 'bar'

10... print(s)

11...

12

13>>> f()

14{'s': 'foo'}

15{'s': 'foo'}

16foo

In this example, loc points to the return value from locals(), which is a copy of the local namespace. The statement x = 20 on **line 6** adds x to the local namespace but *not* to the copy that loc points to. Similarly, the statement on **line 9** modifies the value for key 's' in the copy that loc points to, but this has no effect on the value of s in the actual local namespace.

It’s a subtle difference, but it could cause you trouble if you don’t remember it.

## Modify Variables Out of Scope

Earlier in this series, in the tutorial on user-defined Python functions, you learned that argument passing in Python is a bit like pass-by-value and a bit like pass-by-reference. Sometimes a function can modify its argument in the calling environment by making changes to the corresponding parameter, and sometimes it can’t:

* An **immutable** argument can never be modified by a function.
* A **mutable** argument can’t be redefined wholesale, but it can be modified in place.

A similar situation exists when a function tries to modify a variable outside its local scope. A function can’t modify an immutable object outside its local scope at all:

>>> x = 20

>>> def f():

... x = 40

... print(x)

...

>>> f()

40

>>> x

20

When f() executes the assignment x = 40 on **line 3**, it creates a new local reference to an integer object whose value is 40. At that point, f() loses the reference to the object named x in the global namespace. So, the assignment statement doesn’t affect the global object.

Note that when f() executes print(x) on **line 4**, it displays 40, the value of its own local x. But after f() terminates, x in the global scope is still 20.

A function can modify an object of mutable type that’s outside its local scope if it modifies the object in place:

>>> my\_list = ['foo', 'bar', 'baz']

>>> def f():

... my\_list[1] = 'quux'

...

>>> f()

>>> my\_list

['foo', 'quux', 'baz']

In this case, my\_list is a list, and lists are mutable. f() can make changes inside my\_list even though it’s outside the local scope.

But if f() tries to reassign my\_list entirely, then it will create a new local object and won’t modify the global my\_list:

>>> my\_list = ['foo', 'bar', 'baz']

>>> def f():

... my\_list = ['qux', 'quux']

...

>>> f()

>>> my\_list

['foo', 'bar', 'baz']

This is similar to what happens when f() tries to modify a mutable function argument.

### The global Declaration

What if you really do need to modify a value in the global scope from within f()? This is possible in Python using the global declaration:

>>> x = 20

>>> def f():

... global x

... x = 40

... print(x)

...

>>> f()

40

>>> x

40

The global x statement indicates that while f() executes, references to the name x will refer to the x that is in the global namespace. That means the assignment x = 40 doesn’t create a new reference. It assigns a new value to x in the global scope instead:



As you’ve already seen, globals() returns a reference to the global namespace dictionary. If you wanted to, instead of using a global statement, you could accomplish the same thing using globals():

>>> x = 20

>>> def f():

... globals()['x'] = 40

... print(x)

...

>>> f()

40

>>> x

40

There isn’t much reason to do it this way since the global declaration arguably makes the intent clearer. But it does provide another illustration of how globals() works.

If the name specified in the global declaration doesn’t exist in the global scope when the function starts, then a combination of the global statement and an assignment will create it:

1>>> y

2Traceback (most recent call last):

3 File "<pyshell#79>", line 1, in <module>

4 y

5NameError: name 'y' is not defined

6

7>>> def g():

8... global y

9... y = 20

10...

11

12>>> g()

13>>> y

1420

In this case, there’s no object named y in the global scope when g() starts, but g() creates one with the global y statement on **line 8**.

You can also specify several comma-separated names in a single global declaration:

>>> x, y, z = 10, 20, 30

>>> def f():

... global x, y, z

...

The intent of the global x statement on **line 3** is to make references to x refer to an object in the global scope. But the print() statement on **line 2** refers to x to prior to the global declaration. This raises a SyntaxError exception.

### The nonlocal Declaration

A similar situation exists with nested function definitions. The global declaration allows a function to access and modify an object in the global scope. What if an enclosed function needs to modify an object in the enclosing scope? Consider this example:

>>> def f():

2... x = 20

3...

4... def g():

5... x = 40

6...

7... g()

8... print(x)

9...

10

11>>> f()

1220

In this case, the first definition of x is in the enclosing scope, not the global scope. Just as g() can’t directly modify a variable in the global scope, neither can it modify x in the enclosing function’s scope. Following the assignment x = 40 on **line 5**, x in the enclosing scope remains 20.

The global keyword isn’t a solution for this situation:

>>> def f():

... x = 20

...

... def g():

... global x

... x = 40

...

... g()

... print(x)

...

>>> f()

20

Since x is in the enclosing function’s scope, not the global scope, the global keyword doesn’t work here. After g() terminates, x in the enclosing scope remains 20.

In fact, in this example, the global x statement not only fails to provide access to x in the enclosing scope, but it also creates an object called x in the global scope whose value is 40:

>>> def f():

... x = 20

...

... def g():

... global x

... x = 40

...

... g()

... print(x)

...

>>> f()

20

>>> x

40

To modify x in the enclosing scope from inside g(), you need the analogous keyword [nonlocal](https://realpython.com/python-keywords/#variable-handling-keywords-del-global-nonlocal). Names specified after the nonlocal keyword refer to variables in the nearest enclosing scope:

1>>> def f():

2... x = 20

3...

4... def g():

5... nonlocal x

6... x = 40

7...

8... g()

9... print(x)

10...

11

12>>> f()

1340

After the nonlocal x statement on **line 5**, when g() refers to x, it refers to the x in the nearest enclosing scope, whose definition is in f() on **line 2**:

[](https://files.realpython.com/media/t.b7e3c1b7bd96.png)

The print() statement at the end of f() on **line 9** confirms that the call to g() has changed the value of x in the enclosing scope to 40.

# Referecnces

<https://code.visualstudio.com/docs/languages/python>

<https://code.visualstudio.com/docs/python/python-tutorial>

<https://www.geeksforgeeks.org/python-keywords-and-identifiers/>

<https://codesteps.com/2018/09/02/python-simple-and-compound-statements/>

<http://makemeanalyst.com/python-programming/>

<https://www.geeksforgeeks.org/namespaces-and-scope-in-python/>

https://realpython.com/

<https://realpython.com/python-namespaces-scope/>

<https://realpython.com/python3-object-oriented-programming/>

<https://realpython.com/defining-your-own-python-function/#pass-by-value-vs-pass-by-reference-in-pascal>

<https://realpython.com/python-lambda/>