Contents

[Installation 3](#_Toc73641366)

[Configure and run the debugger 3](#_Toc73641367)

[Install and use packages# 4](#_Toc73641368)

[Python REPL 6](#_Toc73641369)

[Python Identifiers and Keywords 8](#_Toc73641370)

[Keywords 8](#_Toc73641371)

[Identifiers 10](#_Toc73641372)

[Python Simple and Compound Statements 14](#_Toc73641373)

[Simple Statements 14](#_Toc73641374)

[Compound Statements 14](#_Toc73641375)

[Python Values, Types and Variables 17](#_Toc73641376)

[Values and types 17](#_Toc73641377)

[Python Built-in Data Types 18](#_Toc73641378)

[Getting the Data Type 18](#_Toc73641379)

[Setting the Data Type 18](#_Toc73641380)

[Python Statements 20](#_Toc73641381)

[Multi-line statement 20](#_Toc73641382)

[Python : Indentation 21](#_Toc73641383)

[Python Operators and Expressions 22](#_Toc73641384)

[Arithmetic Operators 22](#_Toc73641385)

[Relational Operators 22](#_Toc73641386)

[Bitwise Operators 23](#_Toc73641387)

[Assignment operators 24](#_Toc73641388)

[Identity operators 24](#_Toc73641389)

[Membership Operators 24](#_Toc73641390)

[Expressions 25](#_Toc73641391)

[Order of operations 25](#_Toc73641392)

[String operations: 25](#_Toc73641393)

[Take Input from User in Python 26](#_Toc73641394)

[Take an integer as an Input 26](#_Toc73641395)

[Python Comments 27](#_Toc73641396)

[Multi-line comments 27](#_Toc73641397)

[Docstring in Python 27](#_Toc73641398)

[Declaration of docstrings 27](#_Toc73641399)

[Python Conditional Execution 29](#_Toc73641400)

[if Statement 29](#_Toc73641401)

[Alternative execution: if-else Statement 29](#_Toc73641402)

[Chained conditionals: if…elif…else 29](#_Toc73641403)

[Nested conditionals 29](#_Toc73641404)

[Catching exceptions using try and except 30](#_Toc73641405)

[Python Functions 31](#_Toc73641406)

[Define function and function call 31](#_Toc73641407)

[Abstraction and Reusability 31](#_Toc73641408)

[Modularity 31](#_Toc73641409)

[Namespace Separation 33](#_Toc73641410)

[Parameters and Arguments 33](#_Toc73641411)

[Built-in Functions in Python 33](#_Toc73641412)

[Type Conversion Functions 33](#_Toc73641413)

[Random Numbers 34](#_Toc73641414)

[randint() Function in Python 34](#_Toc73641415)

[choice() Function in Python 34](#_Toc73641416)

[Math functions in Python 34](#_Toc73641417)

[Positional Arguments 34](#_Toc73641418)

[Keyword Arguments 36](#_Toc73641419)

[Default Parameters 37](#_Toc73641420)

[Mutable Default Parameter Values 37](#_Toc73641421)

[Pass-By-Value vs Pass-By-Reference 39](#_Toc73641422)

[Pass-By-Value vs Pass-By-Reference in Python 40](#_Toc73641423)

[Python Lambda Functions 44](#_Toc73641424)

[Lambda Calculus 44](#_Toc73641425)

[History 44](#_Toc73641426)

[First Example 44](#_Toc73641427)

[Anonymous Functions 46](#_Toc73641428)

[Arguments 47](#_Toc73641429)

[Closure 48](#_Toc73641430)

[Python Exceptions 50](#_Toc73641431)

[Exceptions versus Syntax Errors 50](#_Toc73641432)

[Raising an Exception 50](#_Toc73641433)

[The AssertionError Exception 51](#_Toc73641434)

[The try and except Block: Handling Exceptions 51](#_Toc73641435)

[The else Clause 54](#_Toc73641436)

[Cleaning Up After Using finally 55](#_Toc73641437)

[Python Loops 57](#_Toc73641438)

[Python While Loop 57](#_Toc73641439)

[Python “Infinite loops” and break 57](#_Toc73641440)

[Finishing iterations with continue in Python 58](#_Toc73641441)

[Python for Loop 58](#_Toc73641442)

[The range() function in Python 59](#_Toc73641443)

[Bonus Example: Counting and summing loops 59](#_Toc73641444)

[Bonus Example: Maximum and minimum loops 59](#_Toc73641445)

[Python Strings 60](#_Toc73641446)

[Getting the length of a string using len() function 60](#_Toc73641447)

[String Methods 60](#_Toc73641448)

[String % 60](#_Toc73641449)

[Single, Double and Triple Quotes 61](#_Toc73641450)

[Single Quotes 61](#_Toc73641451)

[Double Quotes 61](#_Toc73641452)

[Triple Quotes 62](#_Toc73641453)

[Traversing a  string with a loop 62](#_Toc73641454)

[String slices 62](#_Toc73641455)

[Strings are immutable 62](#_Toc73641456)

[Looping and counting 63](#_Toc73641457)

[The in operator in Python 63](#_Toc73641458)

[String comparison 63](#_Toc73641459)

[Python Lists 64](#_Toc73641460)

[How to create a list? 64](#_Toc73641461)

[Lists are mutable 65](#_Toc73641462)

[How to access elements from a list? 65](#_Toc73641463)

[List Index 65](#_Toc73641464)

[Negative indexing 65](#_Toc73641465)

[Traversing a list 65](#_Toc73641466)

[List operations 66](#_Toc73641467)

[How to slice lists in Python? 66](#_Toc73641468)

[List methods 66](#_Toc73641469)

[How to delete or remove elements from a list? 67](#_Toc73641470)

[***pop*** *operator* 67](#_Toc73641471)

[***del*** operator 67](#_Toc73641472)

[remove() Function 67](#_Toc73641473)

[Lists and functions 67](#_Toc73641474)

[Lists and strings 68](#_Toc73641475)

[List arguments 69](#_Toc73641476)

[Lists Are Ordered 69](#_Toc73641477)

[Lists Can Contain Arbitrary Objects 70](#_Toc73641478)

[List Elements Can Be Accessed by Index 71](#_Toc73641479)

[Lists Can Be Nested 75](#_Toc73641480)

[Lists Are Mutable 77](#_Toc73641481)

[Modifying a Single List Value 77](#_Toc73641482)

[Modifying Multiple List Values 78](#_Toc73641483)

[Prepending or Appending Items to a List 79](#_Toc73641484)

[Methods That Modify a List 80](#_Toc73641485)

[Lists Are Dynamic 83](#_Toc73641486)

[Python Tuples 84](#_Toc73641487)

[Defining and Using Tuples 84](#_Toc73641488)

[Tuple Assignment, Packing, and Unpacking 86](#_Toc73641489)

[Swap in Python using Tuple (assignment) 88](#_Toc73641490)

[Python Dictionaries 90](#_Toc73641491)

[How to create a dictionary? 90](#_Toc73641492)

[How to access elements from a dictionary? 90](#_Toc73641493)

[How to change or add elements in a dictionary? 91](#_Toc73641494)

[Dictionary Membership Test 91](#_Toc73641495)

[How to delete or remove elements from a dictionary? 91](#_Toc73641496)

[Python Dictionary Methods 92](#_Toc73641497)

[Python Sets 94](#_Toc73641498)

[Defining a Set 94](#_Toc73641499)

[Set Size and Membership 96](#_Toc73641500)

[Operating on a Set 97](#_Toc73641501)

[Operators vs. Methods 97](#_Toc73641502)

[Available Operators and Methods 98](#_Toc73641503)

[Modifying a Set 103](#_Toc73641504)

[Augmented Assignment Operators and Methods 103](#_Toc73641505)

[Other Methods For Modifying Sets 105](#_Toc73641506)

[Frozen Sets 107](#_Toc73641507)

[Object Oriented Programming in Python 3 109](#_Toc73641508)

[Define a Class in Python 109](#_Toc73641509)

[Classes vs Instances 110](#_Toc73641510)

[How to Define a Class 110](#_Toc73641511)

[Instantiate an Object in Python 111](#_Toc73641512)

[Class and Instance Attributes 112](#_Toc73641513)

[Instance Methods 114](#_Toc73641514)

[Inherit From Other Classes in Python 115](#_Toc73641515)

[Dog Park Example 116](#_Toc73641516)

[Parent Classes vs Child Classes 117](#_Toc73641517)

[Extend the Functionality of a Parent Class 118](#_Toc73641518)

[Namespaces 121](#_Toc73641519)

[Namespaces in Python 121](#_Toc73641520)

[The Built-In Namespace 121](#_Toc73641521)

[The Global Namespace 122](#_Toc73641522)

[The Local and Enclosing Namespaces 122](#_Toc73641523)

[Variable Scope 123](#_Toc73641524)

[Python Namespace Dictionaries 126](#_Toc73641525)

[The globals() function 126](#_Toc73641526)

[The locals() function 128](#_Toc73641527)

[Modify Variables Out of Scope 129](#_Toc73641528)

[The global Declaration 131](#_Toc73641529)

[The nonlocal Declaration 132](#_Toc73641530)

[Python File Handling 135](#_Toc73641531)

[open() 135](#_Toc73641532)

[read() 135](#_Toc73641533)

[readlines() 136](#_Toc73641534)

[**write()** 136](#_Toc73641535)

[append() 137](#_Toc73641536)

[with() 137](#_Toc73641537)

[split() 138](#_Toc73641538)

[Logging in Python 139](#_Toc73641539)

[The Logging Module 139](#_Toc73641540)

[Basic Configurations 140](#_Toc73641541)

[Formatting the Output 141](#_Toc73641542)

[Logging Variable Data 142](#_Toc73641543)

[Capturing Stack Traces 143](#_Toc73641544)

[Classes and Functions 144](#_Toc73641545)

[Using Handlers 145](#_Toc73641546)

[Python Collections 147](#_Toc73641547)

[Counter 147](#_Toc73641548)

[namedtuple 147](#_Toc73641549)

[defaultdict 148](#_Toc73641550)

[ChainMap 149](#_Toc73641551)

[Python Datetime 152](#_Toc73641552)

[Programming With Dates and Times 152](#_Toc73641553)

[How Computers Count Time 152](#_Toc73641554)

[How Standard Dates Can Be Reported 153](#_Toc73641555)

[How Time Should Be Stored in Your Program 154](#_Toc73641556)

[Using the Python datetime Module 154](#_Toc73641557)

[Creating Python datetime Instances 155](#_Toc73641558)

[Using Strings to Create Python datetime Instances 156](#_Toc73641559)

[Working With Time Zones 158](#_Toc73641560)

[Using dateutil to Add Time Zones to Python datetime 158](#_Toc73641561)

[Comparing Naive and Aware Python datetime Instances 159](#_Toc73641562)

[Doing Arithmetic With Python datetime 159](#_Toc73641563)

[Alternatives to Python datetime and dateutil 161](#_Toc73641564)

[Python RegEx 163](#_Toc73641565)

[RegEx Module 163](#_Toc73641566)

[RegEx in Python 163](#_Toc73641567)

[RegEx Functions 163](#_Toc73641568)

[Metacharacters 163](#_Toc73641569)

[Special Sequences 164](#_Toc73641570)

[Sets 164](#_Toc73641571)

[The findall() Function 165](#_Toc73641572)

[The search() Function 166](#_Toc73641573)

[The split() Function 166](#_Toc73641574)

[The sub() Function 166](#_Toc73641575)

[Match Object 167](#_Toc73641576)

[Python Modules and Packages 178](#_Toc73641577)

[Python Modules: Overview 178](#_Toc73641578)

[The Module Search Path 179](#_Toc73641579)

[The import Statement 181](#_Toc73641580)

[**import <module\_name>** 181](#_Toc73641581)

[from <module\_name> import <name(s)> 181](#_Toc73641582)

[from <module\_name> import <name> as <alt\_name> 183](#_Toc73641583)

[import <module\_name> as <alt\_name> 183](#_Toc73641584)

[The dir() Function 184](#_Toc73641585)

[Executing a Module as a Script 186](#_Toc73641586)

[Reloading a Module 189](#_Toc73641587)

[Python Packages 190](#_Toc73641588)

[Package Initialization 192](#_Toc73641589)

[Importing \* From a Package 194](#_Toc73641590)

[Subpackages 197](#_Toc73641591)

[PyPl & pip 200](#_Toc73641592)

[Pylint 201](#_Toc73641593)

[Referecnces 202](#_Toc73641594)

# 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 Exceptions

A Python program terminates as soon as it encounters an error. In Python, an error can be a syntax error or an exception. Let’s see what an exception is and how it differs from a syntax error.



## Exceptions versus Syntax Errors

Syntax errors occur when the parser detects an incorrect statement. Observe the following example:

>>> print( 0 / 0 ))

File "<stdin>", line 1

print( 0 / 0 ))

^

SyntaxError: invalid syntax

The arrow indicates where the parser ran into the **syntax error**. In this example, there was one bracket too many. Remove it and run your code again:

>>> print( 0 / 0)

Traceback (most recent call last):

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

ZeroDivisionError: integer division or modulo by zero

This time, you ran into an **exception error**. This type of error occurs whenever syntactically correct Python code results in an error. The last line of the message indicated what type of exception error you ran into.

Instead of showing the message exception error, Python details what type of exception error was encountered. In this case, it was a ZeroDivisionError. Python comes with various built-in exceptions as well as the possibility to create self-defined exceptions.

## Raising an Exception

We can use raise to throw an exception if a condition occurs. The statement can be complemented with a custom exception.

If you want to throw an error when a certain condition occurs using raise, you could go about it like this:

x = 10

if x > 5:

raise Exception('x should not exceed 5. The value of x was: {}'.format(x))

When you run this code, the output will be the following:

Traceback (most recent call last):

File "<input>", line 4, in <module>

Exception: x should not exceed 5. The value of x was: 10

The program comes to a halt and displays our exception to screen, offering clues about what went wrong.

## The AssertionError Exception

Instead of waiting for a program to crash midway, you can also start by making an assertion in Python. We assert that a certain condition is met. If this condition turns out to be True, then that is excellent! The program can continue. If the condition turns out to be False, you can have the program throw an AssertionError exception.



Have a look at the following example, where it is asserted that the code will be executed on a Linux system:

import sys

assert ('linux' in sys.platform), "This code runs on Linux only."

If you run this code on a Linux machine, the assertion passes. If you were to run this code on a Windows machine, the outcome of the assertion would be False and the result would be the following:

Traceback (most recent call last):

File "<input>", line 2, in <module>

AssertionError: This code runs on Linux only.

In this example, throwing an AssertionError exception is the last thing that the program will do. The program will come to halt and will not continue. What if that is not what you want?

## The try and except Block: Handling Exceptions

The try and except block in Python is used to catch and handle exceptions. Python executes code following the try statement as a “normal” part of the program. The code that follows the except statement is the program’s response to any exceptions in the preceding try clause.



As you saw earlier, when syntactically correct code runs into an error, Python will throw an exception error. This exception error will crash the program if it is unhandled. The except clause determines how your program responds to exceptions.

The following function can help you understand the try and except block:

def linux\_interaction():

assert ('linux' in sys.platform), "Function can only run on Linux systems."

print('Doing something.')

The linux\_interaction() can only run on a Linux system. The assert in this function will throw an AssertionError exception if you call it on an operating system other than Linux.

You can give the function a try using the following code:

try:

linux\_interaction()

except:

pass

The way you handled the error here is by handing out a pass. If you were to run this code on a Windows machine, you would get no output.

You got nothing. The good thing here is that the program did not crash. But it would be nice to see if some type of exception occurred whenever you ran your code. To this end, you can change the pass into something that would generate an informative message, like so:

try:

linux\_interaction()

except:

print('Linux function was not executed')

Execute this code on a Windows machine and you get:

Linux function was not executed

When an exception occurs in a program running this function, the program will continue as well as inform you about the fact that the function call was not successful.

What you did not get to see was the type of error that was thrown as a result of the function call. In order to see exactly what went wrong, you would need to catch the error that the function threw.

The following code is an example where you capture the AssertionError and output that message to screen:

try:

linux\_interaction()

except AssertionError as error:

print(error)

print('The linux\_interaction() function was not executed')

Running this function on a Windows machine outputs the following:

Function can only run on Linux systems.

The linux\_interaction() function was not executed

The first message is the AssertionError, informing you that the function can only be executed on a Linux machine. The second message tells you which function was not executed.

In the previous example, you called a function that you wrote yourself. When you executed the function, you caught the AssertionError exception and printed it to screen.

Here’s another example where you open a file and use a built-in exception:

try:

with open('file.log') as file:

read\_data = file.read()

except:

print('Could not open file.log')

If file.log does not exist, this block of code will output the following:

Could not open file.log

This is an informative message, and our program will still continue to run. In the Python docs, you can see that there are a lot of built-in exceptions that you can use here. One exception described on that page is the following:

**Exception** FileNotFoundError

**Raised** when a file or directory is requested but doesn’t exist. Corresponds to errno ENOENT.

To catch this type of exception and print it to screen, you could use the following code:

try:

with open('file.log') as file:

read\_data = file.read()

except FileNotFoundError as fnf\_error:

print(fnf\_error)

In this case, if file.log does not exist, the output will be the following:

[Errno 2] No such file or directory: 'file.log'

You can have more than one function call in your try clause and anticipate catching various exceptions. A thing to note here is that the code in the try clause will stop as soon as an exception is encountered.

**Warning:** Catching Exception hides all errors…even those which are completely unexpected. This is why you should avoid bare except clauses in your Python programs. Instead, you’ll want to refer to specific exception classes you want to catch and handle.

Look at the following code. Here, you first call the linux\_interaction() function and then try to open a file:

try:

linux\_interaction()

with open('file.log') as file:

read\_data = file.read()

except FileNotFoundError as fnf\_error:

print(fnf\_error)

except AssertionError as error:

print(error)

print('Linux linux\_interaction() function was not executed')

If the file does not exist, running this code on a Windows machine will output the following:

Function can only run on Linux systems.

Linux linux\_interaction() function was not executed

Inside the try clause, you ran into an exception immediately and did not get to the part where you attempt to open file.log. Now look at what happens when you run the code on a Linux machine:

[Errno 2] No such file or directory: 'file.log'

Here are the key takeaways:

* A try clause is executed up until the point where the first exception is encountered.
* Inside the except clause, or the exception handler, you determine how the program responds to the exception.
* You can anticipate multiple exceptions and differentiate how the program should respond to them.
* Avoid using bare except clauses.

## The else Clause

In Python, using the else statement, you can instruct a program to execute a certain block of code only in the absence of exceptions.



Look at the following example:

try:

linux\_interaction()

except AssertionError as error:

print(error)

else:

print('Executing the else clause.')

If you were to run this code on a Linux system, the output would be the following:

Doing something.

Executing the else clause.

Because the program did not run into any exceptions, the else clause was executed.

You can also try to run code inside the else clause and catch possible exceptions there as well:

try:

linux\_interaction()

except AssertionError as error:

print(error)

else:

try:

with open('file.log') as file:

read\_data = file.read()

except FileNotFoundError as fnf\_error:

print(fnf\_error)

If you were to execute this code on a Linux machine, you would get the following result:

Doing something.

[Errno 2] No such file or directory: 'file.log'

From the output, you can see that the linux\_interaction() function ran. Because no exceptions were encountered, an attempt to open file.log was made. That file did not exist, and instead of opening the file, you caught the FileNotFoundError exception.

## Cleaning Up After Using finally

Imagine that you always had to implement some sort of action to clean up after executing your code. Python enables you to do so using the finally clause.



Have a look at the following example:

try:

linux\_interaction()

except AssertionError as error:

print(error)

else:

try:

with open('file.log') as file:

read\_data = file.read()

except FileNotFoundError as fnf\_error:

print(fnf\_error)

finally:

print('Cleaning up, irrespective of any exceptions.')

In the previous code, everything in the finally clause will be executed. It does not matter if you encounter an exception somewhere in the try or else clauses. Running the previous code on a Windows machine would output the following:

Function can only run on Linux systems.

Cleaning up, irrespective of any exceptions.

# 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.

A list is a collection of arbitrary objects, somewhat akin to an array in many other programming languages but more flexible. Lists are defined in Python by enclosing a comma-separated sequence of objects in square brackets ([]), as shown below:

>>> a = ['foo', 'bar', 'baz', 'qux']

>>> print(a)

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

>>> a

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

The important characteristics of Python lists are as follows:

* Lists are ordered.
* Lists can contain any arbitrary objects.
* List elements can be accessed by index.
* Lists can be nested to arbitrary depth.
* Lists are mutable.
* Lists are dynamic.

## 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

## Lists Are Ordered

A list is not merely a collection of objects. It is an ordered collection of objects. The order in which you specify the elements when you define a list is an innate characteristic of that list and is maintained for that list’s lifetime (y*ou will see a Python data type that is not ordered: dictionaries*).

Lists that have the same elements in a different order are not the same:

>>> a = ['foo', 'bar', 'baz', 'qux']

>>> b = ['baz', 'qux', 'bar', 'foo']

>>> a == b

False

>>> a is b

False

>>> [1, 2, 3, 4] == [4, 1, 3, 2]

False

## Lists Can Contain Arbitrary Objects

A list can contain any assortment of objects. The elements of a list can all be the same type:

>>> a = [2, 4, 6, 8]

>>> a

[2, 4, 6, 8]

Or the elements can be of varying types:

>>> a = [21.42, 'foobar', 3, 4, 'bark', False, 3.14159]

>>> a

[21.42, 'foobar', 3, 4, 'bark', False, 3.14159]

Lists can even contain complex objects, like functions, classes, and modules (*more about modules later*).

>>> int

<class 'int'>

>>> len

<built-in function len>

>>> def foo():

... pass

...

>>> foo

<function foo at 0x035B9030>

>>> import math

>>> math

<module 'math' (built-in)>

>>> a = [int, len, foo, math]

>>> a

[<class 'int'>, <built-in function len>, <function foo at 0x02CA2618>,

<module 'math' (built-in)>]

A list can contain any number of objects, from zero to as many as your computer’s memory will allow:

>>> a = []

>>> a

[]

>>> a = [ 'foo' ]

>>> a

['foo']

>>> a = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20,

... 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40,

... 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60,

... 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80,

... 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 100]

>>> a

[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20,

21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39,

40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58,

59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77,

78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96,

97, 98, 99, 100]

(A list with a single object is sometimes referred to as a singleton list.)

List objects needn’t be unique. A given object can appear in a list multiple times:

>>> a = ['bark', 'meow', 'woof', 'bark', 'cheep', 'bark']

>>> a

['bark', 'meow', 'woof', 'bark', 'cheep', 'bark']

## List Elements Can Be Accessed by Index

Individual elements in a list can be accessed using an index in square brackets. This is exactly analogous to accessing individual characters in a string. List indexing is zero-based as it is with strings.

Consider the following list:

>>> a = ['foo', 'bar', 'baz', 'qux', 'quux', 'corge']

The indices for the elements in a are shown below:

[](https://files.realpython.com/media/t.eb0b38e642c5.png)

Here is Python code to access some elements of a:

>>> a[0]

'foo'

>>> a[2]

'baz'

>>> a[5]

'corge'

Virtually everything about string indexing works similarly for lists. For example, a negative list index counts from the end of the list:

[](https://files.realpython.com/media/t.c11ea56e8ca2.png)

>>> a[-1]

'corge'

>>> a[-2]

'quux'

>>> a[-5]

'bar'

Slicing also works. If a is a list, the expression a[m:n] returns the portion of a from index m to, but not including, index n:

>>> a = ['foo', 'bar', 'baz', 'qux', 'quux', 'corge']

>>> a[2:5]

['baz', 'qux', 'quux']

Other features of string slicing work analogously for list slicing as well:

* Both positive and negative indices can be specified:

>>> a[-5:-2]

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

>>> a[1:4]

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

>>> a[-5:-2] == a[1:4]

True

* Omitting the first index starts the slice at the beginning of the list, and omitting the second index extends the slice to the end of the list:

>>> print(a[:4], a[0:4])

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

>>> print(a[2:], a[2:len(a)])

['baz', 'qux', 'quux', 'corge'] ['baz', 'qux', 'quux', 'corge']

>>> a[:4] + a[4:]

['foo', 'bar', 'baz', 'qux', 'quux', 'corge']

>>> a[:4] + a[4:] == a

True

* You can specify a stride—either positive or negative (e.g.; pick every 2nd element):

>>> a[0:6:2]

['foo', 'baz', 'quux']

>>> a[1:6:2]

['bar', 'qux', 'corge']

>>> a[6:0:-2]

['corge', 'qux', 'bar']

* The syntax for reversing a list works the same way it does for strings:

>>> a[::-1]

['corge', 'quux', 'qux', 'baz', 'bar', 'foo']

* The [:] syntax works for lists. However, there is an important difference between how this operation works with a list and how it works with a string.

If s is a string, s[:] returns a reference to the same object:

>>> s = 'foobar'

>>> s[:]

'foobar'

>>> s[:] is s

True

Conversely, if a is a list, a[:] returns a new object that is a copy of a:

>>> a = ['foo', 'bar', 'baz', 'qux', 'quux', 'corge']

>>> a[:]

['foo', 'bar', 'baz', 'qux', 'quux', 'corge']

>>> a[:] is a

False

Several Python operators and built-in functions can also be used with lists in ways that are analogous to strings:

* The in and not in operators:

>>> a

['foo', 'bar', 'baz', 'qux', 'quux', 'corge']

>>> 'qux' in a

True

>>> 'thud' not in a

True

* The concatenation (+) and replication (\*) operators:

>>> a

['foo', 'bar', 'baz', 'qux', 'quux', 'corge']

>>> a + ['grault', 'garply']

['foo', 'bar', 'baz', 'qux', 'quux', 'corge', 'grault', 'garply']

>>> a \* 2

['foo', 'bar', 'baz', 'qux', 'quux', 'corge', 'foo', 'bar', 'baz',

'qux', 'quux', 'corge']

* The len(), min(), and max() functions:

>>> a

['foo', 'bar', 'baz', 'qux', 'quux', 'corge']

>>> len(a)

6

>>> min(a)

'bar'

>>> max(a)

'qux'

It’s not an accident that strings and lists behave so similarly. They are both special cases of a more general object type called an *iterable*.

By the way, in each example above, the list is always assigned to a variable before an operation is performed on it. But you can operate on a list literal as well:

>>> ['foo', 'bar', 'baz', 'qux', 'quux', 'corge'][2]

'baz'

>>> ['foo', 'bar', 'baz', 'qux', 'quux', 'corge'][::-1]

['corge', 'quux', 'qux', 'baz', 'bar', 'foo']

>>> 'quux' in ['foo', 'bar', 'baz', 'qux', 'quux', 'corge']

True

>>> ['foo', 'bar', 'baz'] + ['qux', 'quux', 'corge']

['foo', 'bar', 'baz', 'qux', 'quux', 'corge']

>>> len(['foo', 'bar', 'baz', 'qux', 'quux', 'corge'][::-1])

6

For that matter, you can do likewise with a string literal:

>>> 'If Comrade Napoleon says it, it must be right.'[::-1]

'.thgir eb tsum ti ,ti syas noelopaN edarmoC fI'

## Lists Can Be Nested

You have seen that an element in a list can be any sort of object. That includes another list. A list can contain sublists, which in turn can contain sublists themselves, and so on to arbitrary depth.

Consider this (admittedly contrived) example:

>>> x = ['a', ['bb', ['ccc', 'ddd'], 'ee', 'ff'], 'g', ['hh', 'ii'], 'j']

>>> x

['a', ['bb', ['ccc', 'ddd'], 'ee', 'ff'], 'g', ['hh', 'ii'], 'j']

The object structure that x references is diagrammed below:



x[0], x[2], and x[4] are strings, each one character long:

>>> print(x[0], x[2], x[4])

a g j

But x[1] and x[3] are sublists:

>>> x[1]

['bb', ['ccc', 'ddd'], 'ee', 'ff']

>>> x[3]

['hh', 'ii']

To access the items in a sublist, simply append an additional index:

>>> x[1]

['bb', ['ccc', 'ddd'], 'ee', 'ff']

>>> x[1][0]

'bb'

>>> x[1][1]

['ccc', 'ddd']

>>> x[1][2]

'ee'

>>> x[1][3]

'ff'

>>> x[3]

['hh', 'ii']

>>> print(x[3][0], x[3][1])

hh ii

x[1][1] is yet another sublist, so adding one more index accesses its elements:

>>> x[1][1]

['ccc', 'ddd']

>>> print(x[1][1][0], x[1][1][1])

ccc ddd

There is no limit, short of the extent of your computer’s memory, to the depth or complexity with which lists can be nested in this way.

All the usual syntax regarding indices and slicing applies to sublists as well:

>>> x[1][1][-1]

'ddd'

>>> x[1][1:3]

[['ccc', 'ddd'], 'ee']

>>> x[3][::-1]

['ii', 'hh']

However, be aware that operators and functions apply to only the list at the level you specify and are not recursive. Consider what happens when you query the length of x using len():

>>> x

['a', ['bb', ['ccc', 'ddd'], 'ee', 'ff'], 'g', ['hh', 'ii'], 'j']

>>> len(x)

5

>>> x[0]

'a'

>>> x[1]

['bb', ['ccc', 'ddd'], 'ee', 'ff']

>>> x[2]

'g'

>>> x[3]

['hh', 'ii']

>>> x[4]

'j'

x has only five elements—three strings and two sublists. The individual elements in the sublists don’t count toward x’s length.

You’d encounter a similar situation when using the in operator:

>>> 'ddd' in x

False

>>> 'ddd' in x[1]

False

>>> 'ddd' in x[1][1]

True

'ddd' is not one of the elements in x or x[1]. It is only directly an element in the sublist x[1][1]. An individual element in a sublist does not count as an element of the parent list(s).

## Lists Are Mutable

Once a list has been created, elements can be added, deleted, shifted, and moved around at will. Python provides a wide range of ways to modify lists.

### Modifying a Single List Value

A single value in a list can be replaced by indexing and simple assignment:

>>> a = ['foo', 'bar', 'baz', 'qux', 'quux', 'corge']

>>> a

['foo', 'bar', 'baz', 'qux', 'quux', 'corge']

>>> a[2] = 10

>>> a[-1] = 20

>>> a

['foo', 'bar', 10, 'qux', 'quux', 20]

You can’t do this with a string:

>>> s = 'foobarbaz'

>>> s[2] = 'x'

Traceback (most recent call last):

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

TypeError: 'str' object does not support item assignment

A list item can be deleted with the del command:

>>> a = ['foo', 'bar', 'baz', 'qux', 'quux', 'corge']

>>> del a[3]

>>> a

['foo', 'bar', 'baz', 'quux', 'corge']

### Modifying Multiple List Values

What if you want to change several contiguous elements in a list at one time? Python allows this with slice assignment, which has the following syntax:

a[m:n] = <iterable>

Again, for the moment, think of an iterable as a list. This assignment replaces the specified slice of a with <iterable>:

>>> a = ['foo', 'bar', 'baz', 'qux', 'quux', 'corge']

>>> a[1:4]

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

>>> a[1:4] = [1.1, 2.2, 3.3, 4.4, 5.5]

>>> a

['foo', 1.1, 2.2, 3.3, 4.4, 5.5, 'quux', 'corge']

>>> a[1:6]

[1.1, 2.2, 3.3, 4.4, 5.5]

>>> a[1:6] = ['Bark!']

>>> a

['foo', 'Bark!', 'quux', 'corge']

The number of elements inserted need not be equal to the number replaced. Python just grows or shrinks the list as needed.

You can insert multiple elements in place of a single element—just use a slice that denotes only one element:

>>> a = [1, 2, 3]

>>> a[1:2] = [2.1, 2.2, 2.3]

>>> a

[1, 2.1, 2.2, 2.3, 3]

Note that this is not the same as replacing the single element with a list:

>>> a = [1, 2, 3]

>>> a[1] = [2.1, 2.2, 2.3]

>>> a

[1, [2.1, 2.2, 2.3], 3]

You can also insert elements into a list without removing anything. Simply specify a slice of the form [n:n] (a zero-length slice) at the desired index:

>>> a = [1, 2, 7, 8]

>>> a[2:2] = [3, 4, 5, 6]

>>> a

[1, 2, 3, 4, 5, 6, 7, 8]

You can delete multiple elements out of the middle of a list by assigning the appropriate slice to an empty list. You can also use the del statement with the same slice:

>>> a = ['foo', 'bar', 'baz', 'qux', 'quux', 'corge']

>>> a[1:5] = []

>>> a

['foo', 'corge']

>>> a = ['foo', 'bar', 'baz', 'qux', 'quux', 'corge']

>>> del a[1:5]

>>> a

['foo', 'corge']

## Prepending or Appending Items to a List

Additional items can be added to the start or end of a list using the + concatenation operator or the += augmented assignment operator:

>>> a = ['foo', 'bar', 'baz', 'qux', 'quux', 'corge']

>>> a += ['grault', 'garply']

>>> a

['foo', 'bar', 'baz', 'qux', 'quux', 'corge', 'grault', 'garply']

>>> a = ['foo', 'bar', 'baz', 'qux', 'quux', 'corge']

>>> a = [10, 20] + a

>>> a

[10, 20, 'foo', 'bar', 'baz', 'qux', 'quux', 'corge']

Note that a list must be concatenated with another list, so if you want to add only one element, you need to specify it as a singleton list:

>>> a = ['foo', 'bar', 'baz', 'qux', 'quux', 'corge']

>>> a += 20

Traceback (most recent call last):

File "<pyshell#58>", line 1, in <module>

a += 20

TypeError: 'int' object is not iterable

>>> a += [20]

>>> a

['foo', 'bar', 'baz', 'qux', 'quux', 'corge', 20]

## Methods That Modify a List

Finally, Python supplies several built-in methods that can be used to modify lists. Information on these methods is detailed below.

**Note:** The string methods you saw in the previous tutorial did not modify the target string directly. That is because strings are immutable. Instead, string methods return a new string object that is modified as directed by the method. They leave the original target string unchanged:

>>> s = 'foobar'

>>> t = s.upper()

>>> print(s, t)

foobar FOOBAR

List methods are different. Because lists are mutable, the list methods shown here modify the target list in place.

**a.append(<obj>)**

Appends an object to a list.

a.append(<obj>) appends object <obj> to the end of list a:

>>> a = ['a', 'b']

>>> a.append(123)

>>> a

['a', 'b', 123]

Remember, list methods modify the target list in place. They do not return a new list:

>>> a = ['a', 'b']

>>> x = a.append(123)

>>> print(x)

None

>>> a

['a', 'b', 123]

Remember that when the + operator is used to concatenate to a list, if the target operand is an iterable, then its elements are broken out and appended to the list individually:

>>> a = ['a', 'b']

>>> a + [1, 2, 3]

['a', 'b', 1, 2, 3]

The .append() method does not work that way! If an iterable is appended to a list with .append(), it is added as a single object:

>>> a = ['a', 'b']

>>> a.append([1, 2, 3])

>>> a

['a', 'b', [1, 2, 3]]

Thus, with .append(), you can append a string as a single entity:

>>> a = ['a', 'b']

>>> a.append('foo')

>>> a

['a', 'b', 'foo']

**a.extend(<iterable>)**

Extends a list with the objects from an iterable.

Yes, this is probably what you think it is. .extend() also adds to the end of a list, but the argument is expected to be an iterable. The items in <iterable> are added individually:

>>> a = ['a', 'b']

>>> a.extend([1, 2, 3])

>>> a

['a', 'b', 1, 2, 3]

In other words, .extend() behaves like the + operator. More precisely, since it modifies the list in place, it behaves like the += operator:

>>> a = ['a', 'b']

>>> a += [1, 2, 3]

>>> a

['a', 'b', 1, 2, 3]

**a.insert(<index>, <obj>)**

Inserts an object into a list.

a.insert(<index>, <obj>) inserts object <obj> into list a at the specified <index>. Following the method call, a[<index>] is <obj>, and the remaining list elements are pushed to the right:

>>> a = ['foo', 'bar', 'baz', 'qux', 'quux', 'corge']

>>> a.insert(3, 3.14159)

>>> a[3]

3.14159

>>> a

['foo', 'bar', 'baz', 3.14159, 'qux', 'quux', 'corge']

**a.remove(<obj>)**

Removes an object from a list.

a.remove(<obj>) removes object <obj> from list a. If <obj> isn’t in a, an exception is raised:

>>> a = ['foo', 'bar', 'baz', 'qux', 'quux', 'corge']

>>> a.remove('baz')

>>> a

['foo', 'bar', 'qux', 'quux', 'corge']

>>> a.remove('Bark!')

Traceback (most recent call last):

File "<pyshell#13>", line 1, in <module>

a.remove('Bark!')

ValueError: list.remove(x): x not in list

**a.pop(index=-1)**

Removes an element from a list.

This method differs from .remove() in two ways:

1. You specify the index of the item to remove, rather than the object itself.
2. The method returns a value: the item that was removed.

a.pop() simply removes the last item in the list:

>>> a = ['foo', 'bar', 'baz', 'qux', 'quux', 'corge']

>>> a.pop()

'corge'

>>> a

['foo', 'bar', 'baz', 'qux', 'quux']

>>> a.pop()

'quux'

>>> a

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

If the optional <index> parameter is specified, the item at that index is removed and returned. <index> may be negative, as with string and list indexing:

>>> a = ['foo', 'bar', 'baz', 'qux', 'quux', 'corge']

>>> a.pop(1)

'bar'

>>> a

['foo', 'baz', 'qux', 'quux', 'corge']

>>> a.pop(-3)

'qux'

>>> a

['foo', 'baz', 'quux', 'corge']

<index> defaults to -1, so a.pop(-1) is equivalent to a.pop().

## Lists Are Dynamic

The last characteristic of a Python List is that lists are dynamic. You have seen many examples of this in the sections above. When items are added to a list, it grows as needed:

>>> a = ['foo', 'bar', 'baz', 'qux', 'quux', 'corge']

>>> a[2:2] = [1, 2, 3]

>>> a += [3.14159]

>>> a

['foo', 'bar', 1, 2, 3, 'baz', 'qux', 'quux', 'corge', 3.14159]

Similarly, a list shrinks to accommodate the removal of items:

>>> a = ['foo', 'bar', 'baz', 'qux', 'quux', 'corge']

>>> a[2:3] = []

>>> del a[0]

>>> a

['bar', 'qux', 'quux', 'corge']

# Python Tuples

Python provides another type that is an ordered collection of objects, called a tuple.

Pronunciation varies depending on whom you ask. Some pronounce it as though it were spelled “too-ple” (rhyming with “Mott the Hoople”), and others as though it were spelled “tup-ple” (rhyming with “supple”). My inclination is the latter, since it presumably derives from the same origin as “quintuple,” “sextuple,” “octuple,” and so on, and everyone I know pronounces these latter as though they rhymed with “supple.”

## Defining and Using Tuples

Tuples are identical to lists in all respects, except for the following properties:

* Tuples are defined by enclosing the elements in parentheses (()) instead of square brackets ([]).
* Tuples are immutable.

Here is a short example showing a tuple definition, indexing, and slicing:

>>> t = ('foo', 'bar', 'baz', 'qux', 'quux', 'corge')

>>> t

('foo', 'bar', 'baz', 'qux', 'quux', 'corge')

>>> t[0]

'foo'

>>> t[-1]

'corge'

>>> t[1::2]

('bar', 'qux', 'corge')

Never fear! Our favorite string and list reversal mechanism works for tuples as well:

>>> t[::-1]

('corge', 'quux', 'qux', 'baz', 'bar', 'foo')

**Note:** Even though tuples are defined using parentheses, you still index and slice tuples using square brackets, just as for strings and lists.

Everything you’ve learned about lists—they are ordered, they can contain arbitrary objects, they can be indexed and sliced, they can be nested—is true of tuples as well. But they can’t be modified:

>>> t = ('foo', 'bar', 'baz', 'qux', 'quux', 'corge')

>>> t[2] = 'Bark!'

Traceback (most recent call last):

File "<pyshell#65>", line 1, in <module>

t[2] = 'Bark!'

TypeError: 'tuple' object does not support item assignment

Why use a tuple instead of a list?

* Program execution is faster when manipulating a tuple than it is for the equivalent list. (This is probably not going to be noticeable when the list or tuple is small.)
* Sometimes you don’t want data to be modified. If the values in the collection are meant to remain constant for the life of the program, using a tuple instead of a list guards against accidental modification.
* There is another Python data type that you will encounter shortly called a dictionary, which requires as one of its components a value that is of an immutable type. A tuple can be used for this purpose, whereas a list can’t be.

In the REPL window, you can display the values of several objects simultaneously by entering them directly at the >>> prompt, separated by commas:

>>> a = 'foo'

>>> b = 42

>>> a, 3.14159, b

('foo', 3.14159, 42)

Python displays the response in parentheses because it is implicitly interpreting the input as a tuple.

There is one peculiarity regarding tuple definition that you should be aware of. There is no ambiguity when defining an empty tuple, nor one with two or more elements. Python knows you are defining a tuple:

>>> t = ()

>>> type(t)

<class 'tuple'>

>>> t = (1, 2)

>>> type(t)

<class 'tuple'>

>>> t = (1, 2, 3, 4, 5)

>>> type(t)

<class 'tuple'>

But what happens when you try to define a tuple with one item:

>>> t = (2)

>>> type(t)

<class 'int'>

*Doh!* Since parentheses are also used to define operator precedence in expressions, Python evaluates the expression (2) as simply the integer 2 and creates an int object. To tell Python that you really want to define a singleton tuple, include a trailing comma (,) just before the closing parenthesis:

>>> t = (2,)

>>> type(t)

<class 'tuple'>

>>> t[0]

2

>>> t[-1]

2

You probably won’t need to define a singleton tuple often, but there has to be a way.

When you display a singleton tuple, Python includes the comma, to remind you that it’s a tuple:

>>> print(t)

(2,)

## Tuple Assignment, Packing, and Unpacking

As you have already seen above, a literal tuple containing several items can be assigned to a single object:

>>> t = ('foo', 'bar', 'baz', 'qux')

When this occurs, it is as though the items in the tuple have been “packed” into the object:



>>> t

('foo', 'bar', 'baz', 'qux')

>>> t[0]

'foo'

>>> t[-1]

'qux'

If that “packed” object is subsequently assigned to a new tuple, the individual items are “unpacked” into the objects in the tuple:



>>> (s1, s2, s3, s4) = t

>>> s1

'foo'

>>> s2

'bar'

>>> s3

'baz'

>>> s4

'qux'

When unpacking, the number of variables on the left must match the number of values in the tuple:

>>> (s1, s2, s3) = t

Traceback (most recent call last):

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

(s1, s2, s3) = t

ValueError: too many values to unpack (expected 3)

>>> (s1, s2, s3, s4, s5) = t

Traceback (most recent call last):

File "<pyshell#17>", line 1, in <module>

(s1, s2, s3, s4, s5) = t

ValueError: not enough values to unpack (expected 5, got 4)

Packing and unpacking can be combined into one statement to make a compound assignment:

>>> (s1, s2, s3, s4) = ('foo', 'bar', 'baz', 'qux')

>>> s1

'foo'

>>> s2

'bar'

>>> s3

'baz'

>>> s4

'qux'

Again, the number of elements in the tuple on the left of the assignment must equal the number on the right:

>>> (s1, s2, s3, s4, s5) = ('foo', 'bar', 'baz', 'qux')

Traceback (most recent call last):

File "<pyshell#63>", line 1, in <module>

(s1, s2, s3, s4, s5) = ('foo', 'bar', 'baz', 'qux')

ValueError: not enough values to unpack (expected 5, got 4)

In assignments like this and a small handful of other situations, Python allows the parentheses that are usually used for denoting a tuple to be left out:

>>> t = 1, 2, 3

>>> t

(1, 2, 3)

>>> x1, x2, x3 = t

>>> x1, x2, x3

(1, 2, 3)

>>> x1, x2, x3 = 4, 5, 6

>>> x1, x2, x3

(4, 5, 6)

>>> t = 2,

>>> t

(2,)

It works the same whether the parentheses are included or not, so if you have any doubt as to whether they’re needed, go ahead and include them.

### Swap in Python using Tuple (assignment)

Tuple assignment allows for a curious bit of idiomatic Python. Frequently when programming, you have two variables whose values you need to swap. In most programming languages, it is necessary to store one of the values in a temporary variable while the swap occurs like this:

>>> a = 'foo'

>>> b = 'bar'

>>> a, b

('foo', 'bar')

>>># We need to define a temp variable to accomplish the swap.

>>> temp = a

>>> a = b

>>> b = temp

>>> a, b

('bar', 'foo')

In Python, the swap can be done with a single tuple assignment:

>>> a = 'foo'

>>> b = 'bar'

>>> a, b

('foo', 'bar')

>>># Magic time!

>>> a, b = b, a

>>> a, b

('bar', 'foo')

As anyone who has ever had to swap values using a temporary variable knows, being able to do it this way in Python is the pinnacle of modern technological achievement. It will never get better than this.

# 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()))

# Python Sets

A rigorous definition of a set can be abstract and difficult to grasp. Practically though, a set can be thought of simply as a well-defined collection of distinct objects, typically called **elements** or **members**.

Grouping objects into a set can be useful in programming as well, and Python provides a built-in set type to do so. Sets are distinguished from other object types by the unique operations that can be performed on them.

## Defining a Set

Python’s built-in set type has the following characteristics:

* Sets are unordered.
* Set elements are unique. Duplicate elements are not allowed.
* A set itself may be modified, but the elements contained in the set must be of an immutable type.

Let’s see what all that means, and how you can work with sets in Python.

A set can be created in two ways. First, you can define a set with the built-in set() function:

x = set(<iter>)

In this case, the argument <iter> is an iterable—again, for the moment, think list or tuple—that generates the list of objects to be included in the set. This is analogous to the <iter> argument given to the .extend() list method:

>>> x = set(['foo', 'bar', 'baz', 'foo', 'qux'])

>>> x

{'qux', 'foo', 'bar', 'baz'}

>>> x = set(('foo', 'bar', 'baz', 'foo', 'qux'))

>>> x

{'qux', 'foo', 'bar', 'baz'}

Strings are also iterable, so a string can be passed to set() as well. You have already seen that list(s) generates a list of the characters in the string s. Similarly, set(s) generates a set of the characters in s:

>>> s = 'quux'

>>> list(s)

['q', 'u', 'u', 'x']

>>> set(s)

{'x', 'u', 'q'}

You can see that the resulting sets are unordered: the original order, as specified in the definition, is not necessarily preserved. Additionally, duplicate values are only represented in the set once, as with the string 'foo' in the first two examples and the letter 'u' in the third.

Alternately, a set can be defined with curly braces ({}):

x = {<obj>, <obj>, ..., <obj>}

When a set is defined this way, each <obj> becomes a distinct element of the set, even if it is an iterable. This behavior is similar to that of the .append() list method.

Thus, the sets shown above can also be defined like this:

>>> x = {'foo', 'bar', 'baz', 'foo', 'qux'}

>>> x

{'qux', 'foo', 'bar', 'baz'}

>>> x = {'q', 'u', 'u', 'x'}

>>> x

{'x', 'q', 'u'}

Observe the difference between these two set definitions:

>>> {'foo'}

{'foo'}

>>> set('foo')

{'f', 'o'}

A set can be empty. However, recall that Python interprets empty curly braces ({}) as an empty dictionary, so the only way to define an empty set is with the set() function:

>>> x = set()

>>> type(x)

<class 'set'>

>>> x

set()

>>> x = {}

>>> type(x)

<class 'dict'>

An empty set is falsy in a Boolean context:

>>> x = set()

>>> bool(x)

False

>>> x or 1

1

>>> x and 1

set()

You might think the most intuitive sets would contain similar objects—for example, even numbers or surnames:

>>> s1 = {2, 4, 6, 8, 10}

>>> s2 = {'Smith', 'McArthur', 'Wilson', 'Johansson'}

Python does not require this, though. The elements in a set can be objects of different types:

>>> x = {42, 'foo', 3.14159, None}

>>> x

{None, 'foo', 42, 3.14159}

Don’t forget that set elements must be immutable. For example, a tuple may be included in a set:

>>> x = {42, 'foo', (1, 2, 3), 3.14159}

>>> x

{42, 'foo', 3.14159, (1, 2, 3)}

But lists and dictionaries are mutable, so they can’t be set elements:

>>> a = [1, 2, 3]

>>> {a}

Traceback (most recent call last):

File "<pyshell#70>", line 1, in <module>

{a}

TypeError: unhashable type: 'list'

>>> d = {'a': 1, 'b': 2}

>>> {d}

Traceback (most recent call last):

File "<pyshell#72>", line 1, in <module>

{d}

TypeError: unhashable type: 'dict'

## Set Size and Membership

The len() function returns the number of elements in a set, and the in and not in operators can be used to test for membership:

>>> x = {'foo', 'bar', 'baz'}

>>> len(x)

3

>>> 'bar' in x

True

>>> 'qux' in x

False

## Operating on a Set

Many of the operations that can be used for Python’s other composite data types don’t make sense for sets. For example, sets can’t be indexed or sliced. However, Python provides a whole host of operations on set objects that generally mimic the operations that are defined for mathematical sets.

### Operators vs. Methods

Most, though not quite all, set operations in Python can be performed in two different ways: by operator or by method. Let’s take a look at how these operators and methods work, using set union as an example.

Given two sets, x1 and x2, the union of x1 and x2 is a set consisting of all elements in either set.

Consider these two sets:

x1 = {'foo', 'bar', 'baz'}

x2 = {'baz', 'qux', 'quux'}

The union of x1 and x2 is {'foo', 'bar', 'baz', 'qux', 'quux'}.

**Note:** Notice that the element 'baz', which appears in both x1 and x2, appears only once in the union. Sets never contain duplicate values.

In Python, set union can be performed with the | operator:

>>> x1 = {'foo', 'bar', 'baz'}

>>> x2 = {'baz', 'qux', 'quux'}

>>> x1 | x2

{'baz', 'quux', 'qux', 'bar', 'foo'}

Set union can also be obtained with the .union() method. The method is invoked on one of the sets, and the other is passed as an argument:

>>> x1.union(x2)

{'baz', 'quux', 'qux', 'bar', 'foo'}

The way they are used in the examples above, the operator and method behave identically. But there is a subtle difference between them. When you use the | operator, both operands must be sets. The .union() method, on the other hand, will take any iterable as an argument, convert it to a set, and then perform the union.

Observe the difference between these two statements:

>>> x1 | ('baz', 'qux', 'quux')

Traceback (most recent call last):

File "<pyshell#43>", line 1, in <module>

x1 | ('baz', 'qux', 'quux')

TypeError: unsupported operand type(s) for |: 'set' and 'tuple'

>>> x1.union(('baz', 'qux', 'quux'))

{'baz', 'quux', 'qux', 'bar', 'foo'}

Both attempt to compute the union of x1 and the tuple ('baz', 'qux', 'quux'). This fails with the | operator but succeeds with the .union() method.

### Available Operators and Methods

Below is a list of the set operations available in Python. Some are performed by operator, some by method, and some by both. The principle outlined above generally applies: where a set is expected, methods will typically accept any iterable as an argument, but operators require actual sets as operands.

**Union**

x1.union(x2[, x3 ...])

x1 | x2 [| x3 ...]

Compute the union of two or more sets.

x1.union(x2) and x1 | x2 both return the set of all elements in either x1 or x2:

>>> x1 = {'foo', 'bar', 'baz'}

>>> x2 = {'baz', 'qux', 'quux'}

>>> x1.union(x2)

{'foo', 'qux', 'quux', 'baz', 'bar'}

>>> x1 | x2

{'foo', 'qux', 'quux', 'baz', 'bar'}

More than two sets may be specified with either the operator or the method:

>>> a = {1, 2, 3, 4}

>>> b = {2, 3, 4, 5}

>>> c = {3, 4, 5, 6}

>>> d = {4, 5, 6, 7}

>>> a.union(b, c, d)

{1, 2, 3, 4, 5, 6, 7}

>>> a | b | c | d

{1, 2, 3, 4, 5, 6, 7}

The resulting set contains all elements that are present in any of the specified sets.

**Intersection**

x1.intersection(x2[, x3 ...])

x1 & x2 [& x3 ...]

Compute the intersection of two or more sets.

x1.intersection(x2) and x1 & x2 return the set of elements common to both x1 and x2:

>>> x1 = {'foo', 'bar', 'baz'}

>>> x2 = {'baz', 'qux', 'quux'}

>>> x1.intersection(x2)

{'baz'}

>>> x1 & x2

{'baz'}

You can specify multiple sets with the intersection method and operator, just like you can with set union:

>>> a = {1, 2, 3, 4}

>>> b = {2, 3, 4, 5}

>>> c = {3, 4, 5, 6}

>>> d = {4, 5, 6, 7}

>>> a.intersection(b, c, d)

{4}

>>> a & b & c & d

{4}

The resulting set contains only elements that are present in all of the specified sets.

**Difference**

x1.difference(x2[, x3 ...])

x1 - x2 [- x3 ...]

Compute the difference between two or more sets.

x1.difference(x2) and x1 - x2 return the set of all elements that are in x1 but not in x2:

>>> x1 = {'foo', 'bar', 'baz'}

>>> x2 = {'baz', 'qux', 'quux'}

>>> x1.difference(x2)

{'foo', 'bar'}

>>> x1 - x2

{'foo', 'bar'}

Another way to think of this is that x1.difference(x2) and x1 - x2 return the set that results when any elements in x2 are removed or subtracted from x1.

Once again, you can specify more than two sets:

>>> a = {1, 2, 3, 30, 300}

>>> b = {10, 20, 30, 40}

>>> c = {100, 200, 300, 400}

>>> a.difference(b, c)

{1, 2, 3}

>>> a - b - c

{1, 2, 3}

When multiple sets are specified, the operation is performed from left to right. In the example above, a - b is computed first, resulting in {1, 2, 3, 300}. Then c is subtracted from that set, leaving {1, 2, 3}:



**Subset**

x1.issubset(x2)

x1 <= x2

Determine whether one set is a subset of the other.

In set theory, a set x1 is considered a subset of another set x2 if every element of x1 is in x2.

x1.issubset(x2) and x1 <= x2 return True if x1 is a subset of x2:

>>> x1 = {'foo', 'bar', 'baz'}

>>> x1.issubset({'foo', 'bar', 'baz', 'qux', 'quux'})

True

>>> x2 = {'baz', 'qux', 'quux'}

>>> x1 <= x2

False

A set is considered to be a subset of itself:

>>> x = {1, 2, 3, 4, 5}

>>> x.issubset(x)

True

>>> x <= x

True

It seems strange, perhaps. But it fits the definition—every element of x is in x.

**Proper Subset**

x1 < x2

Determines whether one set is a proper subset of the other.

A proper subset is the same as a subset, except that the sets can’t be identical. A set x1 is considered a proper subset of another set x2 if every element of x1 is in x2, and x1 and x2 are not equal.

x1 < x2 returns True if x1 is a proper subset of x2:

>>> x1 = {'foo', 'bar'}

>>> x2 = {'foo', 'bar', 'baz'}

>>> x1 < x2

True

>>> x1 = {'foo', 'bar', 'baz'}

>>> x2 = {'foo', 'bar', 'baz'}

>>> x1 < x2

False

While a set is considered a subset of itself, it is not a proper subset of itself:

>>> x = {1, 2, 3, 4, 5}

>>> x <= x

True

>>> x < x

False

**Note:** The < operator is the only way to test whether a set is a proper subset. There is no corresponding method.

**Superset**

x1.issuperset(x2)

x1 >= x2

Determine whether one set is a superset of the other.

A superset is the reverse of a subset. A set x1 is considered a superset of another set x2 if x1 contains every element of x2.

x1.issuperset(x2) and x1 >= x2 return True if x1 is a superset of x2:

>>> x1 = {'foo', 'bar', 'baz'}

>>> x1.issuperset({'foo', 'bar'})

True

>>> x2 = {'baz', 'qux', 'quux'}

>>> x1 >= x2

False

You have already seen that a set is considered a subset of itself. A set is also considered a superset of itself:

>>>

>>> x = {1, 2, 3, 4, 5}

>>> x.issuperset(x)

True

>>> x >= x

True

**Proper Superset**

x1 > x2

Determines whether one set is a proper superset of the other.

A proper superset is the same as a superset, except that the sets can’t be identical. A set x1 is considered a proper superset of another set x2 if x1 contains every element of x2, and x1 and x2 are not equal.

x1 > x2 returns True if x1 is a proper superset of x2:

>>>

>>> x1 = {'foo', 'bar', 'baz'}

>>> x2 = {'foo', 'bar'}

>>> x1 > x2

True

>>> x1 = {'foo', 'bar', 'baz'}

>>> x2 = {'foo', 'bar', 'baz'}

>>> x1 > x2

False

A set is not a proper superset of itself:

>>> x = {1, 2, 3, 4, 5}

>>> x > x

False

**Note:** The > operator is the only way to test whether a set is a proper superset. There is no corresponding method.

## Modifying a Set

Although the elements contained in a set must be of immutable type, sets themselves can be modified. Like the operations above, there are a mix of operators and methods that can be used to change the contents of a set.

### Augmented Assignment Operators and Methods

Each of the union, intersection, difference, and symmetric difference operators listed above has an augmented assignment form that can be used to modify a set. For each, there is a corresponding method as well.

**Union Update**

x1.update(x2[, x3 ...])

x1 |= x2 [| x3 ...]

Modify a set by union.

x1.update(x2) and x1 |= x2 add to x1 any elements in x2 that x1 does not already have:

>>> x1 = {'foo', 'bar', 'baz'}

>>> x2 = {'foo', 'baz', 'qux'}

>>> x1 |= x2

>>> x1

{'qux', 'foo', 'bar', 'baz'}

>>> x1.update(['corge', 'garply'])

>>> x1

{'qux', 'corge', 'garply', 'foo', 'bar', 'baz'}

**Intersection Update**

x1.intersection\_update(x2[, x3 ...])

x1 &= x2 [& x3 ...]

Modify a set by intersection.

x1.intersection\_update(x2) and x1 &= x2 update x1, retaining only elements found in both x1 and x2:

>>> x1 = {'foo', 'bar', 'baz'}

>>> x2 = {'foo', 'baz', 'qux'}

>>> x1 &= x2

>>> x1

{'foo', 'baz'}

>>> x1.intersection\_update(['baz', 'qux'])

>>> x1

{'baz'}

**Difference Update**

x1.difference\_update(x2[, x3 ...])

x1 -= x2 [| x3 ...]

Modify a set by difference.

x1.difference\_update(x2) and x1 -= x2 update x1, removing elements found in x2:

>>> x1 = {'foo', 'bar', 'baz'}

>>> x2 = {'foo', 'baz', 'qux'}

>>> x1 -= x2

>>> x1

{'bar'}

>>> x1.difference\_update(['foo', 'bar', 'qux'])

>>> x1

set()

### Other Methods For Modifying Sets

Aside from the augmented operators above, Python supports several additional methods that modify sets.

**Add**

x.add(<elem>)

Adds an element to a set.

x.add(<elem>) adds <elem>, which must be a single immutable object, to x:

>>> x = {'foo', 'bar', 'baz'}

>>> x.add('qux')

>>> x

{'bar', 'baz', 'foo', 'qux'}

**Remove**

x.remove(<elem>)

Removes an element from a set.

x.remove(<elem>) removes <elem> from x. Python raises an exception if <elem> is not in x:

>>> x = {'foo', 'bar', 'baz'}

>>> x.remove('baz')

>>> x

{'bar', 'foo'}

>>> x.remove('qux')

Traceback (most recent call last):

File "<pyshell#58>", line 1, in <module>

x.remove('qux')

KeyError: 'qux'

**Discard**

x.discard(<elem>)

Removes an element from a set.

x.discard(<elem>) also removes <elem> from x. However, if <elem> is not in x, this method quietly does nothing instead of raising an exception:

>>> x = {'foo', 'bar', 'baz'}

>>> x.discard('baz')

>>> x

{'bar', 'foo'}

>>> x.discard('qux')

>>> x

{'bar', 'foo'}

**Pop**

x.pop()

Removes a random element from a set.

x.pop() removes and returns an arbitrarily chosen element from x. If x is empty, x.pop() raises an exception:

>>> x = {'foo', 'bar', 'baz'}

>>> x.pop()

'bar'

>>> x

{'baz', 'foo'}

>>> x.pop()

'baz'

>>> x

{'foo'}

>>> x.pop()

'foo'

>>> x

set()

>>> x.pop()

Traceback (most recent call last):

File "<pyshell#82>", line 1, in <module>

x.pop()

KeyError: 'pop from an empty set'

**Clear**

x.clear()

Clears a set.

x.clear() removes all elements from x:

>>> x = {'foo', 'bar', 'baz'}

>>> x

{'foo', 'bar', 'baz'}

>>>

>>> x.clear()

>>> x

set()

## Frozen Sets

Python provides another built-in type called a **frozenset**, which is in all respects exactly like a set, except that a frozenset is immutable. You can perform non-modifying operations on a frozenset:

>>> x = frozenset(['foo', 'bar', 'baz'])

>>> x

frozenset({'foo', 'baz', 'bar'})

>>> len(x)

3

>>> x & {'baz', 'qux', 'quux'}

frozenset({'baz'})

But methods that attempt to modify a frozenset fail:

>>> x = frozenset(['foo', 'bar', 'baz'])

>>> x.add('qux')

Traceback (most recent call last):

File "<pyshell#127>", line 1, in <module>

x.add('qux')

AttributeError: 'frozenset' object has no attribute 'add'

>>> x.pop()

Traceback (most recent call last):

File "<pyshell#129>", line 1, in <module>

x.pop()

AttributeError: 'frozenset' object has no attribute 'pop'

>>> x.clear()

Traceback (most recent call last):

File "<pyshell#131>", line 1, in <module>

x.clear()

AttributeError: 'frozenset' object has no attribute 'clear'

>>> x

frozenset({'foo', 'bar', 'baz'})

# 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.

# Python File Handling

Python too supports file handling and allows users to handle files i.e., to read and write files, along with many other file handling options, to operate on files. The concept of file handling has stretched over various other languages, but the implementation is either complicated or lengthy, but alike other concepts of Python, this concept here is also easy and short. Python treats file differently as text or binary and this is important. Each line of code includes a sequence of characters and they form text file. Each line of a file is terminated with a special character, called the EOL or End of Line characters like comma {,} or newline character. It ends the current line and tells the interpreter a new one has begun. Let’s start with Reading and Writing files.

## open()

We use **open ()** function in Python to open a file in read or write mode. As explained above, open ( ) will return a file object. To return a file object we use **open()** function along with two arguments, that accepts file name and the mode, whether to read or write. So, the syntax being: **open(filename, mode)**. There are three kinds of mode, that Python provides and how files can be opened:

* “**r**“, for reading.
* “**w**“, for writing.
* “**a**“, for appending.
* “**r+**“, for both reading and writing.
* “**x**”, Creates a new file. If file already exists, the operation fails.
* “**t**”, This is the default mode. It opens in text mode.
* “**b**”, This opens in binary mode.
* “**+**”, This will open a file for reading and writing (updating)

One must keep in mind that the mode argument is not mandatory. If not passed, then Python will assume it to be “**r**” by default. Let’s look at this program and try to analyze how the read mode works:

# a file named "geek", will be opened with the reading mode.

file = open('geek.txt', 'r')

# This will print every line one by one in the file

for each in file:

    print (each)

The open command will open the file in the read mode and the for loop will print each line present in the file.

## read()

There is more than one way to read a file in Python. If you need to extract a string that contains all characters in the file then we can use **file.read()**. The full code would work like this:

|  |
| --- |
| # Python code to illustrate read() mode  file = open("file.text", "r")  print (file.read()) |

Another way to read a file is to call a certain number of characters like in the following code the interpreter will read the first five characters of stored data and return it as a string:

|  |
| --- |
| # Python code to illustrate read() mode character wise  file = open("file.txt", "r")  print (file.read(5)) |

def main():

f=open("guru99.txt", "r")

if f.mode == 'r':

contents = f.read()

print(contents)

f.close()

if \_\_name\_\_ == “\_\_main\_\_”:

main()

## readlines()

def main():

f=open("guru99.txt", "r")

lines = f.readlines()

for line in lines:

print(line)

f.close()

if \_\_name\_\_ == “\_\_main\_\_”:

main()

## **write()**

Let’s see how to create a file and how write mode works. To manipulate the file, write the following in your Python environment:

# Python code to create a file

file = open('geek.txt','w')

file.write("This is the write command")

file.write("It allows us to write in a particular file")

file.close()

The close() command terminates all the resources in use and frees the system of this particular program.

def main():

f = open("guru99.txt","w+")

for i in range(10):

f.write("This is line %d\r\n" % (i+1))

f.close()

if \_\_name\_\_ == “\_\_main\_\_”:

main()

## append()

Let’s see how the append mode works:

# Python code to illustrate append() mode

file = open('geek.txt','a')

file.write("This will add this line")

file.close()

def main():

f=open("guru99.txt", "a+")

for i in range(2):

f.write("Appended line %d\r\n" % (i+1))

f.close()

if \_\_name\_\_ == “\_\_main\_\_”:

main()

There are also various other commands in file handling that is used to handle various tasks like:

rstrip(): This function strips each line of a file off spaces from the right-hand side.

lstrip(): This function strips each line of a file off spaces from the left-hand side.

It is designed to provide much cleaner syntax and exceptions handling when you are working with code. That explains why it’s good practice to use them with a statement where applicable. This is helpful because using this method any files opened will be closed automatically after one is done, so auto-cleanup.  
**Example:**

# Python code to illustrate with()

with open("file.txt") as file:

    data = file.read()

# do something with data

## **with()**

We can also use write function along with with() function:

# Python code to illustrate with() alongwith write()

with open("file.txt", "w") as f:

    f.write("Hello World!!!")

## **split()**

We can also split lines using file handling in Python. This splits the variable when space is encountered. You can also split using any characters as we wish. Here is the code:

# Python code to illustrate split() function

with open("file.text", "r") as file:

    data = file.readlines()

    for line in data:

        word = line.split()

        print (word)

# Logging in Python

Logging is a very useful tool in a programmer’s toolbox. It can help you develop a better understanding of the flow of a program and discover scenarios that you might not even have thought of while developing.

Logs provide developers with an extra set of eyes that are constantly looking at the flow that an application is going through. They can store information, like which user or IP accessed the application. If an error occurs, then they can provide more insights than a stack trace by telling you what the state of the program was before it arrived at the line of code where the error occurred.

By logging useful data from the right places, you can not only debug errors easily but also use the data to analyze the performance of the application to plan for scaling or look at usage patterns to plan for marketing.

Python provides a logging system as a part of its standard library, so you can quickly add logging to your application.

## The Logging Module

The logging module in Python is a ready-to-use and powerful module that is designed to meet the needs of beginners as well as enterprise teams. It is used by most of the third-party Python libraries, so you can integrate your log messages with the ones from those libraries to produce a homogeneous log for your application.

Adding logging to your Python program is as easy as this:

import logging

With the logging module imported, you can use something called a “logger” to log messages that you want to see. By default, there are 5 standard levels indicating the severity of events. Each has a corresponding method that can be used to log events at that level of severity. The defined levels, in order of increasing severity, are the following:

* DEBUG
* INFO
* WARNING
* ERROR
* CRITICAL

The logging module provides you with a default logger that allows you to get started without needing to do much configuration. The corresponding methods for each level can be called as shown in the following example:

import logging

logging.debug('This is a debug message')

logging.info('This is an info message')

logging.warning('This is a warning message')

logging.error('This is an error message')

logging.critical('This is a critical message')

The output of the above program would look like this:

WARNING:root:This is a warning message

ERROR:root:This is an error message

CRITICAL:root:This is a critical message

Notice that the debug() and info() messages didn’t get logged. This is because, by default, the logging module logs the messages with a severity level of WARNING or above. You can change that by configuring the logging module to log events of all levels if you want. You can also define your own severity levels by changing configurations, but it is generally not recommended as it can cause confusion with logs of some third-party libraries that you might be using.

## Basic Configurations

You can use the basicConfig(\*\**kwargs*) method to configure the logging:

Some of the commonly used parameters for basicConfig() are the following:

* level: The root logger will be set to the specified severity level.
* filename: This specifies the file.
* filemode: If filename is given, the file is opened in this mode. The default is a, which means append.
* format: This is the format of the log message.

By using the level parameter, you can set what level of log messages you want to record. This can be done by passing one of the constants available in the class, and this would enable all logging calls at or above that level to be logged. Here’s an example:

import logging

logging.basicConfig(level=logging.DEBUG)

logging.debug('This will get logged')

OUTPUT:

DEBUG:root:This will get logged

All events at or above DEBUG level will now get logged.

Similarly, for logging to a file rather than the console, filename and filemode can be used, and you can decide the format of the message using format. The following example shows the usage of all three:

import logging

logging.basicConfig(filename='app.log', filemode='w', format='%(name)s - %(levelname)s - %(message)s')

logging.warning('This will get logged to a file')

OUTPUT:

root - ERROR - This will get logged to a file

The message will look like this but will be written to a file named app.log instead of the console. The filemode is set to w, which means the log file is opened in “write mode” each time basicConfig() is called, and each run of the program will rewrite the file. The default configuration for filemode is a, which is append.

You can customize the root logger even further by using more parameters for basicConfig(), which can be found [here](https://docs.python.org/3/library/logging.html#logging.basicConfig) (<https://docs.python.org/3/library/logging.html#logging.basicConfig>).

It should be noted that calling basicConfig() to configure the root logger works only if the root logger has not been configured before. **Basically, this function can only be called once.**

debug(), info(), warning(), error(), and critical() also call basicConfig() without arguments automatically if it has not been called before. This means that after the first time one of the above functions is called, you can no longer configure the root logger because they would have called the basicConfig() function internally.

The default setting in basicConfig() is to set the logger to write to the console in the following format:

ERROR:root:This is an error message

## Formatting the Output

While you can pass any variable that can be represented as a string from your program as a message to your logs, there are some basic elements that are already a part of the LogRecord and can be easily added to the output format. If you want to log the process ID along with the level and message, you can do something like this:

import logging

logging.basicConfig(format='%(process)d-%(levelname)s-%(message)s')

logging.warning('This is a Warning')

OUTPUT:

18472-WARNING-This is a Warning

format can take a string with LogRecord attributes in any arrangement you like. The entire list of available attributes can be found [here](https://docs.python.org/3/library/logging.html#logrecord-attributes) (<https://docs.python.org/3/library/logging.html#logrecord-attributes>).

Here’s another example where you can add the date and time info:

import logging

logging.basicConfig(format='%(asctime)s - %(message)s', level=logging.INFO)

logging.info('Admin logged in')

OUTPUT:

2018-07-11 20:12:06,288 - Admin logged in

%(asctime)s adds the time of creation of the LogRecord. The format can be changed using the datefmt attribute, which uses the same formatting language as the formatting functions in the datetime module, such as time.strftime():

import logging

logging.basicConfig(format='%(asctime)s - %(message)s', datefmt='%d-%b-%y %H:%M:%S')

logging.warning('Admin logged out')

OUTPUT:

12-Jul-18 20:53:19 - Admin logged out

### Logging Variable Data

In most cases, you would want to include dynamic information from your application in the logs. You have seen that the logging methods take a string as an argument, and it might seem natural to format a string with variable data in a separate line and pass it to the log method. But this can actually be done directly by using a format string for the message and appending the variable data as arguments. Here’s an example:

import logging

name = 'John'

logging.error('%s raised an error', name)

OUTPUT:

ERROR:root:John raised an error

The arguments passed to the method would be included as variable data in the message.

While you can use any formatting style, the f-strings introduced in Python 3.6 are an awesome way to format strings as they can help keep the formatting short and easy to read:

import logging

name = 'John'

logging.error(f'{name} raised an error')

OUTPUT:

ERROR:root:John raised an error

### Capturing Stack Traces

The logging module also allows you to capture the full stack traces in an application. Exception information can be captured if the exc\_info parameter is passed as True, and the logging functions are called like this:

import logging

a = 5

b = 0

try:

c = a / b

except Exception as e:

logging.error("Exception occurred", exc\_info=True)

OUTPUT:

ERROR:root:Exception occurred

Traceback (most recent call last):

File "exceptions.py", line 6, in <module>

c = a / b

ZeroDivisionError: division by zero

[Finished in 0.2s]

If exc\_info is not set to True, the output of the above program would not tell us anything about the exception, which, in a real-world scenario, might not be as simple as a ZeroDivisionError. Imagine trying to debug an error in a complicated codebase with a log that shows only this:

ERROR:root:Exception occurred

Here’s a quick tip: if you’re logging from an exception handler, use the logging.exception() method, which logs a message with level ERROR and adds exception information to the message. To put it more simply, calling logging.exception() is like calling logging.error(exc\_info=True). But since this method always dumps exception information, it should only be called from an exception handler. Take a look at this example:

import logging

a = 5

b = 0

try:

c = a / b

except Exception as e:

logging.exception("Exception occurred")

ERROR:root:Exception occurred

Traceback (most recent call last):

File "exceptions.py", line 6, in <module>

c = a / b

ZeroDivisionError: division by zero

[Finished in 0.2s]

Using logging.exception() would show a log at the level of ERROR. If you don’t want that, you can call any of the other logging methods from debug() to critical() and pass the exc\_info parameter as True.

## Classes and Functions

So far, we have seen the default logger named root, which is used by the logging module whenever its functions are called directly like this: logging.debug(). You can (and should) define your own logger by creating an object of the Logger class, especially if your application has multiple modules. Let’s have a look at some of the classes and functions in the module.

The most commonly used classes defined in the logging module are the following:

* **Logger:** This is the class whose objects will be used in the application code directly to call the functions.
* **LogRecord:** Loggers automatically create LogRecord objects that have all the information related to the event being logged, like the name of the logger, the function, the line number, the message, and more.
* **Handler:** Handlers send the LogRecord to the required output destination, like the console or a file. Handler is a base for subclasses like StreamHandler, FileHandler, SMTPHandler, HTTPHandler, and more. These subclasses send the logging outputs to corresponding destinations, like sys.stdout or a disk file.
* **Formatter:** This is where you specify the format of the output by specifying a string format that lists out the attributes that the output should contain.

Out of these, we mostly deal with the objects of the Logger class, which are instantiated using the module-level function logging.getLogger(name). Multiple calls to getLogger() with the same name will return a reference to the same Logger object, which saves us from passing the logger objects to every part where it’s needed. Here’s an example:

import logging

logger = logging.getLogger('example\_logger')

logger.warning('This is a warning')

This is a warning

This creates a custom logger named example\_logger, but unlike the root logger, the name of a custom logger is not part of the default output format and has to be added to the configuration. Configuring it to a format to show the name of the logger would give an output like this:

WARNING:example\_logger:This is a warning

Again, unlike the root logger, a custom logger can’t be configured using basicConfig(). You have to configure it using Handlers and Formatters.

## Using Handlers

Handlers come into the picture when you want to configure your own loggers and send the logs to multiple places when they are generated. Handlers send the log messages to configured destinations like the standard output stream or a file or over HTTP or to your email via SMTP.

A logger that you create can have more than one handler, which means you can set it up to be saved to a log file and also send it over email.

Like loggers, you can also set the severity level in handlers. This is useful if you want to set multiple handlers for the same logger but want different severity levels for each of them. For example, you may want logs with level WARNING and above to be logged to the console, but everything with level ERROR and above should also be saved to a file. Here’s a program that does that:

# logging\_example.py

import logging

# Create a custom logger

logger = logging.getLogger(\_\_name\_\_)

# Create handlers

c\_handler = logging.StreamHandler()

f\_handler = logging.FileHandler('file.log')

c\_handler.setLevel(logging.WARNING)

f\_handler.setLevel(logging.ERROR)

# Create formatters and add it to handlers

c\_format = logging.Formatter('%(name)s - %(levelname)s - %(message)s')

f\_format = logging.Formatter('%(asctime)s - %(name)s - %(levelname)s - %(message)s')

c\_handler.setFormatter(c\_format)

f\_handler.setFormatter(f\_format)

# Add handlers to the logger

logger.addHandler(c\_handler)

logger.addHandler(f\_handler)

logger.warning('This is a warning')

logger.error('This is an error')

\_\_main\_\_ - WARNING - This is a warning

\_\_main\_\_ - ERROR - This is an error

Here, logger.warning() is creating a LogRecord that holds all the information of the event and passing it to all the Handlers that it has: c\_handler and f\_handler.

c\_handler is a StreamHandler with level WARNING and takes the info from the LogRecord to generate an output in the format specified and prints it to the console. f\_handler is a FileHandler with level ERROR, and it ignores this LogRecord as its level is WARNING.

When logger.error() is called, c\_handler behaves exactly as before, and f\_handler gets a LogRecord at the level of ERROR, so it proceeds to generate an output just like c\_handler, but instead of printing it to console, it writes it to the specified file in this format:

2018-08-03 16:12:21,723 - \_\_main\_\_ - ERROR - This is an error

The name of the logger corresponding to the \_\_name\_\_ variable is logged as \_\_main\_\_, which is the name Python assigns to the module where execution starts. If this file is imported by some other module, then the \_\_name\_\_ variable would correspond to its name logging\_example. Here’s how it would look:

# run.py

import logging\_example

logging\_example - WARNING - This is a warning

logging\_example - ERROR - This is an error

# Python Collections

The Python collections module contains a number of specialized data structures that you can use in addition to—or as an alternative to—Python’s built-in containers. Because collections is a module, we have to import it into our program. However it is built into Python, so we do not need to import secondary libraries.

The four most commonly used data structures from the collections module are as follows:

* Counter
* namedtuple
* defaultdict
* ChainMap

## Counter

Counter() is a subclass of the dictionary object and can be used to count hashable objects. The Counter() function takes in an iterable as an argument and returns a dictionary.

So, let’s say that we have a list of sandwich orders for January and want to know how many BLT sandwiches we sold during that month. We could use the Counter() function to do this.

Here’s an example of the code we would use:

from collections import Counter

sandwich\_sales = ["BLT", "Egg Mayo", "Ham", "Ham", "Ham", "Cheese", "BLT", "Cheese"]

our\_counter = Counter(sandwich\_sales) ‘

print(our\_counter["BLT"])

# A Python program to show different

# ways to create Counter

from collections import Counter

# With sequence of items

print(Counter(['B','B','A','B','C','A','B',

               'B','A','C']))

# with dictionary

print(Counter({'A':3, 'B':5, 'C':2}))

# with keyword arguments

print(Counter(A=3, B=5, C=2))

## namedtuple

The namedtuple() method returns a tuple with names for each position in the tuple. When you’re working with a standard tuple, the only way you can access individual values is by referencing the tuple’s index numbers. If you’re working with a big tuple, this can quickly get confusing.

Here’s an example of using the namedtuple() method to store a sandwich’s name and price:

from collections import namedtuple

Sandwich = namedtuple("Sandwich", "name, price")

first\_sandwich = Sandwich("Chicken Teriyaki", "$3.00")

print(first\_sandwich.price)

You can also create a namedtuple() using a list. Here’s an example:

second\_sandwich = Sandwich.\_make(["Spicy Italian", "$3.75"])

print(second\_sandwich.name)

Our program returns: Spicy Italian. In this example, we use \_make in addition to our Sandwich item to denote that we want to turn our list into a namedtuple().

# Python code to demonstrate namedtuple()

from collections import namedtuple

# Declaring namedtuple()

Student = namedtuple('Student',['name','age','DOB'])

# Adding values

S = Student('Nandini','19','2541997')

# Access using index

print ("The Student age using index is : ",end ="")

print (S[1])

# Access using name

print ("The Student name using keyname is : ",end ="")

print (S.name)

## defaultdict

The defaultdict() method can be used to create a Python dictionary that does not throw a KeyError when you try to access an object that does not exist. Instead, if you reference an object that does not exist, the dictionary will return a predefined data type.

Here’s an example that uses the defaultdict() method to declare a dictionary that will return an str if we reference a non-existent object:

from collections import defaultdict

sandwiches = defaultdict(str)

sandwiches[0] = "Ham and Cheese"

sandwiches[1] = "BLT"

print(sandwiches[1])

print(sandwiches[2])

In the above example, we created a dictionary with values at index positions 0 and 1. When we print out sandwiches[1], we can see that our dictionary stored our values. However, when we try to print out the item associated with the index value 2, our program returns a blank line because there is no value assigned to that index.

In a standard dictionary, our program would return a KeyError. However, because we used defaultdict, our program instead returns the data type we specified when we created the dictionary. In the above example, we stated that any invalid key should return an str, but we could have coded it to return an integer or any other valid data type.

This function can be useful when you’re working with a dictionary to perform an operation on multiple items but the operation may not work on each item. Instead of causing your program to return an error, the defaultdict() will return a default value and keep running.

# Python program to demonstrate defaultdict

from collections import defaultdict

# Defining the dict

d = defaultdict(int)

L = [1, 2, 3, 4, 2, 4, 1, 2]

# Iterate through the list

# for keeping the count

for i in L:

    # The default value is 0

    # so there is no need to

    # enter the key first

    d[i] += 1

print(d)

## ChainMap

The ChainMap() method is used to combine two or more dictionaries; it returns a list of dictionaries. For example, let’s say that we have two menus—a standard menu and a secret menu—that we want to merge into one big menu. In order to do this, we could use the ChainMap() function.

Here’s an example of using ChainMap() to merge our standard and secret menus:

from collections import ChainMap

standard\_menu = { "BLT": "$3.05", "Roast Beef": "$3.55", "Cheese": "$2.85", "Shrimp": "$3.55", "Ham": "$2.85" }

secret\_menu = { "Steak": "$3.60", "Tuna Special": "$3.20", "Turkey Club": "$3.20" }

menu = ChainMap(standard\_menu, secret\_menu)

print(menu)

Our code returns a ChainMap object that merged our two menus together, as follows:

ChainMap({'BLT': '$3.05', 'Roast Beef': '$3.55', 'Cheese': '$2.85', 'Shrimp': '$3.55', 'Ham': '$2.85'}, {'Steak': '$3.60', 'Tuna Special': '$3.20', 'Turkey Club': '$3.20'})

We can access each value in our ChainMap by referencing its key name. For example, here’s a line of code that allows us to retrieve the price of the BLT sandwich:

print(menu["BLT"])

Our program returns: $3.05

In addition, it’s important to note that ChainMap updates when the dictionaries it contains are updated. So, if you change a value in the standard\_menu or secret\_menu dictionaries, the ChainMap object will also be updated. Here’s an example:

print(menu)

standard\_menu["BLT"] = "$3.10"

print(menu)

Our code returns:

ChainMap({'BLT': '$3.10', 'Roast Beef': '$3.55', 'Cheese': '$2.85', 'Shrimp': '$3.55', 'Ham': '$2.85'}, {'Steak': '$3.60', 'Tuna Special': '$3.20', 'Turkey Club': '$3.20'})

As you can see, the price of our BLT changed from $3.05 to $3.10 because we changed its price in our standard\_menu dictionary.

The ChainMap object also includes two functions that can be used to retrieve the keys or values from an object. We can illustrate this using the keys() and values() methods. These methods return the keys of our data (which we can use to reference a particular value) and the values they have been assigned:

print(list(menu.keys()))

print(list(menu.values()))

Our code returns the following:

['Steak', 'Tuna Special', 'Turkey Club', 'BLT', 'Roast Beef', 'Cheese', 'Prawn', 'Ham']

['$3.60', '$3.20', '$3.20', '$3.05', '$3.55', '$2.85', '$3.55', '$2.85']

Our code returned the keys and values of each item in our ChainMap object when we used the keys() and values() methods above.

In addition, you can add a new dictionary to a ChainMap object using the new\_child() method. Let’s say that our sandwich chef has been trying out new sandwiches on a test menu and wants to add two of them to our new menu. We could use the following code to achieve this goal:

test\_menu = {"Veggie Deluxe": "$3.00", "House Club Special": "$3.65"}

new\_menu = menu.new\_child(test\_menu)

print(new\_menu)

Our code returns an updated ChainMap with our new sandwiches at the start of the dictionary, as follows:

ChainMap({'Veggie Deluxe': '$3.00', 'House Club Special': '$3.65'}, {'BLT': '$3.05', 'Roast Beef': '$3.55', 'Cheese': '$2.85', 'Shrimp': '$3.55', 'Ham': '$2.85'}, {'Steak': '$3.60', 'Tuna Special': '$3.20', 'Turkey Club': '$3.20'})

# Python Datetime

Working with dates and times is one of the biggest challenges in programming. Between dealing with time zones, daylight saving time, and different written date formats, it can be tough to keep track of which days and times you’re referencing. Fortunately, the built-in Python **datetime** module can help you manage the complex nature of dates and times.

## Programming With Dates and Times

If you’ve ever worked on software that needed to keep track of times across several geographic areas, then you probably have a sense of why programming with time can be such a pain. The fundamental disconnect is that computer programs prefer events that are perfectly ordered and regular, but the way in which most humans use and refer to time is highly irregular.

One great example of this irregularity is **daylight saving time**. In the United States and Canada, clocks are set forward by one hour on the second Sunday in March and set back by one hour on the first Sunday in November. However, this has only been the case since 2007. Prior to 2007, clocks were set forward on the first Sunday in April and set back on the last Sunday in October.

Things get even more complicated when you consider **time zones**. Ideally, time zone boundaries would follow lines of longitude exactly. However, for historical and political reasons, time zone lines are rarely straight. Often, areas that are separated by large distances find themselves in the same time zone, and adjacent areas are in different time zones. There are some time zones out there with pretty funky shapes.

### How Computers Count Time

Nearly all computers count time from an instant called the **Unix epoch**.

**Unix time** (also known as **Epoch time**, **POSIX time**, **seconds since the Epoch**, or **UNIX Epoch time**) is a system for describing a point in time. It is the number of seconds that have elapsed since the *Unix epoch*, minus leap seconds; the Unix epoch is 00:00:00 [UTC](https://en.wikipedia.org/wiki/Coordinated_Universal_Time) on 1 January 1970 (an arbitrary date); leap seconds are ignored, with a leap second having the same Unix time as the second before it, and every day is treated as if it contains exactly 86400 seconds. Due to this treatment Unix time is not a true representation of UTC.

This occurred on January 1, 1970, at 00:00:00 UTC. UTC stands for **Coordinated Universal Time** and refers to the time at a longitude of 0°. UTC is often also called Greenwich Mean Time, or GMT. UTC is not adjusted for daylight saving time, so it consistently keeps twenty-four hours in every day.

By definition, Unix time elapses at the same rate as UTC, so a one-second step in UTC corresponds to a one-second step in Unix time. You can usually figure out the date and time in UTC of any given instant since January 1, 1970, by counting the number of seconds since the Unix epoch, with the exception of **leap seconds**. Leap seconds are occasionally added to UTC to account for the slowing of the Earth’s rotation but are not added to Unix time.

Nearly all programming languages, including Python, incorporate the concept of Unix time. Python’s standard library includes a module called time that can print the number of seconds since the Unix epoch:

>>> import time

>>> time.time()

1579718137.550164

In this example, you import the time module and execute time() to print the Unix time, or number of seconds (excluding leap seconds) since the epoch.

In addition to Unix time, computers need a way to convey time information to users. As you saw in the last example, Unix time is nearly impossible for a human to parse. Instead, Unix time is typically converted to UTC, which can then be converted into a local time using **time zone offsets**.

The **Internet Assigned Numbers Authority (IANA)** maintains a database of all of the values of time zone offsets. IANA also releases regular updates that include any changes in time zone offsets. This database is often included with your operating system, although certain applications may include an updated copy.

The database contains a copy of all the designated time zones and how many hours and minutes they’re offset from UTC. So, during the winter, when daylight saving time is not in effect, the US Eastern time zone has an offset of -05:00, or negative five hours from UTC. Other regions have different offsets, which may not be integer hours. The UTC offset for Nepal, for example, is +05:45, or positive five hours and forty-five minutes from UTC.

### How Standard Dates Can Be Reported

Unix time is how computers count time, but it would be incredibly inefficient for humans to determine the time by calculating the number of seconds from an arbitrary date. Instead, we work in terms of years, months, days, and so forth. But even with these conventions in place, another layer of complexity stems from the fact that different languages and cultures have different ways of writing the date.

For instance, in the United States, dates are usually written starting with the month, then the day, then the year. This means that January 31, 2020, is written as **01-31-2020**. This closely matches the long-form written version of the date.

However, most of Europe and many other areas write the date starting with the day, then the month, then the year. This means that January 31, 2020, is written as **31-01-2020**. These differences can cause all sorts of confusion when communicating across cultures.

To help avoid communication mistakes, the International Organization for Standardization (ISO) developed **ISO 8601**. This standard specifies that all dates should be written in order of most-to-least-significant data. This means the format is year, month, day, hour, minute, and second:

YYYY-MM-DD HH:MM:SS

In this example, YYYY represents a four-digit year, and MM and DD are the two-digit month and day, starting with a zero if necessary. After that, HH, MM, and SS represent the two-digit hours, minutes, and seconds, starting with a zero if necessary.

The advantage of this format is that the date can be represented with no ambiguity. Dates written as DD-MM-YYYY or MM-DD-YYYY can be misinterpreted if the day is a valid month number. You’ll see a little later on how you can use the ISO 8601 format with Python datetime.

### How Time Should Be Stored in Your Program

Most developers who have worked with time have heard the advice to convert local time to UTC and store that value for later reference. In many cases, especially when you’re storing dates from the past, this is enough information to do any necessary arithmetic.

However, a problem can happen if a user of your program inputs a future date in their local time. Time zone and daylight saving time rules change fairly frequently, as you saw earlier with the 2007 change in daylight saving time for the United States and Canada. If the time zone rules for your user’s location change before the future date that they inputted, then UTC won’t provide enough information to convert back to the correct local time.

In this case, you need to store the local time, including the time zone, that the user inputted as well as the version of the IANA time zone database that was in effect when the user saved the time. This way, you’ll always be able to convert the local time to UTC. However, this approach won’t always allow you to convert UTC to the correct local time.

## Using the Python datetime Module

As you can see, working with dates and times in programming can be complicated. Fortunately, you rarely need to implement complicated features from scratch these days since many open-source libraries are available to help out. This is definitely the case in Python, which includes three separate modules in the standard library to work with dates and times:

* **calendar** outputs calendars and provides functions using an idealized Gregorian calendar.
* **datetime** supplies classes for manipulating dates and times.
* **time** provides time-related functions where dates are not needed.

The main focus of datetime is to make it less complicated to access attributes of the object related to dates, times, and time zones. Since these objects are so useful, calendar also returns instances of classes from datetime.

time is less powerful and more complicated to use than datetime. Many functions in time return a special [**struct\_time**](https://docs.python.org/3/library/time.html#time.struct_time) instance. This object has a named tuple interface for accessing stored data, making it similar to an instance of datetime. However, it doesn’t support all of the features of datetime, especially the ability to perform arithmetic with time values.

datetime provides three classes that make up the high-level interface that most people will use:

1. **datetime.date** is an idealized date that assumes the Gregorian calendar extends infinitely into the future and past. This object stores the year, month, and day as attributes.
2. **datetime.time** is an idealized time that assumes there are 86,400 seconds per day with no leap seconds. This object stores the hour, minute, second, microsecond, and tzinfo (time zone information).
3. **datetime.datetime** is a combination of a date and a time. It has all the attributes of both classes.

### Creating Python datetime Instances

The three classes that represent dates and times in datetime have similar **initializers**. They can be instantiated by passing keyword arguments for each of the attributes, such as year, date, or hour. You can try the code below to get a sense of how each object is created:

>>> from datetime import date, time, datetime

>>> date(year=2020, month=1, day=31)

datetime.date(2020, 1, 31)

>>> time(hour=13, minute=14, second=31)

datetime.time(13, 14, 31)

>>> datetime(year=2020, month=1, day=31, hour=13, minute=14, second=31)

datetime.datetime(2020, 1, 31, 13, 14, 31)

In this code, you import the three main classes from datetime and **instantiate** each of them by passing arguments to the constructor. You can see that this code is somewhat verbose, and if you don’t have the information you need as integers, these techniques can’t be used to create datetime instances.

Fortunately, datetime provides several other convenient ways to create datetime instances. These methods don’t require you to use integers to specify each attribute, but instead allow you to use some other information:

1. **date.today()** creates a datetime.date instance with the current local date.
2. **datetime.now()** creates a datetime.datetime instance with the current local date and time.
3. **datetime.combine()** combines instances of datetime.date and datetime.time into a single datetime.datetime instance.

These three ways of creating datetime instances are helpful when you don’t know in advance what information you need to pass into the basic initializers. You can try out this code to see how the alternate initializers work:

>>> from datetime import date, time, datetime

>>> today = date.today()

>>> today

datetime.date(2020, 1, 24)

>>> now = datetime.now()

>>> now

datetime.datetime(2020, 1, 24, 14, 4, 57, 10015)

>>> current\_time = time(now.hour, now.minute, now.second)

>>> datetime.combine(today, current\_time)

datetime.datetime(2020, 1, 24, 14, 4, 57)

In this code, you use date.today(), datetime.now(), and datetime.combine() to create instances of date, datetime, and time objects. Each instance is stored in a different variable:

1. **today** is a date instance that has only the year, month, and day.
2. **now** is a datetime instance that has the year, month, day, hour, minute, second, and microseconds.
3. **current\_time** is a time instance that has the hour, minute, and second set to the same values as now.

On the last line, you combine the date information in today with the time information in current\_time to produce a new datetime instance.

**Warning:** datetime also provides datetime.utcnow(), which returns an instance of datetime at the current UTC. However, the Python documentation recommends against using this method because it doesn’t include any time zone information in the resulting instance.

Using datetime.utcnow() may produce some surprising results when doing arithmetic or comparisons between datetime instances.

### Using Strings to Create Python datetime Instances

Another way to create date instances is to use .fromisoformat(). To use this method, you provide a string with the date in the ISO 8601 format that you learned about earlier. For instance, you might provide a string with the year, month, and date specified:

2020-01-31

This string represents the date January 31, 2020, according to the ISO 8601 format. You can create a date instance with the following example:

>>> from datetime import date

>>> date.fromisoformat("2020-01-31")

datetime.date(2020, 1, 31)

In this code, you use date.fromisoformat() to create a date instance for January 31, 2020. This method is very useful because it’s based on the ISO 8601 standard. But what if you have a string that represents a date and time but isn’t in the ISO 8601 format?

Fortunately, Python datetime provides a method called .strptime() to handle this situation. This method uses a special **mini-language** to tell Python which parts of the string are associated with the datetime attributes.

To construct a datetime from a string using .strptime(), you have to tell Python what each of the parts of the string represents using formatting codes from the mini-language. You can try this example to see how .strptime() works:

1 >>> date\_string = "01-31-2020 14:45:37"

2 >>> format\_string = "%m-%d-%Y %H:%M:%S"

On **line 1**, you create date\_string, which represents the date and time January 31, 2020, at 2:45:37 PM. On **line 2**, you create format\_string, which uses the mini-language to specify how the parts of date\_string will be turned into datetime attributes.

In format\_string, you include several formatting codes and all of the dashes (-), colons (:), and spaces exactly as they appear in date\_string. To process the date and time in date\_string, you include the following formatting codes:

| **Component** | **Code** | **Value** |
| --- | --- | --- |
| Year (as four-digit integer ) | %Y | 2020 |
| Month (as zero-padded decimal) | %m | 01 |
| Date (as zero-padded decimal) | %d | 31 |
| Hour (as zero-padded decimal with 24-hour clock) | %H | 14 |
| Minute (as zero-padded decimal) | %M | 45 |
| Second (as zero-padded decimal) | %S | 37 |

Now that date\_string and format\_string are defined, you can use them to create a datetime instance. Here’s an example of how .strptime() works:

3 >>> from datetime import datetime

4 >>> datetime.strptime(date\_string, format\_string)

5 datetime.datetime(2020, 1, 31, 14, 45, 37)

In this code, you import datetime on **line 3** and use datetime.strptime() with date\_string and format\_string on **line 4**. Finally, **line 5** shows the values of the attributes in the datetime instance created by .strptime(). You can see that they match the values shown in the table above.

**Note:** There are more advanced ways to create datetime instances, but they involve using third-party libraries that must be installed. One particularly neat library is called dateparser, which allows you to provide natural language string inputs. The input is even supported in a number of languages:

1 >>> import dateparser

2 >>> dateparser.parse("yesterday")

3 datetime.datetime(2020, 3, 13, 14, 39, 1, 350918)

4 >>> dateparser.parse("morgen")

5 datetime.datetime(2020, 3, 15, 14, 39, 7, 314754)

In this code, you use dateparser to create two datetime instances by passing two different string representations of time. On **line 1**, you import dateparser. Then, on **line 2**, you use .parse() with the argument "yesterday" to create a datetime instance twenty-four hours in the past. At the time of writing, this was March 13, 2020, at 2:39 PM.

On **line 3**, you use .parse() with the argument "morgen". *Morgen* is the German word for tomorrow, so dateparser creates a datetime instance twenty-four hours in the future. At the time of writing, this was March 15 at 2:39 PM.

## Working With Time Zones

As you saw earlier, storing the time zone in which a date occurs is an important aspect of ensuring your code is correct. Python datetime provides tzinfo, which is an abstract base class that allows datetime.datetime and datetime.time to include time zone information, including an idea of daylight saving time.

However, datetime does not provide a direct way to interact with the IANA time zone database. The Python datetime.tzinfo documentation recommends using a third-party package called dateutil. You can install dateutil with pip:

$ python -m pip install python-dateutil

Note that the name of the package that you install from PyPI, python-dateutil, is different from the name that you use to import the package, which is just dateutil.

### Using dateutil to Add Time Zones to Python datetime

One reason that dateutil is so useful is that it includes an interface to the IANA time zone database. This takes the hassle out of assigning time zones to your datetime instances. Try out this example to see how to set a datetime instance to have your local time zone:

>>> from dateutil import tz

>>> from datetime import datetime

>>> now = datetime.now(tz=tz.tzlocal())

>>> now

datetime.datetime(2020, 1, 26, 0, 55, 3, 372824, tzinfo=tzlocal())

>>> now.tzname()

'Eastern Standard Time'

In this example, you import tz from dateutil and datetime from datetime. You then create a datetime instance set to the current time using .now().

You also pass the tz keyword to .now() and set tz equal to tz.tzlocal(). In dateutil, tz.tzlocal() returns a concrete instance of datetime.tzinfo. This means that it can represent all the necessary time zone offset and daylight saving time information that datetime needs.

You also print the name of the time zone using .tzname(), which prints 'Eastern Standard Time'. This is the output for Windows, but on macOS or Linux, your output might read 'EST' if you’re in the US Eastern time zone during the winter.

You can also create time zones that are not the same as the time zone reported by your computer. To do this, you’ll use tz.gettz() and pass the official IANA name for the time zone you’re interested in. Here’s an example of how to use tz.gettz():

>>> from dateutil import tz

>>> from datetime import datetime

>>> London\_tz = tz.gettz("Europe/London")

>>> now = datetime.now(tz=London\_tz)

>>> now

datetime.datetime(2020, 1, 26, 6, 14, 53, 513460, tzinfo=tzfile('GB-Eire'))

>>> now.tzname()

'GMT'

In this example, you use tz.gettz() to retrieve the time zone information for London, United Kingdom and store it in London\_tz. You then retrieve the current time, setting the time zone to London\_tz.

On Windows, this gives the tzinfo attribute the value tzfile('GB-Eire'). On macOS or Linux, the tzinfo attribute will look something like tzfile('/usr/share/zoneinfo/Europe/London), but it might be slightly different depending on where dateutil pulls the time zone data from.

You also use tzname() to print the name of the time zone, which is now 'GMT', meaning Greenwich Mean Time. This output is the same on Windows, macOS, and Linux.

In an earlier section, you learned that you shouldn’t use .utcnow() to create a datetime instance at the current UTC. Now you know how to use dateutil.tz to supply a time zone to the datetime instance. Here’s an example modified from the recommendation in the Python documentation:

>>> from dateutil import tz

>>> from datetime import datetime

>>> datetime.now(tz=tz.UTC)

datetime.datetime(2020, 3, 14, 19, 1, 20, 228415, tzinfo=tzutc())

In this code, you use tz.UTC to set the time zone of datetime.now() to the UTC time zone. This method is recommended over using utcnow() because utcnow() returns a naive datetime instance, whereas the method demonstrated here returns an aware datetime instance.

### Comparing Naive and Aware Python datetime Instances

Python datetime instances support two types of operation, naive and aware. The basic difference between them is that naive instances don’t contain time zone information, whereas aware instances do.

## Doing Arithmetic With Python datetime

Python datetime instances support several types of arithmetic. As you saw earlier, this relies on using timedelta instances to represent time intervals. timedelta is very useful because it’s built into the Python standard library. Here’s an example of how to work with timedelta:

>>> from datetime import datetime, timedelta

>>> now = datetime.now()

>>> now

datetime.datetime(2020, 1, 26, 9, 37, 46, 380905)

>>> tomorrow = timedelta(days=+1)

>>> now + tomorrow

datetime.datetime(2020, 1, 27, 9, 37, 46, 380905)

In this code, you create now, which stores the current time, and tomorrow, which is a timedelta of +1 days. Next, you add now and tomorrow to produce a datetime instance one day in the future. Note that working with naive datetime instances, as you are here, means that the day attribute of the datetime increments by one and does not account for any repeated or skipped time intervals.

timedelta instances also support negative values as the input to the arguments:

yesterday = timedelta(days=-1)

>>> now + yesterday

datetime.datetime(2020, 1, 25, 9, 37, 46, 380905)

In this example, you provide -1 as the input to timedelta, so when you add now and yesterday, the result is a decrease by one in the days attribute.

timedelta instances support addition and subtraction as well as positive and negative integers for all arguments. You can even provide a mix of positive and negative arguments. For instance, you might want to add three days and subtract four hours:

>>> delta = timedelta(days=+3, hours=-4)

>>> now + delta

datetime.datetime(2020, 1, 29, 5, 37, 46, 380905)

In this example, you add three days and subtract four hours, so the new datetime is at January 29 at 5:37 AM. timedelta is very useful in this way, but it’s somewhat limited because it cannot add or subtract intervals larger than a day, such as a month or a year. Fortunately, dateutil provides a more powerful replacement called [**relativedelta**](https://dateutil.readthedocs.io/en/stable/relativedelta.html).

The basic syntax of relativedelta is very similar to timedelta. You can provide keyword arguments that produce changes of any number of years, months, days, hours, seconds, or microseconds. You can reproduce the first timedelta example with this code:

>>> from dateutil.relativedelta import relativedelta

>>> tomorrow = relativedelta(days=+1)

>>> now + tomorrow

datetime.datetime(2020, 1, 27, 9, 37, 46, 380905)

In this example, you use relativedelta instead of timedelta to find the datetime corresponding to tomorrow. Now you can try adding five years, one month, and three days to now while subtracting four hours and thirty minutes:

>>> delta = relativedelta(years=+5, months=+1, days=+3, hours=-4, minutes=-30)

>>> now + delta

datetime.datetime(2025, 3, 1, 5, 7, 46, 380905)

Notice in this example that the date ends up as March 1, 2025. This is because adding three days to now would be January 29, and adding one month to that would be February 29, which only exists in a leap year. Since 2025 is not a leap year, the date rolls over to the next month.

You can also use relativedelta to calculate the difference between two datetime instances. Earlier, you used the subtraction operator to find the difference between two Python datetime instances, PYCON\_DATE and now. With relativedelta, instead of using the subtraction operator, you need to pass the two datetime instances as arguments:

>>> now

datetime.datetime(2020, 1, 26, 9, 37, 46, 380905)

>>> tomorrow = datetime(2020, 1, 27, 9, 37, 46, 380905)

>>> relativedelta(now, tomorrow)

relativedelta(days=-1)

In this example, you create a new datetime instance for tomorrow by incrementing the days field by one. Then, you use relativedelta and pass now and tomorrow as the two arguments. dateutil then takes the difference between these two datetime instances and returns the result as a relativedelta instance. In this case, the difference is -1 days, since now happens before tomorrow.

dateutil.relativedelta objects have countless other uses. You can use them to find complex calendar information, such as the next year in which October the 13th falls on a Friday or what the date will be on the last Friday of the current month. You can even use them to replace attributes of a datetime instance and create, for example, a datetime one week in the future at 10:00 AM.

## Alternatives to Python datetime and dateutil

Python datetime and dateutil are a powerful combination of libraries when you’re working with dates and times. dateutil is even recommended in the Python documentation. However, there are many other libraries that you can use to work with dates and times in Python. Some of these rely on datetime and dateutil, while others are completely independent replacements:

* [**pytz**](https://pypi.org/project/pytz/) provides time zone information similar to dateutil. It uses a somewhat different interface than the standard datetime.tzinfo, so be aware of the [potential problems](https://blog.ganssle.io/articles/2018/03/pytz-fastest-footgun.html) if you decide to use it.
* [**Arrow**](https://arrow.readthedocs.io/en/latest/) provides a drop-in replacement for datetime. It’s inspired by moment.js, so if you’re coming from web development, then this might be a more familiar interface.
* [**Pendulum**](https://pendulum.eustace.io/) provides another drop-in replacement for datetime. It includes a time zone interface and an improved timedelta implementation.
* [**Maya**](https://github.com/timofurrer/maya) provides a similar interface as datetime. It relies on Pendulum for parts of the parsing library.
* [**dateparser**](https://dateparser.readthedocs.io/en/latest/) provides an interface to generate datetime instances from human-readable text. It’s flexible and supports many languages.

In addition, if you work heavily with [NumPy](https://realpython.com/tutorials/numpy/), [Pandas](https://realpython.com/courses/introduction-pandas-and-vincent/), or other [data science](https://realpython.com/tutorials/data-science/) packages, then there are a few options that might be useful to you:

* [**NumPy**](https://numpy.org/doc/1.18/reference/arrays.datetime.html) provides a similar API to the built-in Python datetime library, but the NumPy version can be used in arrays.
* [**Pandas**](https://pandas.pydata.org/pandas-docs/stable/user_guide/timeseries.html) provides support for time-series data in [DataFrames](https://realpython.com/courses/pandas-dataframes-101/), usually sequential values of time-based events, by using the NumPy datetime module.
* [**cftime**](https://unidata.github.io/cftime/api.html) provides support for calendars other than the [proleptic Gregorian calendar](https://en.wikipedia.org/wiki/Proleptic_Gregorian_calendar) as well as other time units conforming to the Climate and Forecasting (CF) conventions. It’s used by the [xarray](http://xarray.pydata.org/en/stable/time-series.html) package to provide time-series support.

# Python RegEx

A RegEx, or Regular Expression, is a sequence of characters that forms a search pattern.

RegEx can be used to check if a string contains the specified search pattern.

## RegEx Module

Python has a built-in package called re, which can be used to work with Regular Expressions.

Import the re module:

import re

## RegEx in Python

When you have imported the re module, you can start using regular expressions:

**Example**

Search the string to see if it starts with "The" and ends with "Spain":

import re  
  
txt = "The rain in Spain"

x = re.search("^The.\*Spain$", txt)

## RegEx Functions

The re module offers a set of functions that allows us to search a string for a match:

|  |  |
| --- | --- |
| **Function** | **Description** |
| findall | Returns a list containing all matches |
| search | Returns a Match object if there is a match anywhere in the string |
| split | Returns a list where the string has been split at each match |
| sub | Replaces one or many matches with a string |

## Metacharacters

Metacharacters are characters with a special meaning:

|  |  |  |
| --- | --- | --- |
| **Character** | **Description** | **Example** |
| [] | A set of characters | "[a-m]" |
| \ | Signals a special sequence (can also be used to escape special characters) | "\d" |
| . | Any character (except newline character) | "he..o" |
| ^ | Starts with | "^hello" |
| $ | Ends with | "world$" |
| \* | Zero or more occurrences | "aix\*" |
| + | One or more occurrences | "aix+" |
| {} | Exactly the specified number of occurrences | "al{2}" |
| | | Either or | "falls|stays" |
| () | Capture and group |  |

## Special Sequences

A special sequence is a \ followed by one of the characters in the list below, and has a special meaning:

|  |  |  |
| --- | --- | --- |
| **Character** | **Description** | **Example** |
| \A | Returns a match if the specified characters are at the beginning of the string | "\AThe" |
| \b | Returns a match where the specified characters are at the beginning or at the end of a word (the "r" in the beginning is making sure that the string is being treated as a "raw string") | r"\bain" r"ain\b" |
| \B | Returns a match where the specified characters are present, but NOT at the beginning (or at the end) of a word (the "r" in the beginning is making sure that the string is being treated as a "raw string") | r"\Bain" r"ain\B" |
| \d | Returns a match where the string contains digits (numbers from 0-9) | "\d" |
| \D | Returns a match where the string DOES NOT contain digits | "\D" |
| \s | Returns a match where the string contains a white space character | "\s" |
| \S | Returns a match where the string DOES NOT contain a white space character | "\S" |
| \w | Returns a match where the string contains any word characters (characters from a to Z, digits from 0-9, and the underscore \_ character) | "\w" |
| \W | Returns a match where the string DOES NOT contain any word characters | "\W" |
| \Z | Returns a match if the specified characters are at the end of the string | "Spain\Z" |

## Sets

A set is a set of characters inside a pair of square brackets [] with a special meaning:

|  |  |
| --- | --- |
| **Set** | **Description** |
| [arn] | Returns a match where one of the specified characters (a, r, or n) are present |
| [a-n] | Returns a match for any lower case character, alphabetically between a and n |
| [^arn] | Returns a match for any character EXCEPT a, r, and n |
| [0123] | Returns a match where any of the specified digits (0, 1, 2, or 3) are present |
| [0-9] | Returns a match for any digit between 0 and 9 |
| [0-5][0-9] | Returns a match for any two-digit numbers from 00 and 59 |
| [a-zA-Z] | Returns a match for any character alphabetically between a and z, lower case OR upper case |
| [+] | In sets, +, \*, ., |, (), $,{} has no special meaning, so [+] means: return a match for any + character in the string |

## The findall() Function

The findall() function returns a list containing all matches.

**Example**

Print a list of all matches:

import re  
  
txt = "The rain in Spain"

x = re.findall("ai", txt)

print(x)

The list contains the matches in the order they are found. If no matches are found, an empty list is returned.

>>> s = 'foo123bar'

>>> '123' in s

True

>>> s = 'foo123bar'

>>> s.find('123')

3

>>> s.index('123')

3

## The search() Function

The search() function searches the string for a match, and returns a Match object if there is a match.

If there is more than one match, only the first occurrence of the match will be returned:

**Example**

Search for the first white-space character in the string:

import re  
  
txt = "The rain in Spain"

x = re.search("\s", txt)\

print("The first white-space character is located in position:",

x.start())

If no matches are found, the value None is returned.

## The split() Function

The split() function returns a list where the string has been split at each match:

**Example**

Split at each white-space character:

import re  
  
txt = "The rain in Spain"

x = re.split("\s", txt)

print(x)

# split with comma as the delimter.

txt2 = "The,rain,in,Spain"

x2 = re.split("\,", txt2)

print(x2)

You can control the number of occurrences by specifying the maxsplit parameter:

Example

Split the string only at the first occurrence:

import re  
  
txt = "The rain in Spain"

x = re.split("\s", txt, 1)

print(x)

## The sub() Function

The sub() function replaces the matches with the text of your choice:

**Example**

Replace every white-space character with the number 9:

import re  
  
txt = "The rain in Spain"

x = re.sub("\s", "9", txt)

print(x)

You can control the number of replacements by specifying the count parameter:

Example

Replace the first 2 occurrences:

import re  
  
txt = "The rain in Spain"

x = re.sub("\s", "9", txt, 2)

print(x)

## Match Object

A Match Object is an object containing information about the search and the result.

**Note:** If there is no match, the value None will be returned, instead of the Match Object.

**Example**

Do a search that will return a Match Object:

import re  
  
txt = "The rain in Spain"

x = re.search("ai", txt)

print(x) #this will print an object

The Match object has properties and methods used to retrieve information about the search, and the result:

* .span() returns a tuple containing the start-, and end positions of the match.
* .string returns the string passed into the function
* .group() returns the part of the string where there was a match

**Example**

Print the position (start- and end-position) of the first match occurrence.

The regular expression looks for any words that starts with an upper case "S":

import re  
  
txt = "The rain in Spain"

x = re.search(r"\bS\w+", txt)

print(**x.span()**)

**Example**

Print the string passed into the function:

import re  
  
txt = "The rain in Spain"

x = re.search(r"\bS\w+", txt)

print(**x.string**)

**Example**

Print the part of the string where there was a match.

The regular expression looks for any words that starts with an upper case "S":

import re  
  
txt = "The rain in Spain"

x = re.search(r"\bS\w+", txt)

print(**x.group()**)

**Note:** If there is no match, the value None will be returned, instead of the Match Object.

**More examples:**

>>> s = 'foo123bar'

>>> re.search('[0-9][0-9][0-9]', s)

<\_sre.SRE\_Match object; span=(3, 6), match='123'>

>>> re.search('[0-9][0-9][0-9]', 'foo456bar')

<\_sre.SRE\_Match object; span=(3, 6), match='456'>

>>> re.search('[0-9][0-9][0-9]', '234baz')

<\_sre.SRE\_Match object; span=(0, 3), match='234'>

>>> re.search('[0-9][0-9][0-9]', 'qux678')

<\_sre.SRE\_Match object; span=(3, 6), match='678'>

>>> print(re.search('[0-9][0-9][0-9]', '12foo34'))

None

Take a look at another regex metacharacter. The dot (.) metacharacter matches any character except a newline, so it functions like a wildcard:

>>> s = 'foo123bar'

>>> re.search('1.3', s)

<\_sre.SRE\_Match object; span=(3, 6), match='123'>

>>> s = 'foo13bar'

>>> print(re.search('1.3', s))

None

[] Specifies a specific set of characters to match.

>>> re.search('ba[artz]', 'foobarqux')

<\_sre.SRE\_Match object; span=(3, 6), match='bar'>

>>> re.search('ba[artz]', 'foobazqux')

<\_sre.SRE\_Match object; span=(3, 6), match='baz'>

>>> re.search('[a-z]', 'FOObar')

<\_sre.SRE\_Match object; span=(3, 4), match='b'>

[0-9] matches any digit character:

>>> re.search('[0-9][0-9]', 'foo123bar')

<\_sre.SRE\_Match object; span=(3, 5), match='12'>

[0-9a-fA-F] matches any hexadecimal digit character:

>>> re.search('[0-9a-fA-f]', '--- a0 ---')

<\_sre.SRE\_Match object; span=(4, 5), match='a'>

[^0-9] matches any character that isn’t a digit:

>>> re.search('[^0-9]', '12345foo')

<\_sre.SRE\_Match object; span=(5, 6), match='f'>

What if you want the character class to include a literal hyphen character? You can place it as the first or last character or escape it with a backslash (\):

>>> re.search('[-abc]', '123-456')

<\_sre.SRE\_Match object; span=(3, 4), match='-'>

>>> re.search('[abc-]', '123-456')

<\_sre.SRE\_Match object; span=(3, 4), match='-'>

>>> re.search('[ab\-c]', '123-456')

<\_sre.SRE\_Match object; span=(3, 4), match='-'>

If you want to include a literal ']' in a character class, then you can place it as the first character or escape it with backslash:

>>> re.search('[]]', 'foo[1]')

<\_sre.SRE\_Match object; span=(5, 6), match=']'>

>>> re.search('[ab\]cd]', 'foo[1]')

<\_sre.SRE\_Match object; span=(5, 6), match=']'>

dot (.): Specifies a wildcard.

The . metacharacter matches any single character except a newline:

>>> re.search('foo.bar', 'fooxbar')

<\_sre.SRE\_Match object; span=(0, 7), match='fooxbar'>

>>> print(re.search('foo.bar', 'foobar'))

None

>>> print(re.search('foo.bar', 'foo\nbar'))

None

\w or \W: Match based on whether a character is a word character.

\w matches any alphanumeric word character. Word characters are uppercase and lowercase letters, digits, and the underscore (\_) character, so \w is essentially shorthand for [a-zA-Z0-9\_]:

>>> re.search('\w', '#(.a$@&')

<\_sre.SRE\_Match object; span=(3, 4), match='a'>

>>> re.search('[a-zA-Z0-9\_]', '#(.a$@&')

<\_sre.SRE\_Match object; span=(3, 4), match='a'>

\W is the opposite. It matches any non-word character and is equivalent to [^a-zA-Z0-9\_]:

>>> re.search('\W', 'a\_1\*3Qb')

<\_sre.SRE\_Match object; span=(3, 4), match='\*'>

>>> re.search('[^a-zA-Z0-9\_]', 'a\_1\*3Qb')

<\_sre.SRE\_Match object; span=(3, 4), match='\*'>

\d or \D: Match based on whether a character is a decimal digit.

\d matches any decimal digit character. \D is the opposite. It matches any character that *isn’t* a decimal digit:

>>> re.search('\d', 'abc4def')

<\_sre.SRE\_Match object; span=(3, 4), match='4'>

>>> re.search('\D', '234Q678')

<\_sre.SRE\_Match object; span=(3, 4), match='Q'>

\s or \S: Match based on whether a character represents whitespace.

\s matches any whitespace character:

>>> re.search('\s', 'foo\nbar baz')

<\_sre.SRE\_Match object; span=(3, 4), match='\n'>

\S is the opposite of \s. It matches any character that *isn’t* whitespace:

>>> re.search('\S', ' \n foo \n ')

<\_sre.SRE\_Match object; span=(4, 5), match='f'>

Again, \s and \S consider a newline to be whitespace. In the example above, the first non-whitespace character is 'f'.

The character class sequences \w, \W, \d, \D, \s, and \S can appear inside a square bracket character class as well:

>>> re.search('[\d\w\s]', '---3---')

<\_sre.SRE\_Match object; span=(3, 4), match='3'>

>>> re.search('[\d\w\s]', '---a---')

<\_sre.SRE\_Match object; span=(3, 4), match='a'>

>>> re.search('[\d\w\s]', '--- ---')

<\_sre.SRE\_Match object; span=(3, 4), match=' '>

In this case, [\d\w\s] matches any digit, word, or whitespace character. And since \w includes \d, the same character class could also be expressed slightly shorter as [\w\s].

backslash (\): Removes the special meaning of a metacharacter.

>>> re.search('.', 'foo.bar')

2<\_sre.SRE\_Match object; span=(0, 1), match='f'>

3

4>>> re.search('\.', 'foo.bar')

5<\_sre.SRE\_Match object; span=(3, 4), match='.'>

Using backslashes for escaping can get messy. Suppose you have a string that contains a single backslash:

>>> s = r'foo\bar'

>>> print(s)

foo\bar

>>> re.search('\\', s)

Traceback (most recent call last):

File "<pyshell#3>", line 1, in <module>

re.search('\\', s)

File "C:\Python36\lib\re.py", line 182, in search

return \_compile(pattern, flags).search(string)

File "C:\Python36\lib\re.py", line 301, in \_compile

p = sre\_compile.compile(pattern, flags)

File "C:\Python36\lib\sre\_compile.py", line 562, in compile

p = sre\_parse.parse(p, flags)

File "C:\Python36\lib\sre\_parse.py", line 848, in parse

source = Tokenizer(str)

File "C:\Python36\lib\sre\_parse.py", line 231, in \_\_init\_\_

self.\_\_next()

File "C:\Python36\lib\sre\_parse.py", line 245, in \_\_next

self.string, len(self.string) - 1) from None

sre\_constants.error: bad escape (end of pattern) at position 0

1. The Python interpreter is the first to process the string literal '\\'. It interprets that as an escaped backslash and passes only a single backslash to re.search().
2. The regex parser receives just a single backslash, which isn’t a meaningful regex, so the messy error ensues.

There are two ways around this. First, you can escape *both* backslashes in the original string literal:

>>> re.search('\\\\', s)

<\_sre.SRE\_Match object; span=(3, 4), match='\\'>

>>> re.search(r'\\', s)

<\_sre.SRE\_Match object; span=(3, 4), match='\\'>

^ or \A: Anchor a match to the start of <string>.

>>> re.search('^foo', 'foobar')

<\_sre.SRE\_Match object; span=(0, 3), match='foo'>

>>> print(re.search('^foo', 'barfoo'))

None

>>> re.search('\Afoo', 'foobar')

<\_sre.SRE\_Match object; span=(0, 3), match='foo'>

>>> print(re.search('\Afoo', 'barfoo'))

None

$ or \Z: Anchor a match to the end of <string>.

>>> re.search('bar$', 'foobar')

<\_sre.SRE\_Match object; span=(3, 6), match='bar'>

>>> print(re.search('bar$', 'barfoo'))

None

>>> re.search('bar\Z', 'foobar')

<\_sre.SRE\_Match object; span=(3, 6), match='bar'>

>>> print(re.search('bar\Z', 'barfoo'))

None

As a special case, $ (but not \Z) also matches just before a single newline at the end of the search string:

>>> re.search('bar$', 'foobar\n')

<\_sre.SRE\_Match object; span=(3, 6), match='bar'>

\b: Anchors a match to a word boundary.

>>> re.search(r'\bbar', 'foo bar')

2<\_sre.SRE\_Match object; span=(4, 7), match='bar'>

3>>> re.search(r'\bbar', 'foo.bar')

4<\_sre.SRE\_Match object; span=(4, 7), match='bar'>

5

6>>> print(re.search(r'\bbar', 'foobar'))

7None

8

9>>> re.search(r'foo\b', 'foo bar')

10<\_sre.SRE\_Match object; span=(0, 3), match='foo'>

11>>> re.search(r'foo\b', 'foo.bar')

12<\_sre.SRE\_Match object; span=(0, 3), match='foo'>

13

14>>> print(re.search(r'foo\b', 'foobar'))

15None

Using the \b anchor on both ends of the <regex> will cause it to match when it’s present in the search string as a whole word:

>>> re.search(r'\bbar\b', 'foo bar baz')

<\_sre.SRE\_Match object; span=(4, 7), match='bar'>

>>> re.search(r'\bbar\b', 'foo(bar)baz')

<\_sre.SRE\_Match object; span=(4, 7), match='bar'>

>>> print(re.search(r'\bbar\b', 'foobarbaz'))

None

\B: Anchors a match to a location that isn’t a word boundary.

\B does the opposite of \b. It asserts that the regex parser’s current position must *not* be at the start or end of a word:

1>>> print(re.search(r'\Bfoo\B', 'foo'))

2None

3>>> print(re.search(r'\Bfoo\B', '.foo.'))

4None

5

6>>> re.search(r'\Bfoo\B', 'barfoobaz')

7<\_sre.SRE\_Match object; span=(3, 6), match='foo'>

In this case, a match happens on **line 7** because no word boundary exists at the start or end of 'foo' in the search string 'barfoobaz'.

\*: Matches zero or more repetitions of the preceding regex.

For example, a\* matches zero or more 'a' characters. That means it would match an empty string, 'a', 'aa', 'aaa', and so on.

Consider these examples:

1>>> re.search('foo-\*bar', 'foobar') # Zero dashes

2<\_sre.SRE\_Match object; span=(0, 6), match='foobar'>

3>>> re.search('foo-\*bar', 'foo-bar') # One dash

4<\_sre.SRE\_Match object; span=(0, 7), match='foo-bar'>

5>>> re.search('foo-\*bar', 'foo--bar') # Two dashes

6<\_sre.SRE\_Match object; span=(0, 8), match='foo--bar'>

+: Matches one or more repetitions of the preceding regex.

This is similar to \*, but the quantified regex must occur at least once:

1>>> print(re.search('foo-+bar', 'foobar')) # Zero dashes

2None

3>>> re.search('foo-+bar', 'foo-bar') # One dash

4<\_sre.SRE\_Match object; span=(0, 7), match='foo-bar'>

5>>> re.search('foo-+bar', 'foo--bar') # Two dashes

6<\_sre.SRE\_Match object; span=(0, 8), match='foo--bar'>

?: Matches zero or one repetitions of the preceding regex.

Again, this is similar to \* and +, but in this case there’s only a match if the preceding regex occurs once or not at all:

1>>> re.search('foo-?bar', 'foobar') # Zero dashes

2<\_sre.SRE\_Match object; span=(0, 6), match='foobar'>

3>>> re.search('foo-?bar', 'foo-bar') # One dash

4<\_sre.SRE\_Match object; span=(0, 7), match='foo-bar'>

5>>> print(re.search('foo-?bar', 'foo--bar')) # Two dashes

6None

>>> re.match('foo[1-9]\*bar', 'foobar')

<\_sre.SRE\_Match object; span=(0, 6), match='foobar'>

>>> re.match('foo[1-9]\*bar', 'foo42bar')

<\_sre.SRE\_Match object; span=(0, 8), match='foo42bar'>

>>> print(re.match('foo[1-9]+bar', 'foobar'))

None

>>> re.match('foo[1-9]+bar', 'foo42bar')

<\_sre.SRE\_Match object; span=(0, 8), match='foo42bar'>

>>> re.match('foo[1-9]?bar', 'foobar')

<\_sre.SRE\_Match object; span=(0, 6), match='foobar'>

>>> print(re.match('foo[1-9]?bar', 'foo42bar'))

None

{m}: Matches exactly m repetitions of the preceding regex.

This is similar to \* or +, but it specifies exactly how many times the preceding regex must occur for a match to succeed:

>>> print(re.search('x-{3}x', 'x--x')) # Two dashes

None

>>> re.search('x-{3}x', 'x---x') # Three dashes

<\_sre.SRE\_Match object; span=(0, 5), match='x---x'>

>>> print(re.search('x-{3}x', 'x----x')) # Four dashes

None

{m,n}: Matches any number of repetitions of the preceding regex from m to n, inclusive.

In the following example, the quantified <regex> is -{2,4}. The match succeeds when there are two, three, or four dashes between the 'x' characters but fails otherwise:

>>> for i in range(1, 6):

... s = f"x{'-' \* i}x"

... print(f'{i} {s:10}', re.search('x-{2,4}x', s))

...

1 x-x None

2 x--x <\_sre.SRE\_Match object; span=(0, 4), match='x--x'>

3 x---x <\_sre.SRE\_Match object; span=(0, 5), match='x---x'>

4 x----x <\_sre.SRE\_Match object; span=(0, 6), match='x----x'>

5 x-----x None

Omitting m implies a lower bound of 0, and omitting n implies an unlimited upper bound.

m.groups(): Returns a tuple containing all the captured groups from a regex match.

Consider this example:

>>> m = re.search('(\w+),(\w+),(\w+)', 'foo,quux,baz')

>>> m

<\_sre.SRE\_Match object; span=(0, 12), match='foo:quux:baz'>

Each of the three (\w+) expressions matches a sequence of word characters. The full regex (\w+),(\w+),(\w+) breaks the search string into three comma-separated tokens.

Because the (\w+) expressions use grouping parentheses, the corresponding matching tokens are **captured**. To access the captured matches, you can use .groups(), which returns a tuple containing all the captured matches in order:

>>> m.groups()

('foo', 'quux', 'baz')

m.group(<n>): Returns a string containing the <n>th captured match.

>>> m = re.search('(\w+),(\w+),(\w+)', 'foo,quux,baz')

>>> m.groups()

('foo', 'quux', 'baz')

>>> m.group(1)

'foo'

>>> m.group(2)

'quux'

>>> m.group(3)

'baz'

>>> m.group(0)

'foo,quux,baz'

>>> m.group()

'foo,quux,baz'

m.group(<n1>, <n2>, ...): Returns a tuple containing the specified captured matches.

>>> m.groups()

('foo', 'quux', 'baz')

>>> m.group(2, 3)

('quux', 'baz')

>>> m.group(3, 2, 1)

('baz', 'quux', 'foo')

>>> (m.group(3), m.group(2), m.group(1))

('baz', 'quux', 'foo')

# Python Modules and Packages

Python **modules** and Python **packages**, two mechanisms that facilitate **modular programming**.

**Modular programming** refers to the process of breaking a large, unwieldy programming task into separate, smaller, more manageable subtasks or **modules**. Individual modules can then be cobbled together like building blocks to create a larger application.

There are several advantages to **modularizing** code in a large application:

* **Simplicity:** Rather than focusing on the entire problem at hand, a module typically focuses on one relatively small portion of the problem. If you’re working on a single module, you’ll have a smaller problem domain to wrap your head around. This makes development easier and less error-prone.
* **Maintainability:** Modules are typically designed so that they enforce logical boundaries between different problem domains. If modules are written in a way that minimizes interdependency, there is decreased likelihood that modifications to a single module will have an impact on other parts of the program. (You may even be able to make changes to a module without having any knowledge of the application outside that module.) This makes it more viable for a team of many programmers to work collaboratively on a large application.
* **Reusability:** Functionality defined in a single module can be easily reused (through an appropriately defined interface) by other parts of the application. This eliminates the need to duplicate code.
* **Scoping:** Modules typically define a separate **namespace**, which helps avoid collisions between identifiers in different areas of a program. (One of the tenets in the Zen of Python is Namespaces are one honking great idea—let’s do more of those!)

**Functions**, **modules** and **packages** are all constructs in Python that promote code modularization.

## Python Modules: Overview

There are actually three different ways to define a **module** in Python:

1. A module can be written in Python itself.
2. A module can be written in **C** and loaded dynamically at run-time, like the re (**regular expression**) module.
3. A **built-in** module is intrinsically contained in the interpreter, like the itertools module.

A module’s contents are accessed the same way in all three cases: with the import statement.

Here, the focus will mostly be on modules that are written in Python. The cool thing about modules written in Python is that they are exceedingly straightforward to build. All you need to do is create a file that contains legitimate Python code and then give the file a name with a .py extension. That’s it! No special syntax or voodoo is necessary.

For example, suppose you have created a file called mod.py containing the following:

# mod.py

s = "If Comrade Napoleon says it, it must be right."

a = [100, 200, 300]

def foo(arg):

print(f'arg = {arg}')

class Foo:

pass

Several objects are defined in mod.py:

* s (a string)
* a (a list)
* foo() (a function)
* Foo (a class)

Assuming mod.py is in an appropriate location, which you will learn more about shortly, these objects can be accessed by **importing** the module as follows:

>>> import mod

>>> print(mod.s)

If Comrade Napoleon says it, it must be right.

>>> mod.a

[100, 200, 300]

>>> mod.foo(['quux', 'corge', 'grault'])

arg = ['quux', 'corge', 'grault']

>>> x = mod.Foo()

>>> x

<mod.Foo object at 0x03C181F0>

### The Module Search Path

Continuing with the above example, let’s take a look at what happens when Python executes the statement:

import mod

When the interpreter executes the above import statement, it searches for mod.py in a list of directories assembled from the following sources:

* The directory from which the input script was run or the **current directory** if the interpreter is being run interactively
* The list of directories contained in the PYTHONPATH environment variable, if it is set. (The format for PYTHONPATH is OS-dependent but should mimic the PATH environment variable.)
* An installation-dependent list of directories configured at the time Python is installed

The resulting search path is accessible in the Python variable sys.path, which is obtained from a module named sys:

>>> import sys

>>> sys.path

['', 'C:\\Users\\john\\Documents\\Python\\doc', 'C:\\Python36\\Lib\\idlelib',

'C:\\Python36\\python36.zip', 'C:\\Python36\\DLLs', 'C:\\Python36\\lib',

'C:\\Python36', 'C:\\Python36\\lib\\site-packages']

**Note:** The exact contents of sys.path are installation-dependent. The above will almost certainly look slightly different on your computer.

Thus, to ensure your module is found, you need to do one of the following:

* Put mod.py in the directory where the input script is located or the **current directory**, if interactive
* Modify the PYTHONPATH environment variable to contain the directory where mod.py is located before starting the interpreter
  + **Or:** Put mod.py in one of the directories already contained in the PYTHONPATH variable
* Put mod.py in one of the installation-dependent directories, which you may or may not have write-access to, depending on the OS

There is actually one additional option: you can put the module file in any directory of your choice and then modify sys.path at run-time so that it contains that directory. For example, in this case, you could put mod.py in directory C:\Users\john and then issue the following statements:

>>> sys.path.append(r'C:\Users\john')

>>> sys.path

['', 'C:\\Users\\john\\Documents\\Python\\doc', 'C:\\Python36\\Lib\\idlelib',

'C:\\Python36\\python36.zip', 'C:\\Python36\\DLLs', 'C:\\Python36\\lib',

'C:\\Python36', 'C:\\Python36\\lib\\site-packages', 'C:\\Users\\john']

>>> import mod

Once a module has been imported, you can determine the location where it was found with the module’s \_\_file\_\_ attribute:

>>> import mod

>>> mod.\_\_file\_\_

'C:\\Users\\john\\mod.py'

>>> import re

>>> re.\_\_file\_\_

'C:\\Python36\\lib\\re.py'

The directory portion of \_\_file\_\_ should be one of the directories in sys.path.

### The import Statement

**Module** contents are made available to the caller with the import statement. The import statement takes many different forms, shown below.

### **import <module\_name>**

The simplest form is the one already shown above:

import <module\_name>

Note that this *does not* make the module contents *directly* accessible to the caller. Each module has its own **private symbol table**, which serves as the global symbol table for all objects defined *in the module*. Thus, a module creates a separate **namespace**, as already noted.

The statement import <module\_name> only places <module\_name> in the caller’s symbol table. The *objects* that are defined in the module *remain in the module’s private symbol table*.

From the caller, objects in the module are only accessible when prefixed with <module\_name> via **dot notation**, as illustrated below.

After the following import statement, mod is placed into the local symbol table. Thus, mod has meaning in the caller’s local context:

>>> import mod

>>> mod

<module 'mod' from 'C:\\Users\\john\\Documents\\Python\\doc\\mod.py'>

But s and foo remain in the module’s private symbol table and are not meaningful in the local context:

>>> s

NameError: name 's' is not defined

>>> foo('quux')

NameError: name 'foo' is not defined

To be accessed in the local context, names of objects defined in the module must be prefixed by mod:

>>> mod.s

'If Comrade Napoleon says it, it must be right.'

>>> mod.foo('quux')

arg = quux

Several comma-separated modules may be specified in a single import statement:

import <module\_name>[, <module\_name> ...]

### from <module\_name> import <name(s)>

An alternate form of the import statement allows individual objects from the module to be imported directly into the caller’s symbol table:

from <module\_name> import <name(s)>

Following execution of the above statement, <name(s)> can be referenced in the caller’s environment without the <module\_name> prefix:

>>> from mod import s, foo

>>> s

'If Comrade Napoleon says it, it must be right.'

>>> foo('quux')

arg = quux

>>> from mod import Foo

>>> x = Foo()

>>> x

<mod.Foo object at 0x02E3AD50>

Because this form of import places the object names directly into the caller’s symbol table, any objects that already exist with the same name will be *overwritten*:

>>> a = ['foo', 'bar', 'baz']

>>> a

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

>>> from mod import a

>>> a

[100, 200, 300]

It is even possible to indiscriminately import everything from a module at one fell swoop:

from <module\_name> import \*

This will place the names of *all* objects from <module\_name> into the local symbol table, with the exception of any that begin with the underscore (\_) character.

**For example:**

>>> from mod import \*

>>> s

'If Comrade Napoleon says it, it must be right.'

>>> a

[100, 200, 300]

>>> foo

<function foo at 0x03B449C0>

>>> Foo

<class 'mod.Foo'>

This isn’t necessarily recommended in large-scale production code. It’s a bit dangerous because you are entering names into the local symbol table en masse. Unless you know them all well and can be confident there won’t be a conflict, you have a decent chance of overwriting an existing name inadvertently. However, this syntax is quite handy when you are just mucking around with the interactive interpreter, for testing or discovery purposes, because it quickly gives you access to everything a module has to offer without a lot of typing.

### from <module\_name> import <name> as <alt\_name>

It is also possible to import individual objects but enter them into the local symbol table with alternate names:

from <module\_name> import <name> as <alt\_name>[, <name> as <alt\_name> …]

This makes it possible to place names directly into the local symbol table but avoid conflicts with previously existing names:

>>> s = 'foo'

>>> a = ['foo', 'bar', 'baz']

>>> from mod import s as string, a as alist

>>> s

'foo'

>>> string

'If Comrade Napoleon says it, it must be right.'

>>> a

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

>>> alist

[100, 200, 300]

### import <module\_name> as <alt\_name>

You can also import an entire module under an alternate name:

import <module\_name> as <alt\_name>

>>> import mod as my\_module

>>> my\_module.a

[100, 200, 300]

>>> my\_module.foo('qux')

arg = qux

Module contents can be imported from within a function definition. In that case, the import does not occur until the function is called:

>>> def bar():

... from mod import foo

... foo('corge')

...

>>> bar()

arg = corge

However, **Python 3** does not allow the indiscriminate import \* syntax from within a function:

>>> def bar():

... from mod import \*

...

SyntaxError: import \* only allowed at module level

Lastly, a try statement with an except ImportError clause can be used to guard against unsuccessful import attempts:

>>> try:

... # Non-existent module

... import baz

... except ImportError:

... print('Module not found')

...

Module not found

>>> try:

... # Existing module, but non-existent object

... from mod import baz

... except ImportError:

... print('Object not found in module')

...

Object not found in module

### The dir() Function

The built-in function dir() returns a list of defined names in a namespace. Without arguments, it produces an alphabetically sorted list of names in the current **local symbol table**:

>>> dir()

['\_\_annotations\_\_', '\_\_builtins\_\_', '\_\_doc\_\_', '\_\_loader\_\_', '\_\_name\_\_',

'\_\_package\_\_', '\_\_spec\_\_']

>>> qux = [1, 2, 3, 4, 5]

>>> dir()

['\_\_annotations\_\_', '\_\_builtins\_\_', '\_\_doc\_\_', '\_\_loader\_\_', '\_\_name\_\_',

'\_\_package\_\_', '\_\_spec\_\_', 'qux']

>>> class Bar():

... pass

...

>>> x = Bar()

>>> dir()

['Bar', '\_\_annotations\_\_', '\_\_builtins\_\_', '\_\_doc\_\_', '\_\_loader\_\_', '\_\_name\_\_',

'\_\_package\_\_', '\_\_spec\_\_', 'qux', 'x']

Note how the first call to dir() above lists several names that are automatically defined and already in the namespace when the interpreter starts. As new names are defined (qux, Bar, x), they appear on subsequent invocations of dir().

This can be useful for identifying what exactly has been added to the namespace by an import statement:

>>> dir()

['\_\_annotations\_\_', '\_\_builtins\_\_', '\_\_doc\_\_', '\_\_loader\_\_', '\_\_name\_\_',

'\_\_package\_\_', '\_\_spec\_\_']

>>> import mod

>>> dir()

['\_\_annotations\_\_', '\_\_builtins\_\_', '\_\_doc\_\_', '\_\_loader\_\_', '\_\_name\_\_',

'\_\_package\_\_', '\_\_spec\_\_', 'mod']

>>> mod.s

'If Comrade Napoleon says it, it must be right.'

>>> mod.foo([1, 2, 3])

arg = [1, 2, 3]

>>> from mod import a, Foo

>>> dir()

['Foo', '\_\_annotations\_\_', '\_\_builtins\_\_', '\_\_doc\_\_', '\_\_loader\_\_', '\_\_name\_\_',

'\_\_package\_\_', '\_\_spec\_\_', 'a', 'mod']

>>> a

[100, 200, 300]

>>> x = Foo()

>>> x

<mod.Foo object at 0x002EAD50>

>>> from mod import s as string

>>> dir()

['Foo', '\_\_annotations\_\_', '\_\_builtins\_\_', '\_\_doc\_\_', '\_\_loader\_\_', '\_\_name\_\_',

'\_\_package\_\_', '\_\_spec\_\_', 'a', 'mod', 'string', 'x']

>>> string

'If Comrade Napoleon says it, it must be right.'

When given an argument that is the name of a module, dir() lists the names defined in the module:

>>> import mod

>>> dir(mod)

['Foo', '\_\_builtins\_\_', '\_\_cached\_\_', '\_\_doc\_\_', '\_\_file\_\_', '\_\_loader\_\_',

'\_\_name\_\_', '\_\_package\_\_', '\_\_spec\_\_', 'a', 'foo', 's']

>>> dir()

['\_\_annotations\_\_', '\_\_builtins\_\_', '\_\_doc\_\_', '\_\_loader\_\_', '\_\_name\_\_',

'\_\_package\_\_', '\_\_spec\_\_']

>>> from mod import \*

>>> dir()

['Foo', '\_\_annotations\_\_', '\_\_builtins\_\_', '\_\_doc\_\_', '\_\_loader\_\_', '\_\_name\_\_',

'\_\_package\_\_', '\_\_spec\_\_', 'a', 'foo', 's']

### Executing a Module as a Script

Any .py file that contains a **module** is essentially also a Python **script**, and there isn’t any reason it can’t be executed like one.

Here again is mod.py as it was defined above:

# mod.py

s = "If Comrade Napoleon says it, it must be right."

a = [100, 200, 300]

def foo(arg):

print(f'arg = {arg}')

class Foo:

pass

This can be run as a script:

C:\Users\john\Documents>python mod.py

C:\Users\john\Documents>

There are no errors, so it apparently worked. Granted, it’s not very interesting. As it is written, it only defines objects. It doesn’t do anything with them, and it doesn’t generate any output.

Let’s modify the above Python module so it does generate some output when run as a script:

# mod.py

s = "If Comrade Napoleon says it, it must be right."

a = [100, 200, 300]

def foo(arg):

print(f'arg = {arg}')

class Foo:

pass

print(s)

print(a)

foo('quux')

x = Foo()

print(x)

Now it should be a little more interesting:

C:\Users\john\Documents>python mod.py

If Comrade Napoleon says it, it must be right.

[100, 200, 300]

arg = quux

<\_\_main\_\_.Foo object at 0x02F101D0>

Unfortunately, now it also generates output when imported as a module:

>>> import mod

If Comrade Napoleon says it, it must be right.

[100, 200, 300]

arg = quux

<mod.Foo object at 0x0169AD50>

This is probably not what you want. It isn’t usual for a module to generate output when it is imported.

Wouldn’t it be nice if you could distinguish between when the file is loaded as a module and when it is run as a standalone script?

Ask and ye shall receive.

When a .py file is imported as a module, Python sets the special **dunder** variable \_\_name\_\_ to the name of the module. However, if a file is run as a standalone script, \_\_name\_\_ is (creatively) set to the string '\_\_main\_\_'. Using this fact, you can discern which is the case at run-time and alter behavior accordingly:

# mod.py

s = "If Comrade Napoleon says it, it must be right."

a = [100, 200, 300]

def foo(arg):

print(f'arg = {arg}')

class Foo:

pass

if (\_\_name\_\_ == '\_\_main\_\_'):

print('Executing as standalone script')

print(s)

print(a)

foo('quux')

x = Foo()

print(x)

Now, if you run as a script, you get output:

C:\Users\john\Documents>python mod.py

Executing as standalone script

If Comrade Napoleon says it, it must be right.

[100, 200, 300]

arg = quux

<\_\_main\_\_.Foo object at 0x03450690>

But if you import as a module, you don’t:

>>> import mod

>>> mod.foo('grault')

arg = grault

Modules are often designed with the capability to run as a standalone script for purposes of testing the functionality that is contained within the module. This is referred to as **unit testing.** For example, suppose you have created a module fact.py containing a **factorial** function, as follows:

# fact.py

def fact(n):

return 1 if n == 1 else n \* fact(n-1)

if (\_\_name\_\_ == '\_\_main\_\_'):

import sys

if len(sys.argv) > 1:

print(fact(int(sys.argv[1])))

The file can be treated as a module, and the fact() function imported:

>>> from fact import fact

>>> fact(6)

720

But it can also be run as a standalone by passing an integer argument on the command-line for testing:

C:\Users\john\Documents>python fact.py 6

720

### Reloading a Module

For reasons of efficiency, a module is only loaded once per interpreter session. That is fine for function and class definitions, which typically make up the bulk of a module’s contents. But a module can contain executable statements as well, usually for initialization. Be aware that these statements will only be executed the first time a module is imported.

Consider the following file mod.py:

# mod.py

a = [100, 200, 300]

print('a =', a)

>>> import mod

a = [100, 200, 300]

>>> import mod

>>> import mod

>>> mod.a

[100, 200, 300]

The print() statement is not executed on subsequent imports. (For that matter, neither is the assignment statement, but as the final display of the value of mod.a shows, that doesn’t matter. Once the assignment is made, it sticks.)

If you make a change to a module and need to reload it, you need to either restart the interpreter or use a function called reload() from module importlib:

>>> import mod

a = [100, 200, 300]

>>> import mod

>>> import importlib

>>> importlib.reload(mod)

a = [100, 200, 300]

<module 'mod' from 'C:\\Users\\john\\Documents\\Python\\doc\\mod.py'>

## Python Packages

Suppose you have developed a very large application that includes many modules. As the number of modules grows, it becomes difficult to keep track of them all if they are dumped into one location. This is particularly so if they have similar names or functionality. You might wish for a means of grouping and organizing them.

**Packages** allow for a hierarchical structuring of the module namespace using **dot notation**. In the same way that **modules** help avoid collisions between global variable names, **packages** help avoid collisions between module names.

Creating a **package** is quite straightforward, since it makes use of the operating system’s inherent hierarchical file structure. Consider the following arrangement:



Here, there is a directory named pkg that contains two modules, mod1.py and mod2.py. The contents of the modules are:

**mod1.py**

def foo():

print('[mod1] foo()')

class Foo:

pass

**mod2.py**

def bar():

print('[mod2] bar()')

class Bar:

pass

Given this structure, if the pkg directory resides in a location where it can be found (in one of the directories contained in sys.path), you can refer to the two **modules** with **dot notation** (pkg.mod1, pkg.mod2) and import them with the syntax you are already familiar with:

import <module\_name>[, <module\_name> ...]

>>> import pkg.mod1, pkg.mod2

>>> pkg.mod1.foo()

[mod1] foo()

>>> x = pkg.mod2.Bar()

>>> x

<pkg.mod2.Bar object at 0x033F7290>

from <module\_name> import <name(s)>

>>> from pkg.mod1 import foo

>>> foo()

[mod1] foo()

from <module\_name> import <name> as <alt\_name>

>>> from pkg.mod2 import Bar as Qux

>>> x = Qux()

>>> x

<pkg.mod2.Bar object at 0x036DFFD0>

You can import modules with these statements as well:

from <package\_name> import <modules\_name>[, <module\_name> ...]

from <package\_name> import <module\_name> as <alt\_name>

>>> from pkg import mod1

>>> mod1.foo()

[mod1] foo()

>>> from pkg import mod2 as quux

>>> quux.bar()

[mod2] bar()

You can technically import the package as well:

>>> import pkg

>>> pkg

<module 'pkg' (namespace)>

But this is of little avail. Though this is, strictly speaking, a syntactically correct Python statement, it doesn’t do much of anything useful. In particular, it *does not place* any of the modules in pkg into the local namespace:

>>> pkg.mod1

Traceback (most recent call last):

File "<pyshell#34>", line 1, in <module>

pkg.mod1

AttributeError: module 'pkg' has no attribute 'mod1'

>>> pkg.mod1.foo()

Traceback (most recent call last):

File "<pyshell#35>", line 1, in <module>

pkg.mod1.foo()

AttributeError: module 'pkg' has no attribute 'mod1'

>>> pkg.mod2.Bar()

Traceback (most recent call last):

File "<pyshell#36>", line 1, in <module>

pkg.mod2.Bar()

AttributeError: module 'pkg' has no attribute 'mod2'

To actually import the modules or their contents, you need to use one of the forms shown above.

### Package Initialization

If a file named \_\_init\_\_.py is present in a package directory, it is invoked when the package or a module in the package is imported. This can be used for execution of package initialization code, such as initialization of package-level data.

For example, consider the following \_\_init\_\_.py file:

***\_\_init\_\_.py***

print(f'Invoking \_\_init\_\_.py for {\_\_name\_\_}')

A = ['quux', 'corge', 'grault']

Let’s add this file to the pkg directory from the above example:



Now when the package is imported, the global list A is initialized:

>>> import pkg

Invoking \_\_init\_\_.py for pkg

>>> pkg.A

['quux', 'corge', 'grault']

A **module** in the package can access the global variable by importing it in turn:

***mod1.py***

def foo():

from pkg import A

print('[mod1] foo() / A = ', A)

class Foo:

pass

>>> from pkg import mod1

Invoking \_\_init\_\_.py for pkg

>>> mod1.foo()

[mod1] foo() / A = ['quux', 'corge', 'grault']

\_\_init\_\_.py can also be used to effect automatic importing of modules from a package. For example, earlier you saw that the statement import pkg only places the name pkg in the caller’s local symbol table and doesn’t import any modules. But if \_\_init\_\_.py in the pkg directory contains the following:

***\_\_init\_\_.py***

print(f'Invoking \_\_init\_\_.py for {\_\_name\_\_}')

import pkg.mod1, pkg.mod2

then when you execute import pkg, modules mod1 and mod2 are imported automatically:

>>> import pkg

Invoking \_\_init\_\_.py for pkg

>>> pkg.mod1.foo()

[mod1] foo()

>>> pkg.mod2.bar()

[mod2] bar()

**Note:** Much of the Python documentation states that an \_\_init\_\_.py file **must** be present in the package directory when creating a package. This was once true. It used to be that the very presence of \_\_init\_\_.py signified to Python that a package was being defined. The file could contain initialization code or even be empty, but it **had** to be present.

Starting with **Python 3.3**, Implicit Namespace Packages were introduced. These allow for the creation of a package without any \_\_init\_\_.py file. Of course, it **can** still be present if package initialization is needed. But it is no longer required.

### Importing \* From a Package

For the purposes of the following discussion, the previously defined package is expanded to contain some additional modules:



There are now four modules defined in the pkg directory. Their contents are as shown below:

***mod1.py***

def foo():

print('[mod1] foo()')

class Foo:

pass

***mod2.py***

def bar():

print('[mod2] bar()')

class Bar:

pass

***mod3.py***

def baz():

print('[mod3] baz()')

class Baz:

pass

***mod4.py***

def qux():

print('[mod4] qux()')

class Qux:

pass

(Imaginative, aren’t they?)

You have already seen that when import \* is used for a **module**, all objects from the module are imported into the local symbol table, except those whose names begin with an underscore, as always:

>>> dir()

['\_\_annotations\_\_', '\_\_builtins\_\_', '\_\_doc\_\_', '\_\_loader\_\_', '\_\_name\_\_',

'\_\_package\_\_', '\_\_spec\_\_']

>>> from pkg.mod3 import \*

>>> dir()

['Baz', '\_\_annotations\_\_', '\_\_builtins\_\_', '\_\_doc\_\_', '\_\_loader\_\_', '\_\_name\_\_',

'\_\_package\_\_', '\_\_spec\_\_', 'baz']

>>> baz()

[mod3] baz()

>>> Baz

<class 'pkg.mod3.Baz'>

The analogous statement for a **package** is this:

from <package\_name> import \*

What does that do?

>>> dir()

['\_\_annotations\_\_', '\_\_builtins\_\_', '\_\_doc\_\_', '\_\_loader\_\_', '\_\_name\_\_',

'\_\_package\_\_', '\_\_spec\_\_']

>>> from pkg import \*

>>> dir()

['\_\_annotations\_\_', '\_\_builtins\_\_', '\_\_doc\_\_', '\_\_loader\_\_', '\_\_name\_\_',

'\_\_package\_\_', '\_\_spec\_\_']

Hmph. Not much. You might have expected (assuming you had any expectations at all) that Python would dive down into the package directory, find all the modules it could, and import them all. But as you can see, by default that is not what happens.

Instead, Python follows this convention: if the \_\_init\_\_.py file in the **package** directory contains a **list** named \_\_all\_\_, it is taken to be a list of modules that should be imported when the statement from <package\_name> import \* is encountered.

For the present example, suppose you create an \_\_init\_\_.py in the pkg directory like this:

***pkg/\_\_init\_\_.py***

\_\_all\_\_ = [

'mod1',

'mod2',

'mod3',

'mod4'

]

Now from pkg import \* imports all four modules:

>>> dir()

['\_\_annotations\_\_', '\_\_builtins\_\_', '\_\_doc\_\_', '\_\_loader\_\_', '\_\_name\_\_',

'\_\_package\_\_', '\_\_spec\_\_']

>>> from pkg import \*

>>> dir()

['\_\_annotations\_\_', '\_\_builtins\_\_', '\_\_doc\_\_', '\_\_loader\_\_', '\_\_name\_\_',

'\_\_package\_\_', '\_\_spec\_\_', 'mod1', 'mod2', 'mod3', 'mod4']

>>> mod2.bar()

[mod2] bar()

>>> mod4.Qux

<class 'pkg.mod4.Qux'>

Using import \* still isn’t considered terrific form, any more for **packages** than for **modules**. But this facility at least gives the creator of the package some control over what happens when import \* is specified. (In fact, it provides the capability to disallow it entirely, simply by declining to define \_\_all\_\_ at all. As you have seen, the default behavior for packages is to import nothing.)

By the way, \_\_all\_\_ can be defined in a **module** as well and serves the same purpose: to control what is imported with import \*. For example, modify mod1.py as follows:

***pkg/mod1.py***

\_\_all\_\_ = ['foo']

def foo():

print('[mod1] foo()')

class Foo:

pass

Now an import \* statement from pkg.mod1 will only import what is contained in \_\_all\_\_:

>>> dir()

['\_\_annotations\_\_', '\_\_builtins\_\_', '\_\_doc\_\_', '\_\_loader\_\_', '\_\_name\_\_',

'\_\_package\_\_', '\_\_spec\_\_']

>>> from pkg.mod1 import \*

>>> dir()

['\_\_annotations\_\_', '\_\_builtins\_\_', '\_\_doc\_\_', '\_\_loader\_\_', '\_\_name\_\_',

'\_\_package\_\_', '\_\_spec\_\_', 'foo']

>>> foo()

[mod1] foo()

>>> Foo

Traceback (most recent call last):

File "<pyshell#37>", line 1, in <module>

Foo

NameError: name 'Foo' is not defined

foo() (the function) is now defined in the local namespace, but Foo (the class) is not, because the latter is not in \_\_all\_\_.

In summary, \_\_all\_\_ is used by both **packages** and **modules** to control what is imported when import \* is specified. But *the default behavior differs*:

* For a package, when \_\_all\_\_ is not defined, import \* does not import anything.
* For a module, when \_\_all\_\_ is not defined, import \* imports everything (except—you guessed it—names starting with an underscore).

### Subpackages

Packages can contain nested **subpackages** to arbitrary depth. For example, let’s make one more modification to the example **package** directory as follows:



The four modules (mod1.py, mod2.py, mod3.py and mod4.py) are defined as previously. But now, instead of being lumped together into the pkg directory, they are split out into two **subpackage** directories, sub\_pkg1 and sub\_pkg2.

Importing still works the same as shown previously. Syntax is similar, but additional **dot notation** is used to separate **package** name from **subpackage** name:

>>> import pkg.sub\_pkg1.mod1

>>> pkg.sub\_pkg1.mod1.foo()

[mod1] foo()

>>> from pkg.sub\_pkg1 import mod2

>>> mod2.bar()

[mod2] bar()

>>> from pkg.sub\_pkg2.mod3 import baz

>>> baz()

[mod3] baz()

>>> from pkg.sub\_pkg2.mod4 import qux as grault

>>> grault()

[mod4] qux()

In addition, a module in one **subpackage** can reference objects in a **sibling subpackage** (in the event that the sibling contains some functionality that you need). For example, suppose you want to import and execute function foo() (defined in module mod1) from within module mod3. You can either use an **absolute import**:

***pkg/sub\_\_pkg2/mod3.py***

def baz():

print('[mod3] baz()')

class Baz:

pass

from pkg.sub\_pkg1.mod1 import foo

foo()

>>> from pkg.sub\_pkg2 import mod3

[mod1] foo()

>>> mod3.foo()

[mod1] foo()

Or you can use a **relative import**, where .. refers to the package one level up. From within mod3.py, which is in subpackage sub\_pkg2,

* .. evaluates to the parent package (pkg), and
* ..sub\_pkg1 evaluates to subpackage sub\_pkg1 of the parent package.

***pkg/sub\_\_pkg2/mod3.py***

def baz():

print('[mod3] baz()')

class Baz:

pass

from .. import sub\_pkg1

print(sub\_pkg1)

from ..sub\_pkg1.mod1 import foo

foo()

>>> from pkg.sub\_pkg2 import mod3

<module 'pkg.sub\_pkg1' (namespace)>

[mod1] foo()

# PyPl & pip

# Pylint

# 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/>

<https://realpython.com/python-exceptions/#the-try-and-except-block-handling-exceptions>

<https://realpython.com/python-lists-tuples/>

<https://www.geeksforgeeks.org/python-list/>

<https://realpython.com/python-sets/>

<https://careerkarma.com/blog/python-collections/>

<https://www.geeksforgeeks.org/python-collections-module/>

<https://realpython.com/python-datetime/>

<https://www.w3schools.com/python/python_regex.asp>

<https://realpython.com/regex-python/>

<https://www.geeksforgeeks.org/file-handling-python/>

<https://realpython.com/python-modules-packages/>

<https://realpython.com/python-logging/>