Chapter 1: Introduction to Python

Subchapter 1.1: Basics of Python Programming

Introduction to Python

- Python is a high level language
- It is a interpreted language i.e checks and executes line by line
- Huge amount of libraries present
- Object oriented language

Basic Syntax

- Python follows indentation
- We don’t have to specify the type of variable
- # used for single line comment
- “”” “”” triple quotes are used for multiline comments

Variables

Variables are used to store values in

Data Types

- Data type says the type of variable
- We don't have to explicitly declare the variable, python does it on own

Types :

1. **Int** - stores integer values
2. **float** - stores decimal point values
3. **string** - sequence of characters

4. **list , tuple, dictionary and set**

LIST	TUPLE	DICTIONARY	SET
Allows duplicate members	Allows duplicate members	No duplicate members	No duplicate members
Changeble	Not changeable	Changeable, indexed	Cannot be changed, but can be added, non -indexed
Ordered	Ordered	Unordered	Unordered
Square bracket [ ]	Round brackets ( )	Curly brackets{ }	Curly brackets{ }

Assignment: Create a BMI Calculator.

- The Python code functions as a Body Mass Index (BMI) calculator, utilizing metric measures for height in meters and weight in kilograms.
- User input is obtained for height and weight, and the BMI is calculated using the formula **BMI = weight / (height * height).** The result is rounded to one decimal place.
- The program categorises BMI results: under 18.5 suggests underweight, 18.5-25 as normal, 25-30 as overweight (time to work on weight), and above 30 warns of obesity, emphasising health importance.
- Overall, the code serves as a user-friendly tool for assessing BMI and offering guidance on potential health considerations tied to the calculated value using simple python concepts.

Subchapter 1.2: Control Flow

Topics: Conditional Statements, Loops.
Assignment: Develop a Text-based Adventure Game.

Conditional Statements:

if statement:

If a condition satisfies then, a block of code present in the **if** statement gets executed.

elif statement:

If a condition fails, then control comes to the elif block.
There can be any number of elif.

else statement:

If neither of if and elif conditions satisfies, then the else part will be executed.

```
b = -3
if b>0 :
    print(b, " is a positive integer")
elif b<0:
    print(b, " is a negative integer")
else:
    print(b, " is neighter +ve nor -ve")
```

Loops:

Loops are used to repeat a piece of code desired no.of times without writing it again & again.
The code will be executed until the given condition becomes false

while loop:

With the while loop we can execute a set of statements as long as a condition is true.

Break:

When a condition is met, the control comes out of the loop.

Continue:

Skip the current iteration

Else statement:

Used when while is completed.

For loop:

We have all the features which are there in a while, with that we've range().

The range() Function:

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

A range function has start, end and step.

```
for i in range (4):
    print(i)

for i in range (1, 4):
    print(i)

for x in range(5, 101, 5):
    print(x)
else:
    print("the stopping number : ", x)
```

Nested for loop:

Loop inside a loop

```
# nested for loop
names = ["Dave", "Sara", "John"]
actions = ["codes", "eats", "sleeps"]

for x in names:
    for y in actions:
        print(x + " " + " " + y)
```

Assignment:

Game Concept:
Embarks on an adventure in a mysterious forest, filled with magical creatures and hidden treasures. The ultimate quest is to discover the heart of the forest and return to the human world.

While Loop:

- Maintains continuous player engagement by employing `while` loops, ensuring the game progresses until a win or game-over condition is met.

User Choices:

- Utilises `input` and `int` conversions to capture and process user choices, steering the storyline and determining the game's outcome.

Continue and Break:

- Implements `continue` to prompt users for a new input in case of an invalid choice, sustaining the game's flow.
- Utilises `break` to exit the loop and conclude the game when specific conditions, like finding the treasure, are satisfied.

Subchapter 1.3: Functions and Modules

Topics: Creating Functions, Python Modules.
Assignment: Design a Financial Calculator Module.

Functions :

Functions are reusable blocks of code.

1. Functions in Python are defined using the `def` keyword.
2. To execute a function, use its name followed by parentheses in a function call.
3. Parameters are placeholders in a function's definition, while arguments are the actual values passed during a function call.
4. Functions can return values using the `return` statement; if omitted, the function returns `None` by default.
5. Variables inside functions are local; those outside functions are global.
6. Default parameter values can be set in function definitions.
7. Functions can accept a variable number of arguments using `*args` and `**kwargs`.

```
# *args - if we dont know the number of parameters - tuple
def sum(*args):
    print(args)
    print(type(args))
sum(4, 5, 2, 54)

# **kwrgs - defines a dictionary
def items(**kwargs):
    print(kwargs)
    print(type(kwargs))
items(first="vishnu", last="vishwas")
```

Output:
The total sum is : 37
(4, 5, 2, 54)
<class 'tuple'>

{'first': 'vishnu', 'last': 'vishwas'}
<class 'dict'>

Module

A module is a file containing Python definitions and statements.

Different ways to use module

```
import math
from math import pi as p
Import math as m
```

Contents of module

```
print(dir(rd))
```

The modules can be inbuilt or user defined.

To check the main function of current file

```
# prints the main function of code
print(__name__)
#prints the main function of kanasas module
print(kansas.__name__)
```

Assignment :

Building a financial calculator
Making use of python functions and modules
This module can be imported and used by other programs.

- calculate future Value
- calculate periodic Payment Loan
- calucate periodic payment annumity
- calculate rate of intrest
- calculate number of compouding periods
- calculating the present value

Subchapter 1.4: Data Structures

Topics: Lists, Tuples, Dictionaries.
Assignment: Implement a Contact Management System.

Lists

A list is a data structure, where multiple elements can be stored and changed/updated.i.e Mutable

Creating a List

```
users = ['Vishnu', 'Vishwas', 'Ankith', 'Dhanush']
data = ['Vishnu', 20, True]
emptylist = []
```

Checking for element in list

```
print("Vishnu" in users)
print("Vishnu" in emptylist)
```

Slicing

```
print(users[0:3])                # excludes last element
print(users[:])                  # Prints all elements
print(users[-3:-1])              # Accessing from -ve index
```

Length of List

```
print("Length of data is : ", len(data))
```

Adding element to list

```
# Appending element to list
print("Initial List : ", users)

# Adding to last
users.append('Anya')
print(users)
```

```
# updating using operators
users += ['Forger', 'L']
print(users)

# extend()
users.extend(['Robert', 'Albert'])
print(users)

# Adding list to list
emptylist.extend(data)
print(emptylist)
```

Insert method

```
# Insert method
users.insert(0, 'Bob')
print(users)
```

Using Slicing

```
users[3:3] = ['Anath', 'Sid']          # the elements will be moved and
space is made to new ones
print(users)

users[1:4] = ['Romeo', 'Juilet']    # 1, 2 index elements are replaced
by new ones
print(users)
```

Deleting element in List

```
# Remove
anime = ['Naruto', 'Boruto', 'Shinigami', 'Light', 'Zenitsu']
anime.remove('Boruto')
print(anime)

# pop
anime.pop()
print(anime)

# del
del anime[0]
print(anime)
```

```
# Clear all elements in list
anime.clear()          # list will be empty
print(anime)
```

Reversing a list

```
# Reversing a list
reverse_list = anime.reverse()
print(anime)
```

Sorting

```
# sort
anime.sort()             # ascending order
print(anime)

#descending order
anime.sort(reverse=True)
print(anime)
```

Types of copying list

```
list2 = users.copy()
mylist = list(anime)
list1 = anime[:]
```

Tuples

Tuples are similar to list, but tuples are immutable.

Creating tuples

```
tuple1 = (1, 23, 432, 'Vishnu')
tuple2 = tuple((1, 2, 2, 2, 3 , 54))
```

Creating a tuple using list

```
newlist = tuple(tuple1)
newtuple = tuple(newlist)
print(newtuple)
```

Unpacking tuples

```
(one, *two, three) = newtuple       # here * means remaining elements
print(one)
print(two)
print(three)
```

Frequency of element

```
print(tuple2.count(2))
```

Dictionaries

Dictionaries is a data structure used to store key-value pair.

Create dictionary

```
family = {
    "tiger" : "cat",
    "fox" : "dog",
    "racoon" : "dog"
}

# using constructor
family2 = dict(tiger = "cat", fox = "dog")

print(type(family2))
print(len(family))
```

Access items

```
# Access items
print(family["fox"])
print(family.get("tiger"))
```

List all keys, values & items

```
# list all keys, values & items
print(family.keys())
print(family.values())
print(family.items())
```

Verify if key exists

```
# verify if key exists
print("tiger" in family)
print("cat" in family)
```

Change values

```
# Change values
family["tiger"] = "dog"               # update key's value
```

Add new pair

```
family.update({"Lion": "Cat"})        # add new pair
print(family)
```

Change values

```
family["tiger"] = "dog"               # update key's value
family.update({"Lion": "Cat"})        # add new element
```

Remove or clear

```
# remove items
family.pop("tiger")
print(family)

# remove last added pair
family.popitem()
print(family)

# delete and clear
#clear
family2.clear()
print(family2)

#delete
del family2
```

Copy dictionary

```
# Creating a copy
newdictionary = family.copy()
newdictionary1 = dict(family)
```

```
print(newdictionary)
print(newdictionary1)
```

Nested Dictionaries

```
# Nested Dictionaries
type1 = {
    "name" : "plant",
    "instrument" : "vocals"
}

type2 = {
    "name" : "page",
    "instrument" : "guitar"
}

band = {
    "member1" : type1,
    "member2" : type2
}

print("")
print(band)
print(band["member1"])
print(band["member1"]["name"])
```

Sets

Sets does not store duplicate elements.
It is mutable

creating a set

```
# creating a set
nums = {1, 2, 3, 4}
nums2 = set((1, 3, 4, 5, 2 , 1))
print(nums)
print(nums2)
```

```
# True is 1 and Flase is 0
nums = {1, True, 2, False, 3, 4, 0}
print(nums)
```

check if value is in set

```
#check if value is in set
print(2 in nums)
```

Add elements from one set to another

```
# Add elements from one set to another
nums3 = {5, 6 ,7}
nums.update(nums3)
print(nums)
```

merge two sets

```
# merge two sets
one = {1, 2, 3}
two = {4, 5, 6}

newSet = one.union(two)
print(newSet)
```

Intersection

```
# Intersection
one = {1, 2, 3}
two = {4, 2, 1}
newSet = one.intersection(two)
print(newSet)
# or
one = {1, 2, 3}
two = {4, 2, 1}
one.intersection_update(two)
print(one)
```

difference - keep everything except the duplicates

```
# difference - keep everything except the duplicates
one = {1, 2, 3}
two = {4, 2, 1}

one.symmetric_difference_update(two)
print(one)
```

Closures :

A closure is a function object that remembers values in enclosing scopes even if they are not present in memory.

```
# A Closure is a function object that remembers values in enclosing
scopes even if they are not present in memory.

def outerFunction(num1):
    def innerFunction(num2):
        return num1 * num2
    return innerFunction

# first pass the value to outer function
times3 = outerFunction(3)
times5 = outerFunction(5)

# passing the
print(times3(9))
print(times5(times3(3)))
```

```
def parent(person) :
    coins = 3
    def child():
        nonlocal coins
        coins -= 1

        if coins > 1:
            print("\n" + person + " has " + str(coins) + " coins
left.")
        elif coins == 1:
            print("\n" + person + " has " + str(coins) + " coin left.")
        else:
            print("\n" + person + " is out of coins.")

    return child

tommy = parent("Tommy")
jenny = parent("Jenny")

tommy()
```

```
tommy()
jenny()
tommy()
```

Output :
Tommy has 2 coins left.

Tommy has 1 coin left.

Jenny has 2 coins left.

Tommy is out of coins.

Subchapter 1.5: File Handling and Exceptions

Exceptions:

try:
    statement
except Exception as varname:
    Statement

```
class custom(Exception):
    pass

x = 6

try:
    # print(z)
    # print(x/0)
    # if not type(x) is str:
    #     raise TypeError("only strings are allowed")
    raise custom("This a custom error")
except NameError:
    print("We have a name error")
except ZeroDivisionError:
    print("You cannot divide with zero")
# if exception occurs in try block, this will be executed
except Exception as error:
    print(error)
finally:
    print("This block will be printed even if there are no errors")
```

File handling:

```
f = open("names.txt")
# print(f.read())
# # print(f.readline())

# for line in f:
#     print(line)

# f.close()

try:
    f = open("names.txt")
    print(f.read())
except:
    print("The file you want to read doesn't exist")
finally:
    f.close()

# append
f = open("names.txt", 'a')
f.write("Neil\n")
f.close()

# overwrite
f = open("names.txt", 'w')
f.write("The content was overwritten")
f.close()

f = open("names.txt")
print(f.read())
f.close()

import os
if not os.path.exists("dave.txt"):
    f = open("dave.txt", "x")
    f.close()

if os.path.exists("dave.txt"):
    os.remove("dave.txt")
else:
    print("The file dosenot exit to remove")
```

```
# with has built-in implicit exception handling
# close() will be automatically called
with open ("names.txt") as f:
    content = f.read()
```

Lamda

lambda is a keyword in Python for defining the anonymous function.

from functools import reduce
def squared(num): return num * num
# lambda num : num * num
print(squared(2))

def sum_total(a, b): return a + b
# lambda a, b : a + b
print(sum_total(10, 8))

########################

def funcBuilder(x):
    return lambda num: num + x
add_ten = funcBuilder(10)
add_twenty = funcBuilder(20)

print(add_ten(7))
print(add_twenty(7))

########################

map()

The map() function executes a specified function for each item in an iterable. The item is sent to the function as a parameter.

numbers = [3, 7, 12, 18, 20, 21]
squared_nums = map(lambda num: num * num, numbers)
print(list(squared_nums))

################################

filter()

The filter() function returns an iterator where the items are filtered through a function to test if the item is accepted or not by Syntaxs.

```
odd_nums = filter(lambda num: num % 2 != 0, numbers)
print(list(odd_nums))
```

reduce()

In Python, reduce() is a built-in function that applies a given function to the elements of an iterable, reducing them to a single value.

reduce() is a function in Python that combines elements of an iterable using a specified function. It takes the form:

```
numbers = [1, 2, 3, 4, 5, 1]
total = reduce(lambda acc, curr: acc + curr, numbers, 10)
print(total)
print(sum(numbers, 10))

names = ['Dave Gray', 'Sara Ito', 'John Jacob Jingleheimerschmidt']
char_count = reduce(lambda acc, curr: acc + len(curr), names, 0)
print(char_count)
```

Pip

- py -m pip install requests
- py -m pip list
- py -m pip install requests 2.31.0
- python.exe -m pip install --upgrade pip
- pip show requests

Virtual Environments

- **Create** - py -m venv .venv
- **Activate -** .\.venv\Scripts\Activate
- **Deactivate -** deactivate
- py -m pip install python-dotenv

Not to include in git

py -m pip freeze > requirements.txt
Make a file - .gitignore

Oops :

1 Object - a variable or a method with attributes
2 Class - a definition of an object's attributes and methods
3 Polymorphism - having one common name and many forms
4 Encapsulation - hiding attributes or methods as needed
5 Inheritance - makes new object from parent attributes and methods

```
class Dog:
    # class attribute
    attr1 = "mammal"

    # Instance attribute
    def __init__(self, name):
        self.name = name

# Driver code
# Object instantiation
Rodger = Dog("Rodger")
Tommy = Dog("Tommy")

# Accessing class attributes
print("Rodger is a {}".format(Rodger.__class__.attr1))
print("Tommy is also a {}".format(Tommy.__class__.attr1))

# Accessing instance attributes
print("My name is {}".format(Rodger.name))
print("My name is {}".format(Tommy.name))
```

```
class A:
    def a(self,st):
        return 1

    def b(self,s1,s2) :
        d= self.a(7)
        return d
```

```
obj=A()
print(obj.b(1,2))
```