**WHAT IS PYTHON?**

**Python is an interpreted, high-level, general-purpose programming language.**

**It has efficient high-level data structures and a simple but effective approach to object-oriented programming.**

* Python’s elegant syntax and dynamic typing, together with its interpreted nature, make it an ideal language for scripting and rapid application development in many areas on most platforms.
* Python's design philosophy emphasizes code readability with its notable use of significant whitespace.

# ****PYTHON SETUP****

## **WINDOWS**

 avigate to: <https://www.python.org/downloads/windows/>

 Click on "Latest Python 3 Release".

 On the next page, scroll down until you see the "Files" section.

 Click on the "Windows x86-64 executable installer" link.

 This will download the installer for Python.

 **Once it downloads, simply run through the installer.**

**NSTALL PYTHON**

* Most Linux distributions come preloaded with Python.
* You can verify this by typing 'python3' in a terminal.
* If your distribution does not ship with python for some reason, you can fetch it from your package manager.

# Red Hat, CentOS, Fedora

sudo dnf install python3

# Debian, Ubuntu, Pop OS

sudo apt install python3

# openSUSE or SUSE

sudo zypper in python3

**Installing VSCode**

**Steps To Download and Install:**

1. Go to: https://code.visualstudio.com/
2. Scroll down until you see the section shown above.
3. Click on the appropriate version based on your operating system.
4. Once it downloads, run the installer.

**Install the Python extension by Microsoft**

**HELLO WORLD**

|  |  |
| --- | --- |
|  | print("Hello, World!") |

**Steps:**

1. Create a new file. ( CTRL + N )
2. Enter the code shown above in the file.
3. Save the file as "hello.py". ( CTRL + S )
4. Execute the file from your terminal using 'python3 hello.py'  
   (or)
5. Execute the file from vscode by clicking the green 'play' button on the top.

# ****VARIABLES****

**Variables are containers that you can store values in. You can declare a variable by giving it a name followed by any value you want to assign to it.**

|  |  |
| --- | --- |
|  | myVariable = 5 |

**Note**: Python is case sensitive — myVariable is a different variable to myvariable. If you are getting problems in your code, check the casing!

# ****VARIABLES****

|  |  |
| --- | --- |
|  | myVariable |

**You can retrieve the value by just calling the variable by name:**

**After giving a variable a value, you can later choose to change it:**

|  |  |
| --- | --- |
|  | myVariable = 5 |
|  | myVariable # Output: 5 |
|  | myVariable = 9 |
|  | myVariable # Output: 9 |

# VARIABLE DATA TYPES

**Variables may hold values that have different data types:**

|  |  |  |
| --- | --- | --- |
| **Variable Type** | **Explanation Example** | **Example** |
| String | A String is a sequence of text. myVariable = "Bob"; Strings are enclosed in quotes. | myVariable = "Bob"; |
| Number | Any numerical value. myVariable = 10; Numbers don't have quotes. myVariable = 10.5; | myVariable = 10; myVariable = 10.5; |

# ****COMMENTS****

**Comments are essentially short snippets of text that can be added in-between code which is ignored by the Python interpreter.**

**This is how you can add comments in Python.**

|  |  |
| --- | --- |
|  | # This is a comment |
|  | print("Hello, World!") |

**Comments can be placed at the end of a line, and Python will ignore the rest of the line:**

|  |  |
| --- | --- |
|  | print("Hello, World!") # This is a comment |

**Python will ignore string literals that are not assigned to a variable.**

**You can add a multiline string (triple quotes) in your code, and place your comment inside it.**

"""

This is a comment

written in

more than just one line

"""

print("Hello, World!")

**NUMBERS**

**There are 3 numeric types in Python:**

* int
* float
* complex

x = 1 # int

y = 2.8 # float

z = 1j # complex

**You can check the type of variables by using the 'type()' function as shown below:**

x = 1 # int

y = 2.8 # float

z = 1j # complex

print(type(x)) # Output: <class 'int'>

print(type(y)) # Output: <class 'float'>

print(type(z)) # Output: <class 'complex'>

# ****STRINGS****

Strings in Python are simply bits of text that are surrounded by either single or double quotation marks.

'hello' is the same as "hello"

**You can display a string literal with the print() function:**

print("Hello")

print('Hello')

## ASSIGNING STRINGS TO VARIABLES

**Assigning a string to a variable is done with the variable name followed by an equal sign and the string:**

a = "Hello"

print(a) # Output: "Hello"

## MULTI-LINE STRINGS

**You can assign a multiline string to a variable by using three quotes:**

a = """Lorem ipsum dolor sit amet,

consectetur adipiscing elit,

sed do eiusmod tempor incididunt

ut labore et dolore magna aliqua."""

print(a)

## STRINGS WORK LIKE ARRAYS

**Strings in Python can be thought of as arrays of characters.**

Python does not have a specific character data type, so a single character is simply a string with a length of 1.

**Square brackets can be used to access elements of the string:**

a = "Hello, World!"

print(a[1]) # Output: "e"

Remember, Array indices start from '0'.

## SLICING STRINGS

**You can return a range of characters by using the slice syntax.**

Specify the start index and the end index, separated by a colon, to return a part of the string.

b = "Hello, World!"

print(b[2:5]) # Output: "llo"

Remember, Array indices start from '0'.

## STRING LENGTH

**To get the length of a string, use the len() function:**

a = "Hello, World!"

print(len(a)) # Output: 13

## LOWER AND UPPER

**The lower() method returns the string in lower case:**

a = "Hello, World!"

print(a.lower()) # Output: "hello, world!"

**The upper() method returns the string in upper case:**

a = "Hello, World!"

print(a.upper()) # Output: "HELLO, WORLD!"

## REPLACE AND SPLIT

**The replace() method replaces a string with another string:**

a = "Hello, World!"

print(a.replace("H", "J")) # Output: "Jello, World!"

**The split() method splits the string into substrings if it finds instances of the separator:**

a = "Hello, World!"

print(a.split(",")) # returns ['Hello', ' World!']

## CHECK STRING

**To check if a certain phrase or character is present in a string, we can use the keywords in or not in.**

txt = "The rain in Spain stays mainly in the plain"

x = "ain" in txt

y = "ain" not in txt

print(x) # True

print(y) # False

## STRING CONCATENATION

**To concatenate (combine) two strings you can use the + operator.**

a = "Hello"

b = " World"

c = a + b

print(c) # "Hello World"

## STRING FORMAT

**Something interesting to note is that you cannot combine Strings and Numbers as shown below:**

age = 36

txt = "My name is John, I am " + age

print(txt)

## STRING FORMAT

**But we can combine strings and numbers by using the format() method.**

**The format() method takes the passed arguments, formats them, and places them in the string where the placeholders {} are:**

age = 36

txt = "My name is John, and I am {}"

print(txt.format(age))

# Output: My name is John, and I am 36

## STRING FORMAT

**The format() method takes unlimited number of arguments, and are placed into the respective placeholders:**

quantity = 3

itemno = 567

price = 49.95

myorder = "I want {} pieces of item {} for {} dollars."

print(myorder.format(quantity, itemno, price))

# Output: I want 3 pieces of item 567 for 49.95 dollars.

## STRING FORMAT

**You can use index numbers {0} to be sure the arguments are placed in the correct placeholders:**

quantity = 3

itemno = 567

price = 49.95

myorder = "I want to pay {2} dollars for {0} pieces of item {1}."

print(myorder.format(quantity, itemno, price))

# Output: I want to pay 49.95 dollars for 3 pieces of item 567.

**CASTING**

**There may be times when you want to specify a type on to a variable. This can be done with casting.**

**Casting in Python is therefore done using constructor functions:**

* **int()** - constructs an integer number from an integer literal, a float literal (by rounding down to the previous whole number), or a string literal (providing the string represents a whole number)
* **float()** - constructs a float number from an integer literal, a float literal or a string literal (providing the string represents a float or an integer)
* **str()** - constructs a string from a wide variety of data types, including strings, integer literals and float literals

a = int(1) # a will be 1

b = int(2.8) # b will be 2

c = int("3") # c will be 3

x = float(1) # x will be 1.0

y = float(2.8) # y will be 2.8

z = float("3") # z will be 3.0

w = float("4.2") # w will be 4.2

j = str("s1") # j will be 's1'

k = str(2) # k will be '2'

l = str(3.0) # l will be '3.0'

**An operator is a mathematical symbol which produces a result based on two values (or variables).**

In the following table you can see some of the simplest operators, along with some examples to try out on your own!

# OPERATORS

|  |  |  |  |
| --- | --- | --- | --- |
| **Operator** | **Explanation** | **Symbol Example** | **Example** |
| Addition | Used to add two numbers | + 6 + 9 | 6 + 9 |
| Subtraction | Arithmetic Operator | * 9 - 3 | 9 - 3 |
| Multiplication | Arithmetic Operator | \* 6 \* 5 | 6 \* 5 |
| Division | Does Division: Quotient | / 14/2   # returns 7 | 14/2   # returns 7 |
| Modulus | Does Division: Remainder | % 14%2 # returns 0 | 14%2 # retur |

# OPERATORS

|  |  |  |  |
| --- | --- | --- | --- |
| **Operator** | **Explanation Symbol Example** | **Symbol** | **Example** |
| Exponent | Calculates the exponent \*\* 2 \*\* 3 # returns 8 | \*\* | 2 \*\* 3 # returns 8 |
| Floor Division | Performs division and returns the lowest // 9/2   # returns 4.5 integer to the result. 9//2 # returns 4 | // | 9/2   # returns 4.5 9//2 # returns 4 |

# OPERATORS

## ASSIGNMENT OPERATORS

|  |  |  |
| --- | --- | --- |
| **Operator** | **Example** | **Same As** |
| = | x = 5 | x = 5 |
| += | x += 3 | x = x + 3 |
| -= | x -= 3 | x = x - 3 |
| \*= | x \*= 5 | x = x \* 5 |
| /= | x /= 5 | x = x / 5 |
| %= | x %= 2 | x = x % 2 |
| //= | x //= 7 | x = x // 7 |
| \*\*= | x \*\*= 4 | x = x \*\* 4 |

**You can combine the operators you just saw with the '=' sign to create shortcuts for assigning values as shown below!**

## COMPARISON OPERATORS

|  |  |  |  |
| --- | --- | --- | --- |
| **Operator** | **Name** | **Example Result ( x = 5, y = 7 )** | **Result ( x = 5, y = 7 )** |
| == | Equal | x == y False | False |
| != | Not Equal | x != y True | True |
| > | Greater Than | x > y False | False |
| < | Less Than | x < y True | True |
| >= | Greater Than or Equal To | x >= y False | False |
| <= | Less Than or Equal To | x <= y True | True |

**Comparison operators are used to compare two values.**

**They return either True or False.**

## LOGICAL OPERATORS

|  |  |  |  |
| --- | --- | --- | --- |
| **Operator** | **Description** | **Example Result ( x = 4 )** | **Result ( x = 4 )** |
| and | Returns True if both statements are true | x < 5 and  x < 10 True | True |
| or | Returns True if one of the statements is true | x < 5 or x < 4 True | True |
| not | Reverse the result, returns False if the result is true | not(x < 5 and x < 10) False | False |

**Logical operators are used to combine conditional statements.**

**They return either True or False.**

## MEMBERSHIP OPERATORS

|  |  |  |  |
| --- | --- | --- | --- |
| **Operator** | **Description** | **Example Result ( x = 4 , y = [2, 4, 6] )** | **Result ( x = 4 , y = [2, 4, 6] )** |
| in | Returns True if a sequence with the specified value is present in the object | x in y True | True |
| not in | Returns True if a sequence with the specified value is not present in the object | x not in y False | False |

**Membership operators are used to test if a sequence is presented in an object:.**

**They return either True or False**

# ****USER INPUT****

**Python allows for user input.**

That means we are able to ask the user for input.

|  |  |
| --- | --- |
|  | username = input("Enter username:") |
|  | print("Username is: " + username) |
|  | # Output: |
|  | # Enter username: abhiram |
|  | # Username is: abhiram |
|  | CONTROL FLOW |

**A conditional statement is a set of commands that executes if a specified condition is true.**

Unlike other languages, Python only provides **'if'** and **'else'** statements to achieve this.

## IF STATEMENT

Use the **if statement** to execute a statement if a logical condition is **true**.

 Here, the condition can be any expression that evaluates to true or false.

 The statement\_1 on line 2 will ONLY execute if said condition evaluates to true.

if condition:

statement\_1

## ELIF STATEMENT

Use the **elif statement** to setup another condition to check if the **first condition evaluates to 'false'**.

 Here, if the first condition evaluates to false, the program checks the second condition.

 The **statement\_2** on line 4 will **ONLY execute if the elif condition is true**.

 **elif stands for 'Else If'.**

## ELSE STATEMENT

Use the **else**statement to **execute a statement when all previous conditions are false**.

 Here, if both the previous conditions are false, then the else statement is executed.

 The **statement\_3** on line 6 will **ONLY execute if all previous conditions are false.**

if condition:

statement\_1

elif other\_condition:

statement\_2

else:

statement\_3

## INDENTATIONS

if condition:

statement\_1

elif other\_condition:

statement\_2

else:

statement\_3

**Important Notes:**

* **Pay close attention to the indentations in the above code.**
* **Indentations are the spaces in front of the highlighted lines.**
* **Python makes use of these indentations to define the scope of execution.**
* **Each indentation is *4 spaces wide*. You can use tabs as long as you set the tab-length to 4 spaces as well.**

## EXAMPLE

a = 5

b = 7

c = 9

if a > b:

if a > c:

print("a is the greatest")

else:

print("c is the greatest")

elif b > a:

if b > c:

print("b is the greatest")

else:

print("c is the greatest")

What do you think this code does?

# Output: "c is the greatest"

## EXAMPLE

|  |  |
| --- | --- |
|  | a = 5 |
|  | b = 7 |
|  | c = 9 |
|  |  |
|  | if a > b and a > c: |
|  | print("a is the greatest") |
|  | elif b > a and b > c: |
|  | print("b is the greatest") |
|  | elif c > a and c > b: |
|  | print("c is the greatest") |

Using logical operators, we can even write the same code like this:

## STOPPING THE EXECUTION

|  |  |
| --- | --- |
|  | should\_i\_exit = true |
|  |  |
|  | if should\_i\_exit == true: |
|  | raise SystemExit |

Sometimes, we will want to halt the execution of a program when a condition is met. You can stop execution as shown below:

The code on line 4 tells the python interpreter to stop executing the program. So, the program exits gracefully once this line is encountered.

# ****LOOPS****

**Occasionally, you will want to execute the same code multiple amount of times. The process of doing this is called 'Looping'.**

Python has two primitive loop commands:

* **while** loops
* **for** loops

## THE WHILE LOOP

The **while** loop lets us execute a set of statements as long as a condition is true.

# Output:

# 0

# 1

# 2

# 3

# 4

a = 0

while a < 5:

print(a)

a += 1

## THE BREAK STATEMENT

With the **break** statement we can stop the loop even if the while condition is true:

a = 0

while a < 5:

print(a)

if a == 3:

break

a += 1

# Output:

# 0

# 1

# 2

# 3

## THE CONTINUE STATEMENT

With the **continue** statement we can stop the current iteration, and continue with the next:

a = 0

while a < 5:

a += 1

if a == 3:

continue

print(a)

# Output:

# 1

# 2

# 4

# 5

## THE ELSE STATEMENT

With the **else** statement we can run a block of code once when the condition no longer is true:

a = 1

while a < 5:

print(a)

a += 1

else:

print("a is no longer less than 5")

# Output:

# 0

# 1

# 2

# 3

# 4

# a is no longer less than 5

## THE FOR LOOP

A **for loop** is used for iterating over a sequence (that is either a list, a tuple, a dictionary, a set, or a string).

* This is less like the for keyword in other programming languages, and works more like an iterator method as found in other object-orientated programming languages.
* **With the for loop we can execute a set of statements, once for each item in a list, tuple, set etc.**

fruits = ["apple", "banana", "cherry"]

for fruit in fruits:

print(fruit)

# Output:

# apple

# banana

# cherry

## FOR LOOP EXAMPLES

fruits = ["apple", "banana", "cherry"]

for fruit in fruits:

print(fruit)

# Output:

# apple

# banana

# cherry

for x in "banana":

print(x)

# Output:

# b

# a

# n

# a

# n

# a

## THE BREAK STATEMENT

With the **break** statement we can stop the loop before it has looped through all the items:

fruits = ["apple", "banana", "cherry"]

for fruit in fruits:

if fruit == "banana":

break

print(fruit)

# Output:

# apple

## THE CONTINUE STATEMENT

With the **continue** statement we can stop the current iteration of the loop, and continue with the next:

fruits = ["apple", "banana", "cherry"]

for fruit in fruits:

if fruit == "banana":

continue

print(fruit)

# Output:

# apple

# cherry

## THE RANGE() FUNCTION

**To loop through a set of code a specified number of times, we can use the range() function.**

The range() function returns a sequence of numbers, starting from 0 by default, and increments by 1 (by default), and ends at a specified number.

for x in range(6):

print(x)

# Output:

# 0

# 1

# 2

# 3

# 4

# 5

## THE RANGE() FUNCTION

The range() function **defaults to 0 as a starting value**, however it is possible to specify the starting value by adding a parameter: **range(2, 6), which means values from 2 to 6 (but not including 6):**

for x in range(2,6):

print(x)

# Output:

# 2

# 3

# 4

# 5

## THE RANGE() FUNCTION

The range() function defaults to increment the sequence by 1, however it is possible to specify the increment value by adding a third parameter:

**range(2, 8, 2):**

|  |  |
| --- | --- |
|  | for x in range(2,8,2): |
|  | print(x) |
|  | # Output: |
|  | # 2 |
|  | # 4 |
|  | # 6 |

## THE ELSE STATEMENT

The **else keyword** in a for loop specifies a block of code to be executed when the loop is finished:

|  |  |
| --- | --- |
|  | for x in range(6): |
|  | print(x) |
|  | else: |
|  | print("Finally finished!") |
|  | # Output: |
|  | # 0 |
|  | # 1 |
|  | # 2 |
|  | # 3 |
|  | # 4 |
|  | # 5 |
|  | # Finally finished! |

## NESTED LOOPS

**A nested loop is a loop inside a loop.**

The "inner loop" will be executed one time for each iteration of the "outer loop":

|  |  |
| --- | --- |
|  | adjectives = ["red", "big", "tasty"] |
|  | fruits = ["apple", "banana", "cherry"] |
|  |  |
|  | for adjective in adjectives: |
|  | for fruit in fruits: |
|  | print(adjective, fruit) |
|  | # Output: |
|  | # red apple |
|  | # red banana |
|  | # red cherry |
|  | # big apple |
|  | # big banana |
|  | # big cherry |
|  | # tasty apple |
|  | # tasty banana |
|  | # tasty cherry |

# ****LISTS****

## **COPY A LIST**

You cannot copy a list simply by typing list2 = list1, because: list2 will only be a **reference** to list1, and changes made in list1 will automatically also be made in list2.

**There are ways to make a copy, one way is to use the built-in List method copy().**

|  |  |
| --- | --- |
|  | example\_list = ["apple", "banana", "cherry"] |
|  | mylist = example\_list.copy() |
|  | print(mylist) |

## **JOINING MULTIPLE LISTS**

**There are several ways to join, or concatenate, two or more lists in Python.**

**One of the easiest ways are by using the + operator.**

|  |  |
| --- | --- |
|  | list1 = ["a", "b" , "c"] |
|  | list2 = [1, 2, 3] |
|  |  |
|  | list3 = list1 + list2 |
|  | print(list3) # Output: ["a", "b", "c", 1, 2, 3] |

## **THE LIST() CONSTRUCTOR**

**It is also possible to use the list() constructor to make a new list.**

|  |  |
| --- | --- |
|  | example\_list = list(("apple", "banana", "cherry")) # note the double round-brackets |
|  | print(example\_list) # Output: ["apple", "banana", "cherry"] |

**TUPLES**

**A tuple is a collection in which data is ordered and *unchangeable*.**

In Python tuples are written with round brackets.

|  |  |
| --- | --- |
|  | example\_tuple = ("apple", "banana", "cherry") |
|  | print(example\_tuple) |

## **ACCESSING ITEMS**

**You access the tuple items by referring to the index number:**

|  |  |
| --- | --- |
|  | example\_tuple = ("apple", "banana", "cherry") |
|  | print(example\_tuple[1]) # Output: "banana" |

**You can access the tuple items in reverse by using negative index values:**

|  |  |
| --- | --- |
|  | example\_tuple = ("apple", "banana", "cherry") |
|  | print(example\_tuple[-1]) # Output: "cherry" |

## **RANGE OF INDICES**

**You can specify a range of indices by specifying where to start and where to end the range.**

example\_tuple = ("apple", "banana", "cherry", "orange", "kiwi", "melon", "mango")

print(example\_tuple[2:5]) # Output: ("cherry", "orange", "kiwi")

## **RANGE OF INDICES**

**By leaving out the start value, the range will start at the first item:**

|  |  |
| --- | --- |
|  | example\_tuple = ("apple", "banana", "cherry", "orange", "kiwi", "melon", "mango") |
|  | print(example\_tuple[:2]) # Output: ("apple", "banana") |

**By leaving out the end value, the range will go on to the end of the tuple:**

|  |  |
| --- | --- |
|  | example\_tuple = ("apple", "banana", "cherry", "orange", "kiwi", "melon", "mango") |
|  | print(example\_tuple[5:]) # Output: ("melon", "mango") |

## **LOOPING THROUGH A TUPLE**

**You can loop through the tuple items by using a for loop:**

|  |  |
| --- | --- |
|  | example\_tuple = ("apple", "banana", "cherry") |
|  | for item in example\_tuple: |
|  | print(item) |

## **TUPLE LENGTH**

**To determine how many items a tuple has, use the len() function:**

|  |  |
| --- | --- |
|  | example\_tuple = ("apple", "banana", "cherry") |
|  | print(len(example\_tuple)) # Output: 3 |

## **THE TUPLE() CONSTRUCTOR**

**It is also possible to use the tuple() constructor to make a new tuple.**

|  |  |
| --- | --- |
|  | example\_tuple = tuple(("apple", "banana", "cherry")) # note the double round-brackets |
|  | print(example\_tuple) # Output: ("apple", "banana", "cherry") |

**SETS**

**A set is a collection which is *unordered* and *unindexed*.**

In Python sets are written with curly brackets.

|  |  |
| --- | --- |
|  | example\_set = {"apple", "banana", "cherry"} |
|  | print(example\_set) |

## ACCESSING ITEMS

**You cannot access items in a set by referring to an index, since sets are unordered the items have no index.**

But you can **loop through the set items** using a for loop, or ask if a specified value is present in a set, **by using the in keyword**.

|  |  |
| --- | --- |
|  | example\_set = {"apple", "banana", "cherry"} |
|  |  |
|  | for item in example\_set: |
|  | print(item) |

## ADDING ITEMS

To **add one item** to a set use the **add()** method.

To **add more than one item** to a set use the **update()** method.

|  |  |
| --- | --- |
|  | example\_set = {"apple", "banana", "cherry"} |
|  |  |
|  | example\_set.add("orange") # One Item |
|  |  |
|  | example\_set.update(["mango, grape"]) # Multiple Items |
|  |  |
|  | print(example\_set) |

## GETTING THE LENGTH OF A SET

**To determine how many items a set has, use the len() method.**

|  |  |
| --- | --- |
|  | example\_set = {"apple", "banana", "cherry"} |
|  |  |
|  | print(len(example\_set)) |

## REMOVING AN ELEMENT

To remove an item in a set you can use one of the following methods:

* remove() method - **Raises an error if the item to be removed doesn't exist**
* discard() method - **Does NOT raise an error if the item doesn't exist**

|  |  |
| --- | --- |
|  | example\_set = {"apple", "banana", "cherry"} |
|  |  |
|  | example\_set.remove("banana") |
|  |  |
|  | example\_set.discard("cherry") |

## THE SET() CONSTRUCTOR

**It is also possible to use the set() constructor to make a set.**

|  |  |
| --- | --- |
|  | example\_set = set(("apple", "banana", "cherry")) # note the double round-brackets |
|  | print(example\_set) |

**DICTIONARIES**

**A dictionary is a collection which is unordered, changeable and indexed.**

In Python dictionaries are written with curly brackets, and they have keys and values.

|  |  |
| --- | --- |
|  | example\_dict = { |
|  | "brand": "Ford", |
|  | "model": "Mustang", |
|  | "year": 1964 |
|  | } |
|  | print(example\_dict) |

## ACCESSING ITEMS

You can access the items of a dictionary by **referring to its key name**, inside square brackets:

|  |  |
| --- | --- |
|  | example\_dict = { |
|  | "brand": "Ford", |
|  | "model": "Mustang", |
|  | "year": 1964 |
|  | } |
|  |  |
|  | x = example\_dict["model"] |
|  | print(x) # Output: Mustang |

## ACCESSING ITEMS

**There is also a method called get() that will give you the same result:**

example\_dict = {

"brand": "Ford",

"model": "Mustang",

"year": 1964

}

x = example\_dict.get("model")

print(x) # Output: Mustang

## CHANGING VALUES

**You can update values using keys**

example\_dict = {

"brand": "Ford",

"model": "Mustang",

"year": 1964

}

example\_dict["year"] = 2018

## LOOP THROUGH A DICTIONARY

**You can loop through a dictionary by using a for loop.**

When looping through a dictionary, **the return value are the keys of the dictionary**, but there are methods to return the values as well.

example\_dict = {

"brand": "Ford",

"model": "Mustang",

"year": 1964

}

for x in example\_dict:

print(x)

# Output:

# brand

# model

# year

## LOOP THROUGH A DICTIONARY

**To print all values in the dictionary, you can do this:**

example\_dict = {

"brand": "Ford",

"model": "Mustang",

"year": 1964

}

for x in example\_dict:

print(example\_dict[x])

# Output:

# Ford

# Mustang

# 1964

**You can also use the values() function to return values of a dictionary:**

LOOP THROUGH A DICTIONARY

|  |  |
| --- | --- |
|  | example\_dict = { |
|  | "brand": "Ford", |
|  | "model": "Mustang", |
|  | "year": 1964 |
|  | } |
|  | for x in example\_dict.values(): |
|  | print(x) |

|  |  |
| --- | --- |
|  | # Output: |
|  | # Ford |
|  | # Mustang |
|  | # 1964 |

## LOOP THROUGH A DICTIONARY

**Loop through both keys and values, by using the items() function:**

example\_dict = {

"brand": "Ford",

"model": "Mustang",

"year": 1964

}

for x, y in example\_dict.items():

print(x, y)

# Output:

# brand Ford

# model Mustang

# year 1964

## DICTIONARY LENGTH

**To determine how many items (key-value pairs) a dictionary has, use the len() method.**

example\_dict = {

"brand": "Ford",

"model": "Mustang",

"year": 1964

}

print(len(example\_dict))

# Output:

# 3

## ADDING ITEMS

**Adding an item to the dictionary is done by using a new index key and assigning a value to it:**

example\_dict = {

"brand": "Ford",

"model": "Mustang",

"year": 1964

}

example\_dict["color"] = "red"

print(example\_dict)

# Output:

# {

# 'brand': 'Ford',

# 'model': 'Mustang',

# 'year': 1964,

# 'color': 'red'

# }

## REMOVING ITEMS

**The pop() method removes the item with the specified key name:**

example\_dict = {

"brand": "Ford",

"model": "Mustang",

"year": 1964

}

example\_dict.pop("model")

print(example\_dict)

# Output:

# {

# 'brand': 'Ford',

# 'year': 1964,

# }

## REMOVING ITEMS

**The popitem() method removes the last inserted item.**

(in versions before 3.7, a random item is removed instead):

example\_dict = {

"brand": "Ford",

"model": "Mustang",

"year": 1964

}

example\_dict.popitem()

print(example\_dict)

# Output:

# {

# 'brand': 'Ford',

# 'model': 'Mustang',

# }

## NESTED DICTIONARIES

A dictionary can also contain other dictionaries, these are called **nested dictionaries.**

my\_pets = {

"timmy": {

"type": "dog",

"color": "blonde"

},

"chippy": {

"type": "cat",

"color": "brown"

}

}

## THE DICT() CONSTRUCTOR

**It is also possible to use the dict() constructor to make a new dictionary:**

example\_dict = dict(brand="Ford", model="Mustang", year=1964)

print(example\_dict)

**There are 2 things to note when using dict():**

* The keys shouldn't have quotes around them.
* Values are assigned to keys using '=' rather than ':'.

# ****FUNCTIONS****

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

* You can pass data, known as parameters, into a function.
* A function can return data as a result.

## CREATING A FUNCTION

In Python a function is defined **using the def keyword:**

def my\_function():

print("Hello from a function")

## CALLING A FUNCTION

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

def my\_function():

print("Hello from a function")

my\_function() # Output: Hello from a function

## FUNCTION ARGUMENTS

**Information can be passed into functions as arguments.**

* Arguments are specified after the function name, inside the parentheses.
* You can add as many arguments as you want, just separate them with a comma.
* A **parameter** is the variable listed inside the parentheses in the function definition.
* An **argument** is the value that is sent to the function when it is called.

def my\_function(name):

print("Hello " + name)

my\_function("Abhiram") # Output: Hello Abhiram

## NUMBER OF ARGUMENTS

**By default, a function must be called with the correct number of arguments.**

Meaning that if your function expects 2 arguments, you **have** to call the function with 2 arguments.

def my\_function(fname, lname):

print(fname + " " + lname)

my\_function("Abhiram", "K")

## ARBITRARY ARGUMENTS

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

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

def my\_function(\*pets):

print("The youngest pet is " + pets[1])

my\_function("Timmy", "Chippy")

## DEFAULT PARAMETER VALUES

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

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

def my\_function(country = "India"):

print("I am from " + country)

my\_function("Sweden")

my\_function("Japan")

my\_function()

# Output:

# I am from Sweden

# I am from Japan

# I am from India

## RETURN VALUES

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

def my\_function(x):

return 5 \* x

print(my\_function(3))

print(my\_function(5))

print(my\_function(9))

# Output:

# 15

# 25

# 45

## RECURSION

**Python also accepts function recursion, which means a defined function can call itself.**

* Recursion is a common mathematical and programming concept. It means that a function calls itself. This has the benefit of meaning that you can loop through data to reach a result.
* The developer should be very careful with recursion as it can be quite easy to slip into writing a function which never terminates, or one that uses excess amounts of memory or processor power. However, when written correctly recursion can be a very efficient and mathematically-elegant approach to programming.

## RECURSION - EXAMPLE

# Calculate the sum of numbers

# using recursion

def sum(n):

if n <= 1:

return n

else:

return n + sum(n-1)

num = int(input("Enter a number: "))

print("The sum is: ", sum(num))

# Output:

# Enter a number: 10

# The sum is: 55

# ****ERROR HANDLING****

When an error occurs during execution, the Python Interpreter throws an **"Exception"**.

* Usually, when this happens, the execution is stopped and an error message is generated.
* These exceptions can be handled using the **'try'** and **'except'** statement:

|  |  |
| --- | --- |
|  | try: |
|  | print(x) |
|  | except: |
|  | print("An exception occurred") |

**Since the try block raises an error, the except block will be executed.**

Without the try block, the program will crash and raise an error:

|  |  |
| --- | --- |
|  | try: |
|  | print(x) |
|  | except: |
|  | print("An exception occurred") |

## MANY EXCEPTIONS

**You can define as many exception blocks as you want.**

e.g. if you want to execute a special block of code for a special kind of error:

|  |  |
| --- | --- |
|  | try: |
|  | print(x) |
|  | except NameError: |
|  | print("Variable x is not defined") |
|  | except: |
|  | print("Something else went wrong") |

## THE ELSE STATEMENT

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

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

## THE FINALLY STATEMENT

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

try:

print(x)

except:

print("Something went wrong")

finally:

print("The 'try except' is finished")

## THE WITH-AS BLOCK

In python the **with** keyword is used when working with unmanaged resources (like file streams or browser streams).

It allows you to ensure that a resource is "**cleaned up**" when the code that uses it finishes running, **even if exceptions are thrown**.

It provides '**syntactic sugar' for try/finally blocks**.

|  |  |
| --- | --- |
|  | with expression [as variable]: |
|  | statement\_1 |
|  | statement\_2 |

## THE WITH-AS BLOCK

"with-as" is very handy when you have two related operations which you’d like to execute as a pair, with a block of code in between.

The classic example is opening a file, manipulating the file, then closing it:

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

for line in file:

print(line)

## THE WITH-AS BLOCK

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

for line in file:

print(line)

 The above with statement will automatically close the file after the nested block of code.

 The advantage of using a with statement is that it is guaranteed to close the file no matter how the nested block exits.

 If an exception occurs before the end of the block, it will close the file before the exception is caught by an outer exception handler.

 If the nested block were to contain a return statement, or a continue or break statement, the with statement would automatically close the file in those cases, too.

# ****PIP****

## WHAT IS PIP?

PIP is a package manager for Python packages, or modules if you like.

## WHAT IS A PACKAGE?

A package contains all the files you need for a module.

Modules are Python code libraries you can include in your project.

## DOWNLOAD PACKAGES

Downloading a package is very easy.

Open the command line interface and tell PIP to download the package you want.

|  |  |
| --- | --- |
|  | $ pip install requests |

Downloading a package is very easy.

Open the command line interface and tell PIP to download the package you want.

|  |  |
| --- | --- |
|  | $ pip install requests |

Downloading a package is very easy.

Open the command line interface and tell PIP to download the package you want.

|  |  |
| --- | --- |
|  | $ pip install requests |

## USING PACKAGES - ALTERNATE SYNTAX

If you already know the functions you want to import from a package, you can use this syntax to import the functions directly.

|  |  |
| --- | --- |
|  | from requests import get |
|  |  |
|  | get('https://training-support.net') |

Once you import the function, you can simply call the function with it's name as shown.

## FINDING PACKAGES

You can search for packages on [https://pypi.org](https://pypi.org/)

## REMOVING PACKAGES

Use the **uninstall** command to remove a package:

|  |  |
| --- | --- |
|  | $ pip uninstall requests |

## LISTING PACKAGES

Use the **list** command to list all the packages installed on your system:

|  |  |
| --- | --- |
|  | $ pip list |
|  |  |
|  | Package Version |
|  | ----------------------- |
|  | requests 2.23 |

# ****OBJECTS****

**Objects are key to understanding object-oriented technology.**

* As an exercise, look around you right now. You'll find examples of many real world objects like your phone, your desk or even your pet dog!
* **Real-world objects share two characteristics: They all have state and behavior.**
  + For example, your phone has state (size, charge, color, etc) and it also has behavior  
    (ringing, charging, vibrating, etc).

**Software objects are conceptually similar to real-world objects: they too consist of state and related behavior.**

* An object stores its state in ***fields*** (variables) and exposes its behavior through ***methods*** (functions).
* Methods operate on an object's internal state and serve as the primary mechanism for object-to-object communication.

# ****CLASSES****

**In the real world, you'll often find many individual objects all of the same kind.**

* There may be millions of smartphones in existence, all from the same manufacturer and being the same model.
* Each smartphone was built from the same set of blueprints and therefore contains the same components.
* **In object-oriented terms, we say that your smartphone is an instance of the class of objects known as smartphones.**

**A class is the blueprint from which individual objects are created.**

## CREATING A CLASS

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

To create a class, use the keyword ***class***:

## CREATING A CLASS

|  |  |
| --- | --- |
|  | class MyClass: |
|  | 'This is an example class' |
|  | x = 5 |

To create a class, use the keyword ***class***:

You can add a docstring to your class like this.

* A **docstring** is simply a string that goes right under the class definition.
* It is used to document the purpose of the class and it appears in IDE popups related to that class as shown.

## CREATING AN OBJECT

|  |  |
| --- | --- |
|  | class MyClass: |
|  | 'This is an example class' |
|  | x = 5 |
|  |  |
|  | p1 = MyClass() |
|  | print(p1.x) # Output: 5 |

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

## THE \_\_INIT\_\_() FUNCTION

**The examples we just saw are classes and objects in their simplest form, and are not really useful in real life applications.**

* To understand the meaning of classes we have to understand the built-in **\_\_init\_\_()** function.
* All classes have a function called **\_\_init\_\_()**, which is always **executed when the class is being initiated.**
* Use the **\_\_init\_\_()** function to assign values to object properties, or other operations that are necessary to do when the object is being created:

## THE \_\_INIT\_\_() FUNCTION

**Let's take a look at this function by creating a "Person" class:**

|  |  |
| --- | --- |
|  | class Person: |
|  | 'This class represents a person' |
|  |  |
|  | def \_\_init\_\_(self, name, age): |
|  | self.name = name |
|  | self.age = age |
|  |  |
|  | p1 = Person("John", 36) |
|  |  |
|  | print(p1.name) # Output: "John" |
|  | print(p1.age) # Output: 36 |

## THE \_\_INIT\_\_() FUNCTION

**Let's take a look at this function by creating a "Person" class:**

|  |  |
| --- | --- |
|  | class Person: |
|  | 'This class represents a person' |
|  |  |
|  | def \_\_init\_\_(self, name, age): |
|  | self.name = name |
|  | self.age = age |
|  |  |
|  | p1 = Person("John", 36) |
|  |  |
|  | print(p1.name) # Output: "John" |
|  | print(p1.age) # Output: 36 |

|  |  |
| --- | --- |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

* In this example, we declare the \_\_init\_\_() function with 3 parameters:
  + self
  + name
  + age
*  The ***self*** parameter is a reference to the current instance of the class, and is used to access variables that belongs to the class.
*  The other two parameters are just data that you want to initialize your object with.

## ADDING METHODS

**Classes can also contain methods.**

Methods are just functions that belong to a specific class.

|  |  |
| --- | --- |
|  | class Person: |
|  | 'This class represents a person' |
|  |  |
|  | def \_\_init\_\_(self, name, age): |
|  | self.name = name |
|  | self.age = age |
|  |  |
|  | def myFunc(self): |
|  | print("Hello, my name is " + self.name) |
|  |  |
|  | p1 = Person("John", 36) |
|  |  |
|  | p1.myFunc() # Output: "Hello, my name is John" |

## MODIFY OBJECT PROPERTIES

You can modify properties on objects like this:

|  |  |
| --- | --- |
|  | class Person: |
|  | 'This class represents a person' |
|  |  |
|  | def \_\_init\_\_(self, name, age): |
|  | self.name = name |
|  | self.age = age |
|  |  |
|  | def myFunc(self): |
|  | print("Hello, my name is " + self.name) |
|  |  |
|  | p1 = Person("John", 36) |
|  |  |
|  | p1.name = "Harry" |
|  |  |
|  | p1.myFunc() # Output: "Hello, my name is Harry" |

## DELETING OBJECTS

You can delete objects by using the **del** keyword:

|  |  |
| --- | --- |
|  | class Person: |
|  | 'This class represents a person' |
|  |  |
|  | def \_\_init\_\_(self, name, age): |
|  | self.name = name |
|  | self.age = age |
|  |  |
|  | def myFunc(self): |
|  | print("Hello, my name is " + self.name) |
|  |  |
|  | p1 = Person("John", 36) |
|  |  |
|  | del p1 |

# ****DATE AND TIME****

**If we look at the output from the example code, we see that it contains:**

* Year
* Month
* Day
* Hour
* Minute
* Second
* Microsecond

|  |  |
| --- | --- |
|  | from datetime import datetime |
|  |  |
|  | x = datetime.now() |
|  | print(x) |
|  |  |
|  | # Output: 2020-05-01 12:11:17.902820 |

## DATE OUTPUT

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

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

|  |  |
| --- | --- |
|  | from datetime import datetime |
|  |  |
|  | x = datetime.now() |
|  |  |
|  | print(x.year) # Output: 2020 |
|  | print(x.strftime("%A")) # Output: Friday |

## CREATING DATE OBJECTS

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

The **datetime()** class requires three parameters to create a date: **year, month, day.** It also takes parameters for **time** and **timezone** (hour, minute, second, microsecond, tzone), but they are optional, and has a default value of 0, (Default is 'None' for timezone).

|  |  |
| --- | --- |
|  | from datetime import datetime |
|  |  |
|  | x = datetime(2020, 5, 1) |
|  |  |
|  | print(x) # Output: 2020-05-01 00:00:00 |

## THE STRFTIME() METHOD

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

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

from datetime import datetime

x = datetime(2020, 5, 1)

print(x.strftime("%B")) # Output: May

**Here is a reference to all the format codes you can use with strftime():**

## THE FORMAT CODES

|  |  |  |
| --- | --- | --- |
| **Directive** | **Description Example Output** | **Example Output** |
| %a | Weekday, short version Wed | Wed |
| %A | Weekday, full version Wednesday | Wednesday |
| %w | Weekday as a number 0-6, 0 is Sunday 3 | 3 |
| %d | Day of month 01-31 31 | 31 |
| %b | Month name, short version Dec | Dec |
| %B | Month name, full version December | December |
| %m | Month as a number 01-12 12 | 12 |

## THE FORMAT CODES

|  |  |  |
| --- | --- | --- |
| **Directive** | **Description Example Output** | **Example Output** |
| %y | Year, short version, without century 20 | 20 |
| %Y | Year, full version 2020 | 2020 |
| %H | Hour 00-23 17 | 17 |
| %I | Hour 00-12 05 | 05 |
| %p | AM/PM PM | PM |
| %M | Minute 00-59 41 | 41 |
| %S | Second 00-59 08 | 08 |

## THE FORMAT CODES

|  |  |  |
| --- | --- | --- |
| **Directive** | **Description Example Output** | **Example Output** |
| %j | Day number of year 001-366 365 | 365 |
| %U | Week number of year, Sunday as the first day of week, 00-53 52 | 52 |
| %W | Week number of year, Monday as the first day of week, 00-53 07 | 07 |
| %c | Local version of date and time Mon Dec 31 17:41:00 2018 | Mon Dec 31 17:41:00 2018 |
| %x | Local version of date 12/31/18 | 12/31/18 |
| %X | Local version of time 17:41:00 | 17:41:00 |

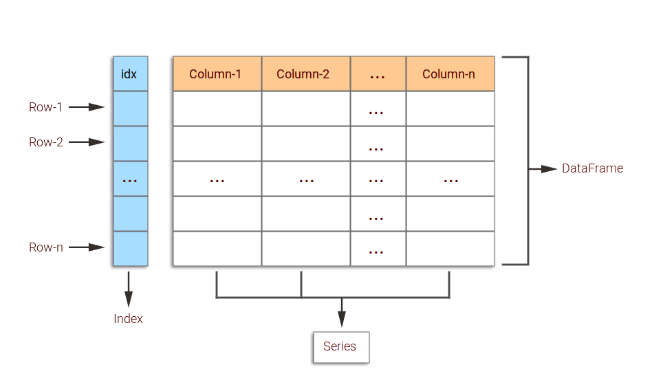
# PANDAS

**Pandas is a Python package providing fast, flexible, and expressive data structures designed to make working with 'relational' or 'labeled' data both easy and intuitive.**

* It aims to be the fundamental high-level building block for doing practical, real world data analysis in Python.
* **pandas** is well suited for many different kinds of data:
  + Tabular data with heterogeneously-typed columns, as in an SQL table or **Excel spreadsheet.**
  + Ordered and unordered (not necessarily fixed-frequency) time series data.
  + Arbitrary matrix data with row and column labels like a **CSV file.**
  + Any other form of observational / statistical data sets.

# PANDAS

## VISUAL DEPICTION



## INSTALLING PANDAS

$ pip install pandas xlrd openpyxl xlwt

**What are we installing?**

* pandas - The pandas package
* xlrd - A package used to extract data from excel files. Pandas depends on it.
* openpyxl - Used to read .xlsx files. xlrd depends on it.
* xlwt - Used to write .xls files. xlrd depends on it.

## PANDAS BASIC EXAMPLE

**Let's create a very basic dataframe to get you acquainted with pandas.**

Please follow along on your own computers.

|  |  |
| --- | --- |
|  | # import pandas |
|  | import pandas as pd |
|  |  |
|  | # Create a dictionary to hold our data |
|  | data = { |
|  | "X": [1, 2, 3, 4, 5], |
|  | "Y": [9, 8, 7, 6, 5], |
|  | "Z": [10, 12, 8, 4, 3] |
|  | } |
|  |  |
|  | # Create a new DataFrame using our dictionary |
|  | table = pd.DataFrame(data) |
|  |  |
|  | # Print the DataFrame |
|  | print(table) |

pandas\_demo.py

X Y Z

0 1 9 10

1 2 8 12

2 3 7 8

3 4 6 4

4 5 5 3

## WHAT DO WE UNDERSTAND SO FAR?

**From the example we can understand that a DataFrame in pandas is simply a tabular representation of data.**

We can use our Python Dictionaries to create new table structures. (DataFrames)

Using this knowledge, let's see how we can read and write CSV and Excel files using pandas.

**PANDAS - HANDLING CSV FILES**

First, let's create a sample csv file we can use.

|  |  |
| --- | --- |
|  | Name,Hire Date,Salary,Sick Days remaining |
|  | Graham Chapman,03/15/14,50000.00,10 |
|  | John Cleese,06/01/15,65000.00,8 |
|  | Eric Idle,05/12/14,45000.00,10 |
|  | Terry Jones,11/01/13,70000.00,3 |
|  | Terry Gilliam,08/12/14,48000.00,7 |
|  | Michael Palin,05/23/13,66000.00,8 |

employees.csv

Save the above file as **employees.csv**.

We will be using this file as an example to read and write data to.

## READING CSV DATA

|  |  |
| --- | --- |
|  | import pandas |
|  | dataframe = pandas.read\_csv('employees.csv') |
|  | print(dataframe) |

|  |  |
| --- | --- |
|  |  |
|  |  |
|  |  |

|  |  |
| --- | --- |
|  |  |
|  |  |
|  |  |

|  |  |
| --- | --- |
|  |  |
|  |  |
|  |  |

read\_csv.py

* We import the pandas package
* We read the csv file using pandas.read\_csv() and store it.
* We print the stored data
* Name Hire Date Salary Sick Days remaining
* 0 Graham Chapman 03/15/14 50000.0 10
* 1 John Cleese 06/01/15 65000.0 8
* 2 Eric Idle 05/12/14 45000.0 10
* 3 Terry Jones 11/01/13 70000.0 3
* 4 Terry Gilliam 08/12/14 48000.0 7
* 5 Michael Palin 05/23/13 66000.0 8

## READING CSV DATA - GETTING SPECIFIC DATA

Luckily for us, that's very simple.

By using the column name as a dictionary "Key" and the data as a list, we can use regular python syntax to get the specific data.

|  |  |
| --- | --- |
|  | import pandas |
|  | dataframe = pandas.read\_csv('employees.csv') |
|  | print(dataframe["Names"][2]) |

read\_csv.py

|  |  |
| --- | --- |
|  | Eric Idle |

## WRITING CSV DATA

**Writing data to a CSV is very simple as well!**

All we need to do is:

* Structure our data into a Python Dictionary
* Create a new DataFrame using that Dictionary as we saw in the pandas example
* Write that DataFrame to a csv file using **dataframe.to\_csv()**

## WRITING CSV DATA - EXAMPLE

**Let's take a look at a simple code example:**

|  |  |
| --- | --- |
|  | # import pandas |
|  | import pandas |
|  |  |
|  | # Structure our data as a dictionary |
|  | data = { |
|  | "Vehicle Type": ["Car", "Car", "Bike"], |
|  | "Manufacturer": ["Maruti", "Toyota", "Royal Enfield"], |
|  | "Model": ["Swift", "Corolla", "Thunderbird"] |
|  | } |
|  |  |
|  | # Create a new DataFrame using the data |
|  | dataframe = pandas.DataFrame(data) |
|  |  |
|  | # Write the data to a csv file |
|  | dataframe.to\_csv("vehicles.csv") |

# ****PANDAS - HANDLING EXCEL FILES****

## WRITING EXCEL DATA

**Writing to an Excel file:**

# Import pandas, ExcelFile and ExcelWriter

import pandas

from pandas import ExcelFile

from pandas import ExcelWriter

# Structure our data as a dictionary

data = {

"Vehicle Type": ["Car", "Car", "Bike"],

"Manufacturer": ["Maruti", "Toyota", "Royal Enfield"],

"Model": ["Swift", "Corolla", "Thunderbird"]

}

# Create a new DataFrame using the data

dataframe = pandas.DataFrame(data)

# Create an ExcelWriter object

writer = ExcelWriter("vehicles.xlsx")

# Write the DataFrame to the Excel file

dataframe.to\_excel(writer, "Sheet1", index=False)

# Save the file

writer.save()

**HANDLING XML FILES**

First, let's create a sample xml file we can use.

Create an excel file using the data provided and save it as **properties.xml**.

We will be using this file as an example to read and write data to.

|  |  |
| --- | --- |
|  | <?xml version="1.0" encoding="UTF-8"?> |
|  | <root> |
|  | <bathrooms> |
|  | <n35237 type="number">1.0</n35237> |
|  | <n32238 type="number">3.0</n32238> |
|  | <n44699 type="number">nan</n44699> |
|  | </bathrooms> |
|  | <price> |
|  | <n35237 type="number">7020000.0</n35237> |
|  | <n32238 type="number">10000000.0</n32238> |
|  | <n44699 type="number">4128000.0</n44699> |
|  | </price> |
|  | <property\_id> |
|  | <n35237 type="number">35237.0</n35237> |
|  | <n32238 type="number">32238.0</n32238> |
|  | <n44699 type="number">44699.0</n44699> |
|  | </property\_id> |
|  | </root> |

Reading with **xml.etree.ElementTree and Pandas**

import xml.etree.ElementTree as ET

import pandas as pd

xml\_data = open('properties.xml', 'r').read() # Read file

root = ET.XML(xml\_data) # Parse XML

# print(xml\_data) # Uncomment this line to see the XML get printed

data = [] # Create a list for data

cols = [] # Create a list for the columns

# Iterate through the data and write them into the data/cols lists

for i, child in enumerate(root):

data.append([subchild.text for subchild in child])

cols.append(child.tag)

df = pd.DataFrame(data).T # Write in DataFrame and transpose it

df.columns = cols # Update column names

print(df)

Reading with **xml.etree.ElementTree and Pandas**

bathrooms price property\_id

0 1.0 7020000.0 35237.0

1 3.0 10000000.0 32238.0

2 nan 4128000.0 44699.0

## OUR GOAL WILL BE TO RECREATE THIS FILE BELOW:

|  |  |
| --- | --- |
|  | <root> |
|  | <A> |
|  | <X>1.3</X> |
|  | <Y>2.6</Y> |
|  | <Z>2.1</Z> |
|  | </A> |
|  | <B> |
|  | <X>1.4</X> |
|  | <Y>1.4</Y> |
|  | <Z>5.6</Z> |
|  | </B> |
|  | <C> |
|  | <X>5.2</X> |
|  | <Y>4.6</Y> |
|  | <Z>4.6</Z> |
|  | </C> |
|  | </root> |

The following code will produce a file named '**coordinates.xml**' in the current directory.

|  |  |
| --- | --- |
|  | import pandas as pd |
|  |  |
|  | df = pd.DataFrame([[1.3, 1.4, 5.2], |
|  | [2.6, 1.4, 4.6], |
|  | [2.1, 5.6, 4.6]], |
|  | columns=['A', 'B', 'C'], |
|  | index=['X', 'Y', 'Z']) |
|  |  |
|  | xml\_data = ['<root>'] |
|  | for column in df.columns: |
|  | xml\_data.append('<{}>'.format(column)) # Opening element tag |
|  | for field in df.index: |
|  | # writing sub-elements |
|  | xml\_data.append('<{0}>{1}</{0}>'.format(field, df[column][field])) |
|  | xml\_data.append('</{}>'.format(column)) # Closing element tag |
|  | xml\_data.append('</root>') |
|  |  |
|  | with open('coordinates.xml', 'w') as f: # Writing in XML file |
|  | for line in xml\_data: |
|  | f.write(line) |

**Python comes out of the box with json support.**

By importing the json module, we can read json files pretty effortlessly.

Assume you have the following json file called "**currency.json**"

|  |  |
| --- | --- |
|  | { |
|  | "usd": 1, |
|  | "eur": 1.2, |
|  | "gbp": 1.2 |
|  | } |

**You can read this file as follows:**

|  |  |
| --- | --- |
|  | import json |
|  |  |
|  | # read file |
|  | with open('currency.json', 'r') as myfile: |
|  | data=myfile.read() |
|  |  |
|  | # parse file |
|  | obj = json.loads(data) |
|  |  |
|  | # show values |
|  | print("usd: " + str(obj['usd'])) # usd: 1 |
|  | print("eur: " + str(obj['eur'])) # eur: 1.2 |
|  | print("gbp: " + str(obj['gbp'])) # gbp: 1.2 |

# ****SCRAPING WEB PAGES****

**The process of automatically gathering data from a webpage is called "scraping".**

Scraping data using python is very very easy thanks to a library called "**BeautifulSoup**".

Let's take a look at an example on the next slide.

On this webpage, [https://training-support.net/selenium/tables](https://www.training-support.net/selenium/tables), there are 2 tables.

Let's assume we want to get all the data from the 2nd table on the page.

Scraping the table is possible with the following code:

|  |  |
| --- | --- |
|  | # 1. Import the necessary libraries |
|  | import requests |
|  | from bs4 import BeautifulSoup |
|  |  |
|  | # 2. Send a get() request and fetch the webpage contents |
|  | response = requests.get("https://www.training-support.net/selenium/tables") |
|  | webpage\_content = response.content |
|  |  |
|  | # 3. Create a BeautifulSoup Object |
|  | soup = BeautifulSoup(webpage\_content, "html.parser") |
|  |  |
|  | # 4. Target the specific table we want |
|  | table = soup.find('table', {'id': 'sortableTable'}) |
|  |  |
|  | # 5. Scrape the data in the table |
|  | for tr in table.find\_all('tr'): # For all the rows in the table |
|  | cells = tr.find\_all('td') # find all the cells |
|  | row = [i.text for i in cells] # iterate through the cells and store the data in a list |
|  | print(row) # print the rows |

# ****SCHEDULING TASKS USING PYTHON-CRONTAB****

**cron is a time-based job scheduler in Unix-like computer operating systems.**

* Users that set up and maintain software environments use cron to schedule jobs to run periodically at fixed times, dates, or intervals.
* A crontab file contains instructions to the cron daemon of the general form: "run this command at this time on this date".
* Each user can have their own crontab.

# ****SCHEDULING TASKS USING PYTHON-CRONTAB****

**python-crontab**is a python library that gives us the ability to interact with crontab.

It allows us to manipulate crontabs on a per-user basis.

**Let's take a look at an example**

|  |
| --- |
| from crontab import CronTab |
|  | cron = CronTab(user='root') |
|  | job = cron.new(command='echo hello\_world') |
|  | job.minute.every(1) |
|  | cron.write() |

You can see more examples on the python-crontab pypi page here: <https://pypi.org/project/python-crontab/>

# ****INTRODUCTION TO PYTEST****

**pytest is a testing framework based on python.**

* **Testing frameworks** are an essential part of any successful automated testing process.
* They can reduce maintenance costs and testing efforts and will provide a higher return on investment (ROI) for QA teams looking to optimize their agile processes.
* The pytest framework makes it easy to write small tests, yet scales to support complex functional testing for applications and libraries.

# ****ADVANTAGES OF PYTEST****

**Some of the advantages of pytest are**

* Very easy to start with because of its simple and easy syntax.
* Can run tests in parallel.
* Can run a specific test or a subset of tests
* Automatically detect tests
* Skip tests
* Open source

# ****INSTALLATION****

**To use pytest, you will need to install the following packages using pip:**

$ pip install pytest pytest-xdist pytest-html

Here's what the packages do:

* **pytest**: The actual testing framework package.
* **pytest-xdist**: Allows us to run our tests in parallel.
* **pytest-html**: Allows us to generate a HTML report of our tests.

# ****VERIFY INSTALLATION****

**To check if pytest was installed correctly, you can type this command into your terminal:**

$ pytest --version

This is pytest version 5.4.1, imported from /usr/local/lib/python3.8/site-packages/pytest/\_\_init\_\_.py

setuptools registered plugins:

pytest-xdist-1.32.0 at /usr/local/lib/python3.8/site-packages/xdist/plugin.py

pytest-xdist-1.32.0 at /usr/local/lib/python3.8/site-packages/xdist/looponfail.py

pytest-html-2.1.1 at /usr/local/lib/python3.8/site-packages/pytest\_html/plugin.py

pytest-metadata-1.9.0 at /usr/local/lib/python3.8/site-packages/pytest\_metadata/plugin.py

pytest-forked-1.1.3 at /usr/local/lib/python3.8/site-packages/pytest\_forked/\_\_init\_\_.py

You should see an output similar to the one shown above.

# ****YOUR FIRST TEST****

**Before you proceed:**

* Make a new folder called 'tests'.
* Make sure your python filename follows the following naming scheme:
  + test\_<filename>.py
  + <filename>\_test.py
* pytest requires that "test\_" or "\_test" to be part of the name so that it can autodetect test files.
* **YOUR FIRST TEST**

|  |  |
| --- | --- |
|  | import pytest |
|  | import math |
|  |  |
|  | def test\_sqrt(): |
|  | num = 25 |
|  | assert math.sqrt(num) == 5 |
|  |  |
|  | def testsquare(): |
|  | num = 7 |
|  | assert num\*num == 40 |
|  |  |
|  | def tesequality(): |
|  | assert 10 == 11 |

 First, we import pytest

 Then we define test methods:

* Test method names need to be of the format "test\*".
* They always need to resolve to a boolean value. This is usually accomplished with assertions.
* **You can run your test file using the command 'pytest' inside your project folder.**
* **Note** − pytest command will execute all the files of format **test\_\*** or **\*\_test** in the current directory and subdirectories.

$ pytest

====================================== test session starts ======================================

platform linux -- Python 3.8.2, pytest-5.4.1, py-1.8.1, pluggy-0.13.1

rootdir: /home/abhiram/Development/Python\_programs/PyTest

plugins: xdist-1.32.0, html-2.1.1, metadata-1.9.0, forked-1.1.3

collected 2 items

test\_learning.py .F [100%]

=========================================== FAILURES ============================================

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ testsquare \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

def testsquare():

num = 7

> assert num\*num == 40

E assert (7 \* 7) == 40

test\_learning.py:9: AssertionError

==================================== short test summary info ====================================

FAILED test\_learning.py::testsquare - assert (7 \* 7) == 40

================================== 1 failed, 1 passed in 0.05s ==================================

**Let's take a closer look at the important parts of the output:**

* See the first line of the result. It displays the file name and the results. 'F' represents a test failure and dot '.' represents a test success.
* So in this case, since it has ".F", we can tell that one test passed and one test failed.

test\_learning.py .F [100%]

**Let's take a closer look at the important parts of the output:**

* Below that, we can see the details of the failed tests. It will show the method, and the statement in that method where the test has failed.
* In our case, we can see that **'num\*num'** was compared for equality against 40, which is not correct.  
  You can the same comparison with the values substituted on line 9.

|  |
| --- |
| =========================================== FAILURES ============================================ |
|  | \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ testsquare \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ |
|  |  |
|  | def testsquare(): |
|  | num = 7 |
|  | > assert num\*num == 40 |
|  | E assert (7 \* 7) == 40 |

**Let's take a closer look at the important parts of the output:**

* Then finally, we see a test summary which shows us how many tests failed and how many tests passed.
* Do note that **'tesequality()'** is not executed because pytest will not consider it as a test since its name is not of the format **test\***.
* ==================================== short test summary info ====================================
* FAILED test\_learning.py::testsquare - assert (7 \* 7) == 40
* ================================== 1 failed, 1 passed in 0.05s ==================================
* **You can also run your test file using the command 'pytest -v' to get a more verbose output.**
* **Note** − pytest command will execute all the files of format **test\_\*** or **\*\_test** in the current directory and subdirectories.

pytest -v

====================================== test session starts ======================================

platform linux -- Python 3.8.2, pytest-5.4.1, py-1.8.1, pluggy-0.13.1 -- /usr/bin/python3

cachedir: .pytest\_cache

metadata: {'Python': '3.8.2', 'Platform': 'Linux-5.6.4-300.fc32.x86\_64-x86\_64-with-glibc2.2.5',

'Packages': {'pytest': '5.4.1', 'py': '1.8.1', 'pluggy': '0.13.1'},

'Plugins': {'xdist': '1.32.0', 'html': '2.1.1', 'metadata': '1.9.0', 'forked': '1.1.3'}}

rootdir: /home/abhiram/Development/Python\_programs/PyTest

plugins: xdist-1.32.0, html-2.1.1, metadata-1.9.0, forked-1.1.3

collected 2 items

test\_learning.py::test\_sqrt PASSED [ 50%]

test\_learning.py::testsquare FAILED [100%]

=========================================== FAILURES ============================================

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ testsquare \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

def testsquare():

num = 7

> assert num\*num == 40

E assert 49 == 40

E +49

E -40

test\_learning.py:9: AssertionError

==================================== short test summary info ====================================

FAILED test\_learning.py::testsquare - assert 49 == 40

================================== 1 failed, 1 passed in 0.05s ==================================

# ****EXECUTING FILES****

To execute the tests from a specific file, use the following syntax:

$ pytest <filename>

You can use this syntax to execute test files that don't follow the 'test\_' or '\_test' naming scheme as well.

# ****EXECUTING A SUBSET OF THE TEST SUITE****

**When working on real world projects, we typically work with multiple test files and each test file will have a number of tests in it.**

* These test files are collectively called the "Test Suite".
* Tests in the test suite generally cover various modules and functionalities.
* What if we only wanted to run a specific set of tests? How do we go about it?

**Pytest provides two ways to run the subset of the test suite.**

* Select tests to run based on substring matching of test names.
* Select test groups to run based on the markers applied.

# SUBSTRING MATCHING OF TEST NAMES

**To execute tests containing a specific string in their name we can use the following command:**

$ pytest -k <substring>

Let's take a look at an example so we can better understand

# SUBSTRING MATCHING OF TEST NAMES

Consider the following test file:

import pytest

def test\_greater():

x = 2

y = 5

assert y > x

def test\_greater\_equal():

x = 6

y = 6

assert y >= x

def test\_lesser():

x = 2

y = 5

assert x < y

**In this test file, we have 3 test methods:**

* test\_greater()
* test\_greater\_equal()
* test\_lesser()

**Using the command we saw in the previous slide, we can execute the first 2 methods like this:**

$ pytest -k great

We can see in the summary that "2 passed, 1 deselected".

This is because only the tests with \*great\* in their names were executed.

The test\_lesser() method was not executed because it did not match our criteria.

$ pytest -k great

====================================== test session starts ======================================

platform linux -- Python 3.8.2, pytest-5.4.1, py-1.8.1, pluggy-0.13.1

rootdir: /home/abhiram/Development/Python\_programs/PyTest

plugins: xdist-1.32.0, html-2.1.1, metadata-1.9.0, forked-1.1.3

collected 3 items / 1 deselected / 2 selected

test\_compares.py .. [100%]

================================ 2 passed, 1 deselected in 0.01s ================================

# GROUPING TESTS

**Pytest allows us to use markers on test functions.**

* Markers are used to set various features/attributes to test functions.
* Pytest provides many inbuilt markers such as xfail, skip and parametrize.
* Apart from that, users can create their own marker names.

# GROUPING TESTS

**Markers are applied on the tests using the syntax given below:**

@pytest.mark.<markername>

**To run the marked tests, we can use the following command:**

$ pytest -m <markername>

Lets revisit our **test\_compare.py** file:

|  |  |
| --- | --- |
|  | import pytest |
|  |  |
|  | def test\_greater(): |
|  | x = 2 |
|  | y = 5 |
|  | assert y > x |
|  |  |
|  | def test\_greater\_equal(): |
|  | x = 6 |
|  | y = 6 |
|  | assert y >= x |
|  |  |
|  | def test\_lesser(): |
|  | x = 2 |
|  | y = 5 |
|  | assert x < y |

We can add markers like this:

|  |  |
| --- | --- |
|  | import pytest |
|  |  |
|  | @pytest.mark.great |
|  | def test\_greater(): |
|  | x = 2 |
|  | y = 5 |
|  | assert y > x |
|  |  |
|  | @pytest.mark.great |
|  | def test\_greater\_equal(): |
|  | x = 6 |
|  | y = 6 |
|  | assert y >= x |
|  |  |
|  | @pytest.mark.others |
|  | def test\_lesser(): |
|  | x = 2 |
|  | y = 5 |
|  | assert x < y |

Now to run the tests marked as **others**, run the following command:

$ pytest -m others

$ pytest -m others

====================================== test session starts ======================================

platform linux -- Python 3.8.2, pytest-5.4.1, py-1.8.1, pluggy-0.13.1

rootdir: /home/abhiram/Development/Python\_programs/PyTest

plugins: xdist-1.32.0, html-2.1.1, metadata-1.9.0, forked-1.1.3

collected 3 items / 2 deselected / 1 selected

test\_compares.py . [100%]

======================================= warnings summary ========================================

test\_compares.py:3

/home/abhiram/Development/Python\_programs/PyTest/test\_compares.py:3: PytestUnknownMarkWarning:

Unknown pytest.mark.great - is this a typo? You can register custom marks to avoid this warning

@pytest.mark.great

test\_compares.py:9

/home/abhiram/Development/Python\_programs/PyTest/test\_compares.py:9: PytestUnknownMarkWarning:

Unknown pytest.mark.great - is this a typo? You can register custom marks to avoid this warning

@pytest.mark.great

test\_compares.py:15

/home/abhiram/Development/Python\_programs/PyTest/test\_compares.py:15: PytestUnknownMarkWarning:

Unknown pytest.mark.others - is this a typo? You can register custom marks to avoid this warning

@pytest.mark.others

-- Docs: https://docs.pytest.org/en/latest/warnings.html

========================== 1 passed, 2 deselected, 3 warnings in 0.01s ==========================

If we look at the summary, we can see that the test did pass, but we see a bunch of warnings.

**This is because we need to register our custom marker first before using it.**

======================================= warnings summary ========================================

test\_compares.py:3

/home/abhiram/Development/Python\_programs/PyTest/test\_compares.py:3: PytestUnknownMarkWarning:

Unknown pytest.mark.great - is this a typo? You can register custom marks to avoid this warning

@pytest.mark.great

test\_compares.py:9

/home/abhiram/Development/Python\_programs/PyTest/test\_compares.py:9: PytestUnknownMarkWarning:

Unknown pytest.mark.great - is this a typo? You can register custom marks to avoid this warning

@pytest.mark.great

test\_compares.py:15

/home/abhiram/Development/Python\_programs/PyTest/test\_compares.py:15: PytestUnknownMarkWarning:

Unknown pytest.mark.others - is this a typo? You can register custom marks to avoid this warning

@pytest.mark.others

-- Docs: https://docs.pytest.org/en/latest/warnings.html

========================== 1 passed, 2 deselected, 3 warnings in 0.01s ==========================

## REGISTERING CUSTOM MARKERS

To register custom markers, create a file called **'pytest.ini'** inside your tests folder and add the following code into it:

|  |  |
| --- | --- |
|  | [pytest] |
|  | markers = |
|  | great: mark a test as part of the great group. |
|  | others: mark a test as part of the others group. |

This will register the markers for use in pytest.

## REGISTERING CUSTOM MARKERS

To register custom markers, create a file called **'pytest.ini'** inside your tests folder and add the following code into it:

|  |  |
| --- | --- |
|  | [pytest] |
|  | markers = |
|  | great: mark a test as part of the great group. |
|  | others: mark a test as part of the others group. |
|  | <name>: <description> |

You can register more markers by simply appending them to the list as shown.

Let's run the tests again and check the output.

## REGISTERING CUSTOM MARKERS

$ pytest -m others

====================================== test session starts ======================================

platform linux -- Python 3.8.2, pytest-5.4.1, py-1.8.1, pluggy-0.13.1

rootdir: /home/abhiram/Development/Python\_programs/PyTest, inifile: pytest.ini

plugins: xdist-1.32.0, html-2.1.1, metadata-1.9.0, forked-1.1.3

collected 3 items / 2 deselected / 1 selected

test\_compares.py . [100%]

================================ 1 passed, 2 deselected in 0.01s ================================

Now we can see that our warnings disappeared.

# ****FIXTURES****

**Fixtures are functions which will run before each test function to which it is applied.**

Fixtures are used to feed some data to the tests such as database connections, URLs to test, input data, and even selenium driver initializations.

Therefore, instead of running the same code for every test, we can attach fixture function to the tests and it will run and return the data to the test before executing each test.

A function is marked as a **Fixture** by adding the following line of code before function declaration:

@pytest.fixture

Let's take a look at an example

Create a file **test\_divisibility.py** and add the below code to it

|  |  |
| --- | --- |
|  | import pytest |
|  |  |
|  | @pytest.fixture |
|  | def input\_value(): |
|  | input = 39 |
|  | return input |
|  |  |
|  | def test\_divisible\_by\_3(input\_value): |
|  | assert input\_value % 3 == 0 |
|  |  |
|  | def test\_divisible\_by\_6(input\_value): |
|  | assert input\_value % 6 == 0 |

 Here, we have a fixture function named **input\_value**, which supplies the input to the tests.

 To access the fixture function, the tests have to mention the fixture name as input parameter.

 While the tests is getting executed, Pytest will see the fixture name as input parameter.

 It then executes the fixture function and the returned value is stored to the input\_value parameter, which can be used by the test.

$ pytest -k divisible

test\_divisibility.py .F [100%]

=========================================== FAILURES ============================================

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ test\_divisible\_by\_6 \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

input\_value = 39

def test\_divisible\_by\_6(input\_value):

> assert input\_value % 6 == 0

E assert (39 % 6) == 0

test\_divisibility.py:12: AssertionError

==================================== short test summary info ====================================

FAILED test\_divisibility.py::test\_divisible\_by\_6 - assert (39 % 6) == 0

=========================== 1 failed, 1 passed, 4 deselected in 0.11s ===========================

## DEFINING SCOPE

A **fixture’s scope** defines how manytimes a fixture will be invoked during a test. Fixtures can have 4 Scopes:

1. **Module:**If the Module scope is defined, the fixture will be created/invoked only once per module.
2. **Class:**With Class scope, one fixture will be created per class object.
3. **Session:**With the Session scope, the fixture will be created only once for entire test session. This is mainly used to manage Webdriver sessions.
4. **Function:**This is the default value for fixture scope and the fixture is executed/run once per test session.

## DEFINING SCOPE

You can set the scope by adding a "scope" attribute to your fixture definition:

|  |  |
| --- | --- |
|  | @pytest.fixture(scope="module") |
|  | def example\_fixture(): |
|  | statements |
|  |  |
|  | @pytest.fixture(scope="class") |
|  | def example\_fixture(): |
|  | statements |
|  |  |
|  | @pytest.fixture(scope="session") |
|  | def example\_fixture(): |
|  | statements |
|  |  |
|  | @pytest.fixture(scope="function") |
|  | def example\_fixture(): |
|  | statements |

**CONFTEST.PY**

We can define the fixture functions in this file to make them accessible across multiple test files.

Create a new file '**conftest.py**' and add the below code into it:

|  |  |
| --- | --- |
|  | import pytest |
|  |  |
|  | @pytest.fixture |
|  | def input\_value(): |
|  | input = 39 |
|  | return input |

Edit the '**test\_divisibility.py**' from the previous example and remove the fixture definition from it:

|  |  |
| --- | --- |
|  | import pytest |
|  |  |
|  | def test\_divisible\_by\_3(input\_value): |
|  | assert input\_value % 3 == 0 |
|  |  |
|  | def test\_divisible\_by\_6(input\_value): |
|  | assert input\_value % 6 == 0 |

Now we have these two methods making use of a fixture defined in your conftest.py. Let's execute it!

$ pytest -k divisible

test\_divisibility.py .F [100%]

=========================================== FAILURES ============================================

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ test\_divisible\_by\_6 \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

input\_value = 39

def test\_divisible\_by\_6(input\_value):

> assert input\_value % 6 == 0

E assert (39 % 6) == 0

test\_divisibility.py:7: AssertionError

==================================== short test summary info ====================================

FAILED test\_divisibility.py::test\_divisible\_by\_6 - assert (39 % 6) == 0

=========================== 1 failed, 1 passed, 4 deselected in 0.10s ===========================

## SCOPE OF THE FILE

**conftest.py** is applied at a "Directory" or "Module" level.

Here's an example:

tests/

├── conftest.py # Applies to all tests

└── A/

├── conftest.py # Applies only to tests in this module/folder (i.e. A).

├── test\_a.py

└── B/

├── conftest.py # Applies only to tests in this module/folder (i.e. B).

├── test\_b.py

# ****TEST PARAMETERIZATION****

**Parameterizing of a test is done to run the test against multiple sets of inputs.**

We can do this by using the following marker:

@pytest.mark.parametrize

Let's take a look at a code example!

# ****TEST PARAMETERIZATION****

Create a new file '**test\_multiplication.py**' and put this code into it:

import pytest

@pytest.mark.parametrize("num, output", [(1, 11), (2, 22), (3, 33)])

def test\_multiplication\_11(num, output):

assert 11 \* num == output

**Here the test multiplies an input with 11 and compares the result with the expected output.**

The test has 3 **sets of inputs**, each has 2 values:

One is the number to be multiplied with 11 and the other is the expected result.

# ****TEST PARAMETERIZATION****

$ pytest -k multiplication

====================================== test session starts ======================================

platform linux -- Python 3.8.2, pytest-5.4.1, py-1.8.1, pluggy-0.13.1

rootdir: /home/abhiram/Development/Python\_programs/PyTest, inifile: pytest.ini

plugins: xdist-1.32.0, html-2.1.1, metadata-1.9.0, forked-1.1.3

collected 9 items / 6 deselected / 3 selected

test\_multiplication.py ... [100%]

================================ 3 passed, 6 deselected in 0.07s ================================

When we execute it, we can see that 3 tests passed.

Since we had 3 sets of parameters, we know that our parameterization worked!

# ****XFAIL AND TEST SKIPPING****

**While designing tests, you will sometimes run into the following scenarios:**

* A test might not be relevant for a while due to some reasons.
* A new feature might be under development and we already added a test for that feature.
* In these situations, we have the option to **xfail** the test or **skip** the tests.
* **Pytest will execute the xfailed test, but it will not be considered as a failed or passed test.**
* Details of these tests will not be printed even if the test fails (remember pytest usually prints the failed test details).
* We can **xfail** tests using the following marker:
* @pytest.mark.xfail
* **Skipping a test means that the test will not be executed.**
* We can skip tests using the following marker:
* @pytest.mark.skip
* Later, when the test becomes relevant we can remove the markers.
* Let's edit our **'test\_compares.py'** file that we have to include the xfail and skip markers:

|  |  |
| --- | --- |
|  | import pytest |
|  |  |
|  | @pytest.mark.great |
|  | def test\_greater(): |
|  | x = 2 |
|  | y = 5 |
|  | assert y > x |
|  |  |
|  | @pytest.mark.great |
|  | def test\_greater\_equal(): |
|  | x = 6 |
|  | y = 6 |
|  | assert y >= x |
|  |  |
|  | @pytest.mark.others |
|  | def test\_lesser(): |
|  | x = 2 |
|  | y = 5 |
|  | assert x < y |

$ pytest test\_compares.py

====================================== test session starts ======================================

platform linux -- Python 3.8.2, pytest-5.4.1, py-1.8.1, pluggy-0.13.1

rootdir: /home/abhiram/Development/Python\_programs/PyTest, inifile: pytest.ini

plugins: xdist-1.32.0, html-2.1.1, metadata-1.9.0, forked-1.1.3

collected 3 items

test\_compares.py XXs [100%]

================================= 1 skipped, 2 xpassed in 0.02s =================================

**In the output, we can now see that '1 skipped, 2 xpassed'.**

# ****STOP TEST SUITE AFTER X TEST FAILURES****

**In a real scenario, once a new version of the code is ready to deploy, it is first deployed into pre-prod/ staging environment. Then a test suite runs on it.**

* The code is qualified for deploying to production only if the test suite passes. If there is test failure, whether it is one or many, the code is not production ready.
* Therefore, what if we want to stop the execution of test suite soon after n number of test fails.

**This can be done in pytest by executing the tests using the maxfail flag.**

**The syntax to stop the execution of test suite soon after n number of test fails is as follows:**

$ pytest --maxfail = <num>

Here, <num> is the number of failures to wait for before stopping test suite execution.

# ****RUN TESTS IN PARALLEL****

**By default, pytest runs tests in sequential order.**

In a real scenario, a test suite will have a number of test files and each file will have a bunch of tests. This will lead to a large execution time.

To overcome this, pytest provides us with an option to run tests in parallel.

**We can run tests by executing the tests using the 'n' flag.**

$ pytest -n 3

'-n <num>' runs the tests by using multiple workers (threads), here it is 3.

We will not be having much time difference when there is only a few tests to run. However, it matters when the test suite is large.

**TEST RESULTS IN XML**

**We can generate the details of the test execution in an xml file.**

This xml file is mainly useful in cases where we have a dashboard that projects the test results. In such cases, the xml can be parsed to get the details of the execution.

**Lets execute the tests from test\_multiplcation.py and generate the xml by running the following command:**

|  |  |
| --- | --- |
|  | $ pytest test\_multiplication.py --junitxml="result.xml" |

## RESULT.XML

<?xml version="1.0" encoding="utf-8"?>

<testsuites>

<testsuite errors="0" failures="0" hostname="my-pc" name="pytest" skipped="0" tests="3" time="0.020" timestamp="2020-05-08T04:52:25.620140">

<testcase classname="test\_multiplication" file="test\_multiplication.py" line="2" name="test\_multiplication\_11[1-11]" time="0.001"></testcase>

<testcase classname="test\_multiplication" file="test\_multiplication.py" line="2" name="test\_multiplication\_11[2-22]" time="0.000"></testcase>

<testcase classname="test\_multiplication" file="test\_multiplication.py" line="2" name="test\_multiplication\_11[3-33]" time="0.001"></testcase>

</testsuite>

</testsuites>

**TEST RESULTS IN HTML**

**We can generate the details of the test execution in a html file.**

This html report can be zipped and shared with anyone so that it can be reviewed at a glance in a browser.

**Lets execute the tests from test\_multiplcation.py and generate the html report by running the following command:**

|  |  |
| --- | --- |
|  | $ pytest test\_multiplication.py --html="report.html" |

## RESULT.HTML

