UNIT4 / WEEK4: CONTROL FLOW

Python has two primitive loop commands:

- while loops
- for loops

The while Loop

- With the while loop we can execute a set of statements as long as a condition is true.
- The while loop requires relevant variables to be ready, in this example we need to define an indexing variable, i, which we set to 1.

```
# Print i as long as i is less than 6:

i = 1
    while i < 6:
        print(i)
        i += 1
```

The break Statement

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

```
# Exit the loop when i is 3

i = 1
while i < 6:
  print(i)
  if i == 3:
    break
  i += 1
```

The continue Statement

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

Continue to the next iteration if i is 3:

```
i = 0
while i < 6:
  i += 1
  if i == 3:
    continue
  print(i)
```

The else Statement

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

#Print a message once the condition is false:

```
i = 1
while i < 6:
  print(i)
  i += 1
else:
  print("i is no longer less than 6")
```

The For Loops

- 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.
- The for loop does not require an indexing variable to set beforehand.

Print each fruit in a fruit list

```
fruits = ["apple", "banana", "cherry"]
for x in fruits:
  print(x)
```

Looping Through a String

- Even strings are iterable objects, they contain a sequence of characters:

```
# Loop through the letters in the word "banana":

for x in "banana":
  print(x)
```

The break Statement

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

```
# Exit the loop when x is "banana"

fruits = ["apple", "banana", "cherry"]
for x in fruits:
  print(x)
  if x == "banana":
    break
```

output: apple , banana

```
#Exit the loop when x is "banana", but this time the break comes before the print:

fruits = ["apple", "banana", "cherry"]
for x in fruits:
  if x == "banana":
    break
  print(x)
```

output: apple

The continue Statement

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

```
#Do not print banana

fruits = ["apple", "banana", "cherry"]
for x in fruits:
  if x == "banana":
    continue
  print(x)
```

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.
- Note that range(6) is not the values of 0 to 6, but the values 0 to 5.

```
#Using the range() function:

for x in range(6):
  print(x)
```

output:0 1 2 3 4 5

- 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):

```
#Using the start parameter:

for x in range(2, 6):
  print(x)
```

output: 2 3 4 5

- 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, 30, **3**):

```
#Increment the sequence with 3 (default is 1)

for x in range(2, 30, 3):
  print(x)
```

output:2 5 8 11 14 17 20 23 26 29

Else in For Loop

- The else keyword in a for loop specifies a block of code to be executed when the loop is finished
- The else block will NOT be executed if the loop is stopped by a break statement.

#Print all numbers from 0 to 5, and print a message when the loop has ended:

```
for x in range(6):
  print(x)
else:
  print("Finally finished!")
```

output: 0 1 2 3 4 5

Finally finished!

#Break the loop when x is 3, and see what happens with the else block

```
for x in range(6):
  if x == 3: break
  print(x)
else:
  print("Finally finished!")
```

output:0 1 2

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":

#Print each adjective for every fruit

```
adj = ["red", "big", "tasty"]
fruits = ["apple", "banana", "cherry"]

for x in adj:
  for y in fruits:
    print(x, y)
```

output: red apple red banana red cherry

big apple big banana big cherry

tasty apple tasty banana tasty cherry

The pass Statement

- for loops cannot be empty, but if you for some reason have a for loop with no content, put in the pass statement to avoid getting an error.

```
for x in [0, 1, 2]:
  pass
```

UNIT5 / WEEK5: DATA COLLECTIONS

- ‘mutable’ is the ability of objects to change their values. These are often the objects that store a collection of data.
- Objects of built-in type that are mutable are:

- Lists
- Sets
- Dictionaries
- User-Defined Classes (It purely depends upon the user to define the characteristics)

Python Sets

- A set is an unordered collection of items. Every set element is unique (no duplicates) and must be immutable (cannot be changed).
- However, a set itself is mutable. We can add or remove items from it.
- Sets can also be used to perform mathematical set operations like union, intersection, symmetric difference, etc.

Creating Python Sets

- A set is created by placing all the items (elements) inside curly braces {}, separated by comma, or by using the built-in set() function.
- It can have any number of items and they may be of different types (integer, float, tuple, string etc.).
- But a set cannot have mutable elements like lists, sets or dictionaries as its elements.

```
# Different types of sets in Python
my_set = {1, 2, 3}          # set of integers
print(my_set)
my_set = {1.0, "Hello", (1, 2, 3)}          # set of mixed datatypes
```

```
print(my_set)
```

OUTPUT:

{1, 2, 3}

{1.0, (1, 2, 3), 'Hello'}

Creating an empty set

Empty curly braces {} will make an empty dictionary in Python. To make a set without any elements, we use the set() function without any argument.

```
# Distinguish set and dictionary while creating empty set
a = {}                  # initialize a with {}
print(type(a))          # check data type of
a = set()               # initialize a with set()
print(type(a))          # check data type of a
```

Modifying a set in Python

- We cannot access or change an element of a set using indexing or slicing. Set data type does not support it.
- We can add a single element using the add() method, and multiple elements using the update() method. The update() method can take tuples, lists, strings or other sets as its argument. In all cases, duplicates are avoided

```
my_set = {1, 3}                 # initialize my_set
print(my_set)
my_set.add(2)                   # add an element
print(my_set)                   # Output: {1, 2, 3}
```

```
my_set.update([2, 3, 4])        # add multiple elements
print(my_set)                   # Output: {1, 2, 3, 4}
```

Removing elements from a set

- A particular item can be removed from a set using the methods discard() and remove().
- The only difference between the two is that the discard() function leaves a set unchanged if the element is not present in the set. On the other hand, the remove() function will raise an error in such a condition (if element is not present in the set).

```
# Difference between discard() and remove()
my_set = {1, 3, 4, 5, 6}        # initialize my_set
print(my_set)
my_set.discard(4)               # discard an element
print(my_set)                   # Output: {1, 3, 5, 6}
my_set.remove(6)                # remove an element
print(my_set)                   # Output: {1, 3, 5}
```

Python Set Operations

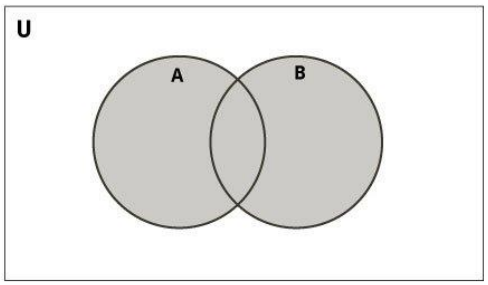
- Sets can be used to carry out mathematical set operations like union, intersection, difference and symmetric difference.
- We can do this with operators or methods.

Let us consider the following two sets for the following operations.

A= {1, 2, 3, 4, 5}

B= {4, 5, 6, 7, 8}

Set Union

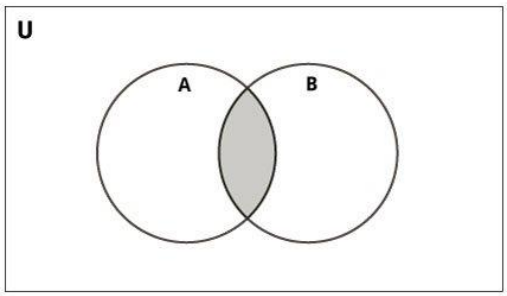


- Union of A and B is a set of all elements from both sets.
- Union is performed using | operator. Same can be accomplished using the union() method

```
# Set union method
A = {1, 2, 3, 4, 5}                    # initialize A and B
B = {4, 5, 6, 7, 8}
print(A | B)                           # use | operator
```

OUTPUT: {1, 2, 3, 4, 5, 6, 7, 8}

Set Intersection

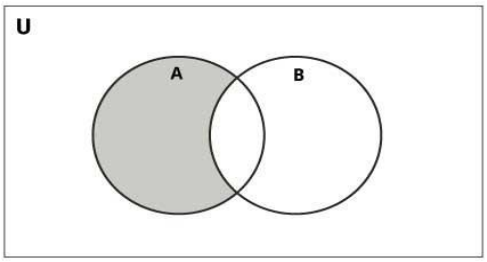


- Intersection of A and B is a set of elements that are common in both the sets.
- Intersection is performed using & operator. Same can be accomplished using the intersection() method.

```
# Intersection of sets
A = {1, 2, 3, 4, 5}                    # initialize A and B
B = {4, 5, 6, 7, 8}
print(A & B)                           # use & operator
```

OUTPUT:{4,5}

Set Difference

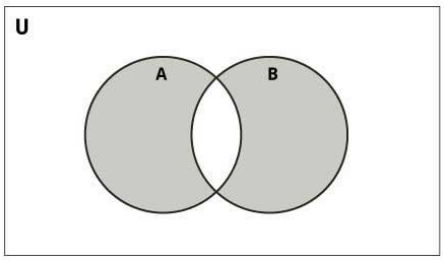


- Difference of the set B from set A(A - B) is a set of elements that are only in A but not in B. Similarly, B - A is a set of elements in B but not in A.
- Difference is performed using - operator. Same can be accomplished using the difference() method.

```
# Difference of two sets
A = {1, 2, 3, 4, 5}                    # initialize A and B
B = {4, 5, 6, 7, 8}
print(A - B)                           # use - operator on A
```

OUTPUT:{1,2,3}

Set Symmetric Difference



- Symmetric Difference of A and B is a set of elements in A and B but not in both (excluding the intersection).
- Symmetric difference is performed using ^ operator. Same can be accomplished using the method symmetric_difference().

```
# Symmetric difference of two sets
A = {1, 2, 3, 4, 5}                    # initialize A and B
B = {4, 5, 6, 7, 8}
print(A ^ B)                           # use ^ operator
```

OUTPUT: {1, 2, 3, 6, 7, 8}

Python Tuple

A tuple in Python is similar to a list. The difference between the two is that we cannot change the elements of a tuple once it is assigned whereas we can change the elements of a list.

Creating a Tuple

- A tuple is created by placing all the items (elements) inside parentheses (), separated by commas. The parentheses are optional, however, it is a good practice to use them.

- A tuple can have any number of items and they may be of different types (integer, float, list, string, etc.).

```
# Different types of tuples
my_tuple = ()                                   # Empty tuple
print(my_tuple)
my_tuple = (1, 2, 3)                            # Tuple having integers
print(my_tuple)
my_tuple = (1, "Hello", 3.4)                    # tuple with mixed datatypes
print(my_tuple)
my_tuple = ("mouse", [8, 4, 6], (1, 2, 3))      # nested tuple
print(my_tuple)
```

- Creating a tuple with one element is a bit tricky.

```
my_tuple = 3, 4.6, "dog"
print(my_tuple)
a, b, c = my_tuple                  # tuple unpacking is also possible
print(a)      # 3
print(b)      # 4.6
print(c)      # dog
```

- Having one element within parentheses is not enough. We will need a trailing comma to indicate that it is, in fact, a tuple.
- A tuple can also be created without using parentheses. This is known as tuple packing

```
# Creating a tuple having one element
my_tuple = ("hello",)
print(type(my_tuple))  # <class 'tuple'>
my_tuple = "hello",                     # Parentheses is optional
```

```
print(type(my_tuple))  # <class 'tuple'>
```

Access Tuple Elements

There are various ways in which we can access the elements of a tuple.

Indexing

- We can use the index operator `[]` to access an item in a tuple, where the index starts from 0.
- So, a tuple having 6 elements will have indices from 0 to 5. Trying to access an index outside of the tuple index range(6,7,... in this example) will raise an `IndexError`.
- The index must be an integer, so we cannot use float or other types. This will result in `TypeError`.

```
# Accessing tuple elements using indexing
my_tuple = ('p','e','r','m','i','t')
print(my_tuple[0])   # 'p'
print(my_tuple[5])   # 't'
OUTPUT: p , t

# print(my_tuple[6])  # IndexError: list index out of range


# my_tuple[2.0]              # Index must be an integer
# TypeError: list indices must be integers, not float

# nested tuple
n_tuple = ("mouse", [8, 4, 6], (1, 2, 3))
print(n_tuple[0][3])       # 's'                    # nested index
```

```
print(n_tuple[1][1])       # 4
```

OUTPUT: s, 4

Negative Indexing

- Python allows negative indexing for its sequences.
- The index of -1 refers to the last item, -2 to the second last item and so on.

```
# Negative indexing for accessing tuple elements
my_tuple = ('p', 'e', 'r', 'm', 'i', 't')
print(my_tuple[-1])
print(my_tuple[-6])
```

OUTPUT: ‘t’ , 'p'

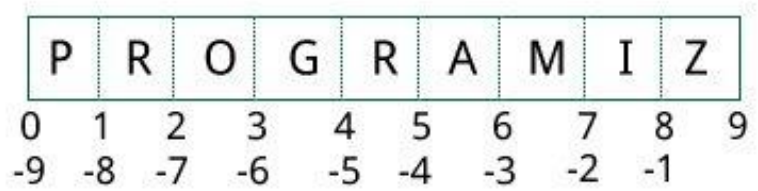
Slicing

We can access a range of items in a tuple by using the slicing operator colon `:`.

```
# Accessing tuple elements using slicing
my_tuple = ('p','r','o','g','r','a','m','i','z')

print(my_tuple[1:4])              # elements 2nd to 4th       # Output: ('r', 'o', 'g')
print(my_tuple[:-7])       # elements beginning to 2nd      # Output: ('p', 'r')
print(my_tuple[7:])        # elements 8th to end       # Output: ('i', 'z')
print(my_tuple[:])
# elements beginning to end       # Output: ('p', 'r', 'o', 'g', 'r', 'a', 'm', 'i', 'z')
```

- Slicing can be best visualized by considering the index to be between the elements as shown below.
- So if we want to access a range, we need the index that will slice the portion from the tuple.



Element Slicing in Python

Changing a Tuple

- elements of a tuple cannot be changed once they have been assigned. But, if the element is itself a mutable data type like a list, its nested items can be changed.

```
# Changing tuple values
my_tuple = (4, 2, 3, [6, 5])
my_tuple[1] = 9     # TypeError: 'tuple' object does not support item assignment

# However, item of mutable element can be changed
my_tuple[3][0] = 9    # Output: (4, 2, 3, [9, 5])
print(my_tuple)

# Tuples can be reassigned
my_tuple = ('p', 'r', 'o', 'g', 'r', 'a', 'm', 'i', 'z')
print(my_tuple)              # Output: ('p', 'r', 'o', 'g', 'r', 'a', 'm', 'i', 'z')
```

- We can use `+` operator to combine two tuples. This is called **concatenation**.
- We can also **repeat** the elements in a tuple for a given number of times using the `*` operator.

- Both `+` and `*` operations result in a new tuple.

```
# Concatenation
print((1, 2, 3) + (4, 5, 6))          # Output: (1, 2, 3, 4, 5, 6)
# Repeat
print(("Repeat",) * 3)                # Output: ('Repeat', 'Repeat', 'Repeat')
```

Deleting a Tuple

- As discussed above, we cannot change the elements in a tuple. It means that we cannot delete or remove items from a tuple.
- Deleting a tuple entirely, however, is possible using the keyword del.

```
# Deleting tuples
my_tuple = ('p', 'r', 'o', 'g', 'r', 'a', 'm', 'i', 'z')
del my_tuple[3]                       # can't delete items
# TypeError: 'tuple' object doesn't support item deletion
del my_tuple                          # Can delete an entire tuple
print(my_tuple)                       # NameError: name 'my_tuple' is not defined
```

UNIT6 / WEEK6: LIST

Python List

Python lists are one of the most versatile data types that allow us to work with multiple elements at once.

Create Python Lists

In Python, a list is created by placing elements inside square brackets `[]`, separated by commas.

```
my_list=[1,2,3]                    #list of integers
```

A list can have any number of items and they may be of different types (integer, float, string, etc.).

```
my_list=[ ]                        #empty list
my_list=[1,"hello",3.4]            #list with mixed datatypes
```

A list can also have another list as an item. This is called a nested list.

```
my_list=["mouse", [1 , 2, 3], ['a'] ]          #nested list
```

Access List Elements

There are various ways in which we can access the elements of a list.

List Index

- We can use the index operator `[]` to access an item in a list. In Python, indices start at 0. So, a list having 5 elements will have an index from 0 to 4.
- Trying to access indexes other than these will raise an `IndexError`.

- The index must be an integer. We can't use float or other types, this will result in TypeError.
- Nested lists are accessed using nested indexing.

```
my_list = ['p', 'r', 'o', 'b', 'e']

print(my_list[0])                    # first item                    # p

print(my_list[2])                    # third item                    # o

print(my_list[4])                    # fifth item                    # e

n_list = ["Happy", [2, 0, 1, 5]]             # Nested List

print(n_list[0][1])                  # Nested indexing

print(n_list[1][3])

print(my_list[4.0])          # Error! Only integer can be used for indexing
```

Negative indexing

- Python allows negative indexing for its sequences. The index of -1 refers to the last item, -2 to the second last item and so on.

```
# Negative indexing in lists

my_list = ['p','r','o','b','e']

print(my_list[-1])                           # last item

print(my_list[-5])                           # fifth last item
```

List Slicing in Python

We can access a range of items in a list by using the slicing operator `:`.

```
# List slicing in Python

my_list = ['p','r','o','g','r','a','m','i','z']

print(my_list[2:5])        # elements from index 2 to index 4

print(my_list[5:])         # elements from index 5 to end

print(my_list[:])          # elements beginning to end
```

Add/Change List Elements

- Lists are mutable, meaning their elements can be changed unlike string or tuple.
- We can use the assignment operator `=` to change an item or a range of items.

```
# Correcting mistake values in a list

odd = [2, 4, 6, 8]

odd[0] = 1                      # change the 1st item

print(odd)

odd[1:4] = [3, 5, 7]            # change 2nd to 4th items

print(odd)
```

OUTPUT: [1,4,6,8]

[1,3,5,7]

- We can add one item to a list using the append() method or add several items using the extend() method.

```
# Appending and Extending lists in Python

odd = [1, 3, 5]

odd.append(7)

print(odd)

odd.extend([9, 11, 13])

print(odd)
```

OUTPUT: [1,3,5,7]

[1,3,5,7,9,11,13]

- We can also use + operator to combine two lists. This is also called concatenation.
- The * operator repeats a list for the given number of times.

```
# Concatenating and repeating lists

odd = [1, 3, 5]

print(odd + [9, 7, 5])

print(["re"] * 3)
```

OUTPUT: [1,3,5,9,7,5]

['re','re','re']

- Furthermore, we can insert one item at a desired location by using the method insert() or insert multiple items by squeezing it into an empty slice of a list.

```
# Demonstration of list insert() method

odd = [1, 9]

odd.insert(1,3)

print(odd)

odd[2:2] = [5, 7]

print(odd)
```

OUTPUT: [1,3,9]

[1,3,5,7,9]

Delete List Elements

We can delete one or more items from a list using the Python del statement. It can even delete the list entirely.

```
# Deleting list items

my_list = ['p', 'r', 'o', 'b', 'l', 'e', 'm']

del my_list[2]                  # delete one item

print(my_list)

del my_list[1:5]                # delete multiple items

print(my_list)

del my_list                     # delete the entire list

print(my_list)                  # Error: List not defined
```

- We can use `remove()` to remove the given item or `pop()` to remove an item at the given index.
- The `pop()` method removes and returns the last item if the index is not provided. This helps us implement lists as stacks (first in, last out data structure).
- And, if we have to empty the whole list, we can use the `clear()` method.

```
my_list = ['p','r','o','b','l','e','m']

my_list.remove('p')

print(my_list)

print(my_list.pop(1))

print(my_list)

print(my_list.pop())

print(my_list)

my_list.clear()

print(my_list)
```

OUTPUT: ['r', 'o', 'b', 'l', 'e', 'm']

'o'

['r', 'b', 'l', 'e', 'm']

'm'

['r', 'b', 'l', 'e']

List Comprehension: Elegant way to create Lists

- List comprehension is an elegant and concise way to create a new list from an existing list in Python.
- A list comprehension consists of an expression followed by for statement inside square brackets.
- Here is an example to make a list with each item being increasing power of 2.

```
pow2 = [2 ** x for x in range(10)]

print(pow2)
```

OUTPUT: [1,2,4,8,16,32,64,128,256,512]

UNIT7 / WEEK7: DICTIONARY

Python Dictionary

- Python dictionary is an unordered collection of items. Each item of a dictionary has a key/value pair.
- Dictionaries are optimized to retrieve values when the key is known.

Creating Python Dictionary

- Creating a dictionary is as simple as placing items inside curly braces {} separated by commas.
- An item has a key and a corresponding value that is expressed as a pair (**key: value**).
- While the values can be of any data type and can repeat, keys must be of immutable type (string, number or tuple with immutable elements) and must be unique.

```
my_dict = {}                                  # empty dictionary
my_dict = {1: 'apple', 2: 'ball'}             # dictionary with integer keys
my_dict = {'name': 'John', 1: [2, 4, 3]}      # dictionary with mixed keys
my_dict = dict({1:'apple', 2:'ball'})         # using dict()
my_dict = dict([(1,'apple'), (2,'ball')])     # sequence having each item as pair
```

Accessing Elements from Dictionary

- While indexing is used with other data types to access values, a dictionary uses keys. Keys can be used either inside square brackets [] or with the get() method.
- If we use the square brackets [], KeyError is raised in case a key is not found in the dictionary. On the other hand, the get() method returns None if the key is not found.

```
my_dict = {'name': 'Jack', 'age': 26}          # get vs [] for retrieving elements
print(my_dict['name'])
print(my_dict.get('age'))
print(my_dict.get('address'))      # access keys which doesn't exist throws err
print(my_dict['address'])
```

OUTPUT:

Jack

26

None

KeyError: 'address'

Changing and Adding Dictionary elements

- Dictionaries are mutable. We can add new items or change the value of existing items using an assignment operator.
- If the key is already present, then the existing value gets updated. In case the key is not present, a new (**key: value**) pair is added to the dictionary.

```
# Changing and adding Dictionary Elements
my_dict = {'name': 'Jack', 'age': 26}
my_dict['age'] = 27                              # update value
print(my_dict)
my_dict['address'] = 'Downtown'                  # add item
print(my_dict)
```

OUTPUT:

{'name': 'Jack', 'age': 27}

{'name': 'Jack', 'age': 27, 'address': 'Downtown'}

Removing elements from Dictionary

- We can remove a particular item in a dictionary by using the pop() method. This method removes an item with the provided key and returns the value.
- The popitem() method can be used to remove and return an arbitrary (key, value) item pair from the dictionary. All the items can be removed at once, using the clear() method.
- We can also use the del keyword to remove individual items or the entire dictionary itself.

```
# Removing elements from a dictionary
squares = {1: 1, 2: 4, 3: 9, 4: 16, 5: 25}          # create a dictionary
print(squares.pop(4))          # remove a particular item, returns its value
print(squares)                 # remove an arbitrary item, return (key,value)
print(squares.popitem())
print(squares)
squares.clear()                # remove all items
print(squares)
del squares                    # delete the dictionary itself
print(squares)                 # Throws Error
```

OUTPUT:

```
16
{1: 1, 2: 4, 3: 9, 5: 25}
(5, 25)
{1: 1, 2: 4, 3: 9}
{}
NameError: name 'squares' is not defined
```

Dictionary Built-in Functions

Built-in functions like all(), any(), len(), cmp(), sorted(), etc. are commonly used with dictionaries to perform different tasks.

Function	Description
all()	Return True if all keys of the dictionary are True (or if the dictionary is empty).
any()	Return True if any key of the dictionary is true. If the dictionary is empty, return False.
len()	Return the length (the number of items) in the dictionary.
cmp()	Compares items of two dictionaries. (Not available in Python 3)
sorted()	Return a new sorted list of keys in the dictionary.

```
# Dictionary Built-in Functions
squares = {0: 0, 1: 1, 3: 9, 5: 25, 7: 49, 9: 81}
print(all(squares))
print(any(squares))
print(len(squares))
print(sorted(squares))
```

OUTPUT:

```
False
True
6
[0, 1, 3, 5, 7, 9]
```

Python Dictionary Comprehension

- Dictionary comprehension is an elegant and concise way to create a new dictionary from an iterable in Python.
- Dictionary comprehension consists of an expression pair (**key: value**) followed by a for statement inside curly braces {}.
- Here is an example to make a dictionary with each item being a pair of a number and its square.

```
# Dictionary Comprehension
squares = {x: x*x for x in range(6)}
print(squares)
```

(OR) This code is equivalent to

```
squares = {}
for x in range(6):
    squares[x] = x*x
print(squares)
```

OUTPUT:

{0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25}

UNIT8 / WEEK8: ARRAYS AND STRINGS

Python Arrays

An array is a collection of items stored at contiguous memory locations. The idea is to store multiple items of the same type together.

Creating a Array

- Array in Python can be created by importing array module.
- **array(*data_type*, *value_list*)** is used to create an array with data type and value list specified in its arguments.

```
# creating an array with integer type

import array as arr
a = arr.array('i', [1, 2, 3])
for i in range (0, 3):
    print (a[i], end =" ")
print()
```

OUTPUT: 1 2 3

Adding Elements to a Array

- Elements can be added to the Array by using built-in insert() function. Insert is used to insert one or more data elements into an array.
- append() is also used to add the value mentioned in its arguments at the end of the array.

```
# Adding Elements to a Array
import array as arr                 # importing "array" for array creations
a = arr.array('i', [1, 2, 3])       # array with int type
print ("Array before insertion : ", end =" ")
for i in range (0, 3):
        print (a[i], end =" ")
print()
a.insert(1, 4)                      # inserting array using insert() function
```

```
print ("Array after insertion : ", end =" ")
for i in (a):
        print (i, end =" ")
print()
a.append(5)                    # adding an element using append()
print ("Array after append : ", end =" ")
for i in (a):
        print (i, end =" ")
print()
```

OUTPUT:

Array before insertion : 1 2 3

Array after insertion : 1 4 2 3

Array after append : 1 4 2 3 5

Accessing elements from the Array

- In order to access the array items refer to the index number.
- Use the index operator [ ] to access an item in a array. The index must be an integer.

```
# accessing of element from list
import array as arr                    # importing array module
a = arr.array('i', [1, 2, 3, 4, 5, 6])    # array with int type
print("Access element is: ", a[0])        # accessing element of array
print("Access element is: ", a[3])        # accessing element of array
```

OUTPUT:

Access element is: 1

Access element is: 4

Removing Elements from the Array

- Elements can be removed from the array by using built-in remove() function but an Error arises if element doesn't exist in the set.
- Remove() method only removes one element at a time, to remove range of elements, iterator is used.
- pop() function can also be used to remove and return an element from the array, but by default it removes only the last element of the array, to remove element from a specific position of the array, index of the element is passed as an argument to the pop() method.

```
# Removal of elements in a Array
import array                    #import array as arr
arr = array.array('i', [1, 2, 3, 1, 5])    # initializes array with signed integers
print ("The new created array is : ", end ="")        # printing original array
for i in range (0, 5):
    print (arr[i], end =" ")
print ("\r")
print ("The popped element is : ", end ="")
print (arr.pop(2))   # using pop() to remove element at 2nd position
print ("The array after popping is : ", end ="")
for i in range (0, 4):
    print (arr[i], end =" ")
print("\r")
arr.remove(1)              # using remove() to remove 1st occurrence of 1
print ("The array after removing is : ", end ="")
for i in range (0, 3):
    print (arr[i], end =" ")
```

OUTPUT:

The new created array is : 1 2 3 1 5

The popped element is : 3

The array after popping is : 1 2 1 5

The array after removing is : 2 1 5

Process finished with exit code 0

Searching element in a Array

- In order to search an element in the array we use a python in built index() method.
- This function returns the index of the first occurrence of value mentioned in arguments.

```
# searching an element in array
import array
arr = array.array('i', [1, 2, 3, 1, 2, 5])
print ("The new created array is : ", end ="")
for i in range (0, 6):
        print (arr[i], end =" ")
print ("\r")
print ("The index of 1st occurrence of 2 is : ", end ="")
print (arr.index(2))
print ("The index of 1st occurrence of 1 is : ", end ="")
print (arr.index(1))
```

OUTPUT:

The new created array is : 1 2 3 1 2 5

The index of 1st occurrence of 2 is : 1

The index of 1st occurrence of 1 is : 0

Updating Elements in a Array

In order to update an element in the array we simply reassign a new value to the desired index we want to update.

```
# how to update an element in array
import array
arr = array.array('i', [1, 2, 3, 1, 2, 5])
print ("Array before updation : ", end ="")
for i in range (0, 6):
        print (arr[i], end =" ")
print ("\r")
arr[2] = 6
print("Array after updation : ", end ="")
for i in range (0, 6):
        print (arr[i], end =" ")
print()
arr[4] = 8
print("Array after updation : ", end ="")
for i in range (0, 6):
        print (arr[i], end =" ")
```

OUTPUT:

```
Array before updation : 1 2 3 1 2 5
Array after updation : 1 2 6 1 2 5
Array after updation : 1 2 6 1 8 5
```

Python Strings

- A string is a sequence of characters.
- Computers do not deal with characters, they deal with numbers (binary).

- Even though you may see characters on your screen, internally it is stored and manipulated as a combination of 0s and 1s.
- This conversion of character to a number is called encoding, and the reverse process is decoding. ASCII and Unicode are some of the popular encodings used.
- In Python, a string is a sequence of Unicode characters.
- Unicode was introduced to include every character in all languages and bring uniformity in encoding.

How to create a string in Python?

Strings can be created by enclosing characters inside a single quote or double-quotes. Even triple quotes can be used in Python but generally used to represent multiline strings and docstrings.

```
# defining strings in Python
my_string = 'Hello'
print(my_string)
my_string = "Hello"
print(my_string)
my_string = '''Hello'''
print(my_string)
my_string = """Hello, welcome to
        the world of Python"""
print(my_string)
```

OUTPUT:

```
Hello
Hello
Hello
Hello, welcome to
        the world of Python
```

How to access characters in a string?

- We can access individual characters using indexing and a range of characters using slicing.
- Index starts from 0.
- Trying to access a character out of index range will raise an IndexError. The index must be an integer.
- We can't use floats or other types, this will result into TypeError.
- Python allows negative indexing for its sequences.
- The index of -1 refers to the last item, -2 to the second last item and so on.
- We can access a range of items in a string by using the slicing operator :(colon).

```
#Accessing string characters in Python
str = 'POLYTECHNIC'
print('str = ', str)
print('str[0] = ', str[0])
print('str[-1] = ', str[-1])
print('str[1:5] = ', str[1:5])
print('str[5:-2] = ', str[5:-2])                #slicing 6th to 2nd last character
```

OUTPUT:

```
str = POLYTECHNIC
str[0] =  p
str[-1] =  z
str[1:5] =  rogr
str[5:-2] =  am
```

How to change or delete a string

Strings are immutable. This means that elements of a string cannot be changed once they have been assigned. We can simply reassign different strings to the same name.

```
my_string = 'programiz'
 my_string[5] = 'a'
TypeError: 'str' object does not support item assignment
 my_string = 'Python
```

We cannot delete or remove characters from a string. But deleting the string entirely is possible using the `del` keyword.

```
del my_string[1]
```

Built-in functions to Work with Python

- Various built-in functions that work with sequence work with strings as well.
- Some of the commonly used ones are `enumerate()` and `len()`.
  The `enumerate()` function returns an enumerate object.
- It contains the index and value of all the items in the string as pairs.
- This can be useful for iteration.
- Similarly, `len()` returns the length (number of characters) of the string.

```
str = 'cold'
list_enumerate = list(enumerate(str))                # enumerate()
print('list(enumerate(str) = ', list_enumerate)
print('len(str) = ', len(str))                       #character count
```

OUTPUT:

```
list(enumerate(str) =  [(0, 'c'), (1, 'o'), (2, 'l'), (3, 'd')]
len(str) =  4
```

UNIT9 / WEEK9: FUNCTIONS

Python Functions

- A function is a block of code which only runs when it is called.
- we 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()
```

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.
- *Arguments* are often shortened to *args* in Python documentations.
- The terms *parameter* and *argument* can be used for the same thing: information that are passed into a function
- 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(fname):
  print(fname + " Pal")
my_function("Tulasi")
my_function("Bhakthi")
my_function("Druva")
```

OUTPUT: Tulasi Pal

Bhakthi Pal

Druva Pal

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, not more, and not less.

```
def my_function(fname, lname):
  print(fname + " " + lname)
my_function("VIJAY", "RAJ")
```

Arbitrary Arguments, *args

- *Arbitrary Arguments* are often shortened to **args* in Python documentations.
- 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.

```
def my_function(*kids):
  print("The youngest child is " + kids[2])
my_function("Emil", "Tobias", "Linus")
```

OUTPUT: The youngest child is Linus

Keyword Arguments

- You can also send arguments with the *key* = *value* syntax.
- This way the order of the arguments does not matter.
- The phrase *Keyword Arguments* are often shortened to *kwargs* in Python documentations.

```
def my_function(child3, child2, child1):
  print("The youngest child is " + child3)
my_function(child1 = "Emil", child2 = "Tobias", child3 = "Linus")
```

- If the number of keyword arguments is unknown, add a double ** before the parameter name:

```
def my_function(**kid):
```

Passing a List as an Argument

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

```
def my_function(food):
  for x in food:
    print(x)
fruits = ["apple", "banana", "cherry"]
my_function(fruits)
```

Return Values

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

```
def my_function(x):
  return 5 * x
print(my_function(3))
print(my_function(5))
print(my_function(9))
```

The pass Statement

function definitions cannot be empty, but a function definition with no content, put in the pass statement to avoid getting an error.

```
def myfunction():
  pass
```

Scope and Lifetime of variables

- Scope of a variable is the portion of a program where the variable is recognized.
- Parameters and variables defined inside a function are not visible from outside the function. Hence, they have a local scope.
- The lifetime of a variable is the period throughout which the variable exists in the memory.
- The lifetime of variables inside a function is as long as the function executes.
- They are destroyed once we return from the function. Hence, a function does not remember the value of a variable from its previous calls.

```
def my_func():
        x = 10
        print("Value inside function:",x)
x = 20
my_func()
print("Value outside function:",x)
```

OUTPUT:

Value inside function: 10

Value outside function: 20

- This is because the variable x inside the function is different (local to the function) from the one outside.
- Although they have the same names, they are two different variables with different scopes.
- On the other hand, variables outside of the function are visible from inside. they have a global scope.
- We can read these values from inside the function but cannot change (write) them.
- In order to modify the value of variables outside the function, they must be declared as global variables using the keyword global.

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.
- In this example, tri_recursion() is a function that we have defined to call itself ("recurse"). We use the k variable as the data, which decrements (-1) every time we recurse. The recursion ends when the condition is not greater than 0 (i.e. when it is 0).

```
def tri_recursion(k):
  if(k > 0):
    result = k + tri_recursion(k - 1)
    print(result)
  else:
```

```
    result = 0
  return result

print("\n\nRecursion Example Results")
tri_recursion(6)
```

OUTPUT:

Recursion Example Results

1

3

6

10

15

21

Anonymous function

- A lambda function is a small anonymous function.
- A lambda function can take any number of arguments, but can only have one expression.

Syntax

lambda *arguments* : *expression*

```
x = lambda a : a + 10
print(x(5))
```

OUTPUT: 15

Multiply argument a with argument b and return the result:

```
x = lambda a, b : a * b
print(x(5, 6))
```

OUTPUT: 30

Why Use Lambda Functions?

- The power of lambda is better shown when you use them as an anonymous function inside another function.
- Use lambda functions when an anonymous function is required for a short period of time.
- Say you have a function definition that takes one argument, and that argument will be multiplied with an unknown number:

```
def myfunc(n):
  return lambda a : a * n
```

Use that function definition to make a function that always doubles the number you send in:

```
def myfunc(n):
  return lambda a : a * n
mydoubler = myfunc(2)
print(mydoubler(11))
```

Or, use the same function definition to make a function that always *triples* the number you send in:

```
def myfunc(n):
  return lambda a : a * n
mytripler = myfunc(3)
print(mytripler(11))
```

Or, use the same function definition to make both functions, in the same program:

```
def myfunc(n):
  return lambda a : a * n
mydoubler = myfunc(2)
mytripler = myfunc(3)
print(mydoubler(11))
print(mytripler(11))
```

UNIT10 / WEEK10: MODULES AND PACKAGES

Python Module

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

Create a Module

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

```
def greeting(name):
  print("Hello, " + name)
```

Use a Module

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

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

```
def greeting(name):
  print("Hello, " + name)

import mymodule
mymodule.greeting("Jonathan")
```

- When using a function from a module, use the **syntax: *module_name.function_name*.**

Variables in Module

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

Save this code in the file mymodule.py

```
person1 = {
  "name": "John",
  "age": 36,
  "country": "Norway"
}
```

```
import mymodule
```

```
a = mymodule.person1["age"]
print(a)
```

Naming a Module

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

Import From Module

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

```
def greeting(name):
  print("Hello, " + name)
person1 = {
  "name": "John",
  "age": 36,
  "country": "Norway"
}0
```

OUTPUT:

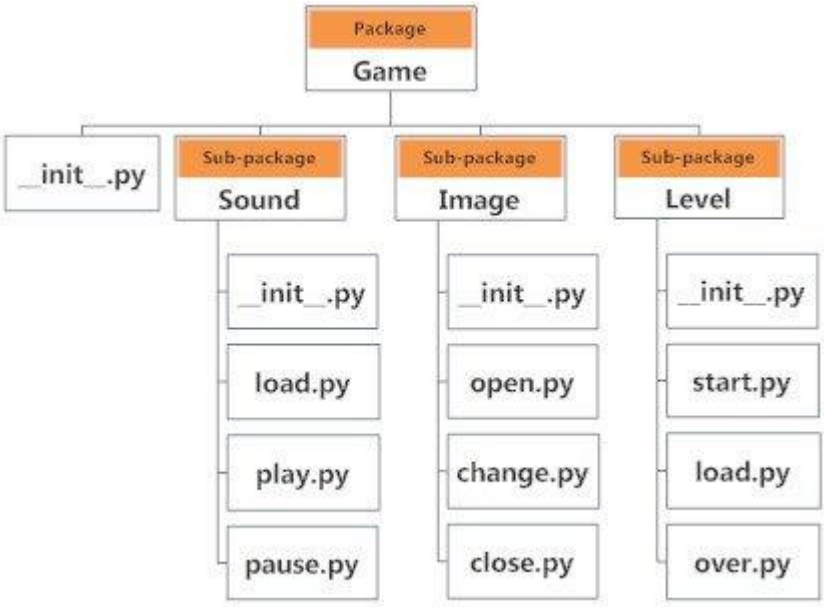
36

36

Python packages

- Python has packages for directories and modules for files.
- As our application program grows larger in size with a lot of modules, we place similar modules in one package and different modules in different packages.
- This makes a project (program) easy to manage and conceptually clear.
- Similarly, as a directory can contain subdirectories and files, a Python package can have sub-packages and modules.
- A directory must contain a file named __init__.py in order for Python to consider it as a package.

- This file can be left empty but we generally place the initialization code for that package in this file.
- Here is an example. Suppose we are developing a game. One possible organization of packages and modules could be as shown in the figure below.



Package Module Structure in Python Programming

Importing module from a package

- We can import modules from packages using the dot (.) operator.
- For example, if we want to import the start module in the above example, it can be done as follows:

- from python_packages import first,second

  first.tuna()

  second.tuna()
- Another way of importing just the required function (or class or variable) from a module within a package would be as follows:
- from python_packages import first,second

Check if PIP is Installed

- pip –version , type this command in command prompt

Install PIP

- pip install camelCase, type this command in command prompt

- pip list ,command to list all the packages installed on your system

Python - Package __init__.py Files

- Each directory named within the path of a package import statement should contain a file named __init__.py.
- Consider the following import statement:

```
import dir1.dir2.mod
```

- Both dir1 and dir2 must contain a file called __init__.py.

__init__.py

- The __init__.py files can contain Python code, just like normal module files.
- Their code is run automatically the first time a Python program imports a directory.
- They perform initialization steps required by the package.
- These files can also be completely empty.

Python Random Module

- Python offers random module that can generate random numbers.
- These are pseudo-random number as the sequence of number generated depends on the seed.

- If the seeding value is same, the sequence will be the same.

For example, if you use 2 as the seeding value, you will always see the following sequence.

```
import random
random.seed(2)
print(random.random())
print(random.random())
print(random.random())
```

OUTPUT:

0.9560342718892494

0.9478274870593494

0.05655136772680869

Python Math Module

Python math module is defined as the most famous mathematical functions, which includes trigonometric functions, representation functions, logarithmic functions, etc. Furthermore, it also defines two mathematical constants, i.e., Pie and Euler number, etc.

math.log10()

This method returns base 10 logarithm of the given number and called the standard logarithm.

Example

```
1. import math
2. x=13  # small value of of x
3. print('log10(x) is :', math.log10(x))
```

OUTPUT;
log10(x) is : 1.1139433523068367

Python Emojis

- There are multiple ways we can print the Emojis in Python.
- Let's see how to print Emojis with Unicodes, CLDR names and emoji module.

Using Unicodes:

Every emoji has a Unicode associated with it.

- Emojis also have a CLDR short name, which can also be used.
- From the list of unicodes, replace "+" with "000". For example – "U+1F600" will become "U0001F600" and prefix the unicode with "\" and print it.

```
print("\U0001f600")          # grinning face
print("\U0001F606")          # grinning squinting face
print("\U0001F923")          # rolling on the floor laughing
```

UNIT11/ WEEK11:NUMPY AND PANDAS

Numpy

- NumPy is a Python library used for working with arrays.
- It also has functions for working in domain of linear algebra, fourier transform, and matrices.
- NumPy stands for Numerical Python.
- NumPy aims to provide an array object that is up to 50x faster than traditional Python lists.
- The array object in NumPy is called ndarray, it provides a lot of supporting functions that make working with ndarray very easy.
- Arrays are very frequently used in data science, where speed and resources are very important.

Installation of NumPy

If you have Python and PIP already installed on a system, then installation of NumPy is very easy.

Install it using this command:

C:\Users\RJSP>**pip install numpy**

Numpy Arithmetic

You could use arithmetic operators + - * / directly between NumPy arrays, but this section discusses an extension of the same where we have functions that can take any array-like objects e.g. lists, tuples etc. and perform arithmetic *conditionally*.

Addition

The add() function sums the content of two arrays, and return the results in a new array.

Add the values in arr1 to the values in arr2:

```
import numpy as np
arr1 = np.array([10, 11, 12, 13, 14, 15])
arr2 = np.array([20, 21, 22, 23, 24, 25])
newarr = np.add(arr1, arr2)
print(newarr)
```

OUTPUT:

```
[30 32 34 36 38 40]
```

Subtraction

The subtract() function subtracts the values from one array with the values from another array, and return the results in a new array.

Subtract the values in arr2 from the values in arr1:

```
import numpy as np
arr1 = np.array([10, 20, 30, 40, 50, 60])
arr2 = np.array([20, 21, 22, 23, 24, 25])
newarr = np.subtract(arr1, arr2)
print(newarr)
```

OUTPUT:

```
[-10 -1 8 17 26 35]
```

Multiplication

The multiply() function multiplies the values from one array with the values from another array, and return the results in a new array.

Multiply the values in arr1 with the values in arr2:

```
import numpy as np
arr1 = np.array([10, 20, 30, 40, 50, 60])
arr2 = np.array([20, 21, 22, 23, 24, 25])
newarr = np.multiply(arr1, arr2)
print(newarr)
```

OUTPUT:

```
[200 420 660 920 1200 1500]
```

Division

The divide() function divides the values from one array with the values from another array, and return the results in a new array.

Divide the values in arr1 with the values in arr2:

```
import numpy as np
arr1 = np.array([10, 20, 30, 40, 50, 60])
arr2 = np.array([3, 5, 10, 8, 2, 33])
newarr = np.divide(arr1, arr2)
print(newarr)
```

OUTPUT:

```
[3.33333333 4. 3. 5. 25. 1.81818182]
```

NumPy Array

- NumPy is a package for scientific computing which has support for a powerful N-dimensional array object
- NumPy provides multidimensional array of numbers

```
import numpy as np
A = np.array([[1, 2, 3], [3, 4, 5]])
print(A)
A = np.array([[1.1, 2, 3], [3, 4, 5]]) # Array of floats
print(A)
A = np.array([[1, 2, 3], [3, 4, 5]], dtype = complex) # Array of complex numbers
print(A)
```

OUTPUT:

```
[[1 2 3]
 [3 4 5]]
[[1.1 2.  3. ]
 [3.  4.  5. ]]
```

```
[[1.+0.j 2.+0.j 3.+0.j]
 [3.+0.j 4.+0.j 5.+0.j]]
```

NumPy Statistical Functions

NumPy is equipped with the following statistical functions:

1. np.amin()- This function determines the minimum value of the element along a specified axis.
2. np.amax()- This function determines the maximum value of the element along a specified axis.
3. np.mean()- It determines the mean value of the data set.
4. np.median()- It determines the median value of the data set.
5. np.std()- It determines the standard deviation
6. np.var – It determines the variance.
7. np.ptp()- It returns a range of values along an axis.
8. np.average()- It determines the weighted average
9. np.percentile()- It determines the nth percentile of data along the specified axis.

```
import numpy as np
arr= np.array([[1,23,78],[98,60,75],[79,25,48]])
print(arr)
#Minimum Function
print(np.amin(arr))
#Maximum Function
print(np.amax(arr))
```

Python Pandas

- Pandas is a Python library used for working with data sets.
- It has functions for analyzing, cleaning, exploring, and manipulating data.
- Pandas allows us to analyze big data and make conclusions based on statistical theories.
- Pandas can clean messy data sets, and make them readable and relevant.

Installation of Pandas

```
C:\Users\RJSP>pip install pandas
```

```
import pandas

mydataset = {

  'cars': ["BMW", "Volvo", "Ford"],

  'passings': [3, 7, 2]

}

myvar = pandas.DataFrame(mydataset)

print(myvar)
```

OUTPUT:

```
cars  passings

0    BMW        3

1  Volvo        7

2   Ford        2
```

Pandas Series

- A Pandas Series is like a column in a table.
- It is a one-dimensional array holding data of any type.

```
import pandas as pd

a = [1, 7, 2]

myvar = pd.Series(a)

print(myvar)
```

OUTPUT:

```
0    1
```

1 7

2 2

dtype: int64

Labels

If nothing else is specified, the values are labeled with their index number. First value has index 0, second value has index 1 etc.

This label can be used to access a specified value.

DataFrames

A Pandas DataFrame is a 2 dimensional data structure, like a 2 dimensional array, or a table with rows and columns.

Create a simple Pandas DataFrame:

```
import pandas as pd
data = {
  "calories": [420, 380, 390],
  "duration": [50, 40, 45]
}
#load data into a DataFrame object:
df = pd.DataFrame(data)
print(df)
```

Locate Row

- As you can see from the result above, the DataFrame is like a table with rows and columns.
- Pandas use the loc attribute to return one or more specified row(s)

```
#refer to the row index:
print(df.loc[0])
```

UNIT12 / WEEK12: CONTROL FLOW

Python File I/O

- Files are named locations on disk to store related information. They are used to permanently store data in a non-volatile memory (e.g. hard disk).
- Since Random Access Memory (RAM) is volatile (which loses its data when the computer is turned off), we use files for future use of the data by permanently storing them.
- When we want to read from or write to a file, we need to open it first. When we are done, it needs to be closed so that the resources that are tied with the file are freed.

Hence, in Python, a file operation takes place in the following order:

1. Open a file
2. Read or write (perform operation)
3. Close the file

File Handling

- The key function for working with files in Python is the open() function.
- The open() function takes two parameters; *filename*, and *mode*.
- There are four different methods (modes) for opening a file:

"r" - Read - Default value. Opens a file for reading, error if the file does not exist

"a" - Append - Opens a file for appending, creates the file if it does not exist

"w" - Write - Opens a file for writing, creates the file if it does not exist

"x" - Create - Creates the specified file, returns an error if the file exists

In addition you can specify if the file should be handled as binary or text mode

"t" - Text - Default value. Text mode

"b" - Binary - Binary mode (e.g. images

Syntax

- To open a file for reading it is enough to specify the name of the file:

```
f = open("demofile.txt")
```

The code above is the same as:

```
f = open("demofile.txt", "rt")
```

Because "r" for read, and "t" for text are the default values, you do not need to specify them.

Open a File on the Server

Assume we have the following file, located in the same folder as Python:

demofile.txt

```
Hello! Welcome to demofile.txt
This file is for testing purposes.
Good Luck!
```

- To open the file, use the built-in open() function.
- The open() function returns a file object, which has a read() method for reading the content of the file:

```
f = open("demofile.txt", "r")
print(f.read())
```

OUTPUT:

Hello! Welcome to demofile.txt

This file is for testing purposes.

Good Luck!

- If the file is located in a different location, you will have to specify the file path, like this:

Open a file on a different location:

```
f = open("D:\\myfiles\welcome.txt", "r")
print(f.read())
```

Read Only Parts of the File

- By default the read() method returns the whole text, but you can also specify how many characters you want to return:

Return the 5 first characters of the file:

```
f = open("demofile.txt", "r")
print(f.read(5))
```

OUTPUT: Hello

Read Lines

- You can return one line by using the readline() method:

Read one line of the file:

```
f = open("demofile.txt", "r")
print(f.readline())
```

OUTPUT:

Hello! Welcome to demofile.txt

- By calling readline() two times, you can read the two first lines:

Read two lines of the file:

```
f = open("demofile.txt", "r")
print(f.readline())
print(f.readline())
```

OUTPUT:

Hello! Welcome to demofile.txt

This file is for testing purposes.

- By looping through the lines of the file, you can read the whole file, line by line:

Loop through the file line by line:

```
f = open("demofile.txt", "r")
for x in f:
  print(x)
```

OUTPUT:

Hello! Welcome to demofile.txt

This file is for testing purposes.

Good Luck!

Hello! Welcome to demofile.txt

Close Files

It is a good practice to always close the file when you are done with it.

Close the file when you are finish with it:

```
f = open("demofile.txt", "r")
print(f.readline())
f.close()
```

Note: You should always close your files, in some cases, due to buffering, changes made to a file may not show until you close the file.

Write to an Existing File

To write to an existing file, you must add a parameter to the open() function:

"a" - Append - will append to the end of the file

"w" - Write - will overwrite any existing content

Open the file "demofile2.txt" and append content to the file:

```
f = open("demofile.txt", "a")
f.write("Now the file has more content!")
f.close()
```

```
f = open("demofile1.txt", "r")    #open and read the file after the appending:
print(f.read())
```

OUTPUT:

```
Hello! Welcome to demofile2.txt
This file is for testing purposes.
Good Luck!Now the file has more content!
```

Open the file "demofile3.txt" and overwrite the content:

```
f = open("demofile3.txt", "w")
f.write("Woops! I have deleted the content!")
f.close()
#open and read the file after the appending:
f = open("demofile3.txt", "r")
print(f.read())
```

OUTPUT:

```
 Woops! I have deleted the content!
```

Note: the "w" method will overwrite the entire file.

Create a New File

To create a new file in Python, use the open() method, with one of the following parameters:

"x" - Create - will create a file, returns an error if the file exist

"a" - Append - will create a file if the specified file does not exist

"w" - Write - will create a file if the specified file does not exist

Create a file called "myfile.txt":

```
f = open("myfile.txt", "x")
```

Result: a new empty file is created!

Create a new file if it does not exist:

```
f = open("myfile.txt", "w"pip

)
```

Delete a File

To delete a file, you must import the OS module, and run its os.remove() function:

Remove the file "demofile.txt":

```
import os
os.remove("demofile.txt")
```

Check if File exist:

To avoid getting an error, you might want to check if the file exists before you try to delete it:

Check if file exists, *then* delete it:

```
import os
if os.path.exists("demofile.txt"):
  os.remove("demofile.txt")
else:
  print("The file does not exist")
```

Delete Folder

To delete an entire folder, use the os.rmdir() method:

Remove the folder "myfolder":

```
import os
os.rmdir("myfolder")
```

Note: You can only remove *empty* folders.

UNIT13 / WEEK13: ERROR AND EXCEPTION HANDLING

Python Try Except

- The try block lets you test a block of code for errors.
- The except block lets you handle the error.
- The finally block lets you execute code, regardless of the result of the try- and except blocks.

Exception Handling

- When an error occurs, or exception as we call it, Python will normally stop and generate an error message.
- These exceptions can be handled using the try statement:

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

```
try:
  print(x)
except:
  print("An exception occurred")
```

OUTPUT:

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:

This statement will raise an error, because x is not defined:

```
print(x)
```

OUTPUT: error

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:

Print one message if the try block raises a NameError and another for other errors:

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

Else

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

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

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

OUTPUT:

Hello

Nothing went wrong

Finally

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

```
try:
  print(x)
except:
  print("Something went wrong")
finally:
  print("The 'try except' is finished")
```

OUTPUT:

Something went wrong

The 'try except' is finished

- This can be useful to close objects and clean up resources:

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

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

OUTPUT:

Something went wrong when opening the file

The program can continue, without leaving the file object open.

Raise an exception

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

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

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

```
x = -1

if x < 0:
  raise Exception("Sorry, no numbers below zero")
```

OUTPUT: error

- The raise keyword is used to raise an exception.
- You can define what kind of error to raise, and the text to print to the user.

Raise a TypeError if x is not an integer:

```
x = "hello"
if not type(x) is int:
  raise TypeError("Only integers are allowed")
```

OUTPUT: error