Contents

Intermediate 2

Object Oriented Programming

- It's a programming paradigm
- In OOP we have classes and objects

4 Pillars of OOP

- **Encapsulation**: Putting things together
- **Abstraction**: Hiding the irrelevant features
- **Inheritance**: objects of one class can inherit the features of another
- **Polymorphism**: Same entity but different behavior

Class

-

• Class is a blueprint that defines the methods and properties of an object.
 	
• `self` argument is the reference to the object itself
	`class Dog:`
	`  kind = "canine"`
	`  def __init__(self, name):`
	`  self.name = name`

• **Private properties**: only accessible inside class definition.
 	
• append '__' as a prefix in property name to make it private.
 	
Note: Nothing can be completely private in python.
	`class BankAccount:`
	`  def __init__(self, balance):`
	`  self.__balance = balance`

Constructor

- It's the first function that'll be called whenever an object is created.
- Python doesn't provide direct access to constructors

Note: __init__ is not the constructor, it's initializing function

Dunder/Magic Methods

- methods in class can be modified/overloaded to change the default behavior of that object.
- some useful magic methods

modifying print() behavior	__str__(self)
modifying call behavior	__call__(self)
addition behavior	__add__(self, other)
less than operator behavior	__lt__(self, other)

Inheritance

objects of one class can inherit the features of another	`class Parent:`
	` def __init__(self):`
	`  print("parent class")`
	
	`class child(Parent):`
	` def __init__(self):`
	`  print("child class")`
	
	`# inheriting from Parent class`

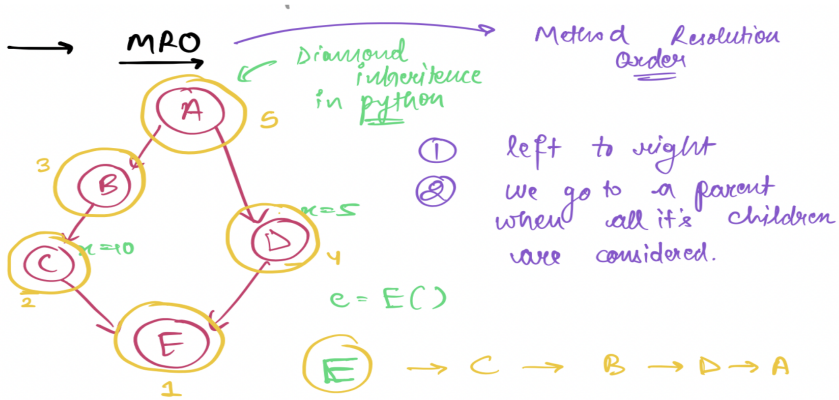
Multiple inheritance, is when inheriting from multiple classes	`class A:`
	`  def __init__(self, a):`
	`  self.a = a`
	
	`class B:`
	`  def __init__(self, b):`
	`  self.b = b`
	
	`# C inherits from both A and B`
	`class C(A, B):`
	`  def __init__(self, a, b, c):`
	`  A.__init__(self, a)`
	`  B.__init__(self, b)`
	`  self.c = c`

super() method is used to call methods of parent class	`class child(Parent):`
	` def __init__(self):`
	`  super().__init__()`

	`print("child class")`

Method Resolution Order

- It is the order in which a method is searched for in a classes hierarchy
- Rules:
  - Left to right
  - visit parent when all its children are already visited



Lambda Functions

short syntax to write functions in Python	`lambda arg1, arg2 : return "output"`

Higher Order functions

A function that returns function	`def outer_fn(n):`
	`  def inner_fn(x):`
	`  return x+2`
	`  return inner_fn`

Decorators

High order functions that take another function as input and add the extra behavior in along with the functionality of	`def pretty(func):` `  def inner():`

passed function	``` print("-"*50)
	        func()
	        print("-"*50)
	    return inner```

Using decorator on other function	```@pretty # usage
	def foo():
	    print("WHATTTT?")```

Principles of functional programming

- Data should be separated from mutations
- treat variable as immutable
- treat functions as FCC

Maps

takes multiple iterables and perform some function on them and returns output map	map(function_to_perform, *iterables) # example: ```>> list(map(lambda x: x**2, [1,2,3])) # [1,4,9]```

Filter

Filter out elements from a list on the basis of some condition	```>> list(filter(lambda x: x%2 == 0, [1,2,3,4,5])) # [2,4]```

Reduce

reduces an iterable into single value	```from functools import reduce```

	`reduce(lambda x, y: x + y, [1,2,3])` `# 6`

Zip

returns an iterator that will aggregate elements from two or more iterables.	`a = [1,2,3]`
	`b = ["a", "b", "c", "d", "e"]`
	`>> list(zip(b,a))`
	`# [('a', 1), ('b', 2), ('c', 3)]`

Args and Kwargs

args are variable size positional arguments stored inside tuple	`def sum_number(x, y, *args):`
	`  result = x + y`
	`  if args:`
	`  result += sum(args)`
	`  return result`

krwargs are variable size keyword arguments stored inside dictionary	`def create_person(name, age, gender, **extra_info):`
	`  Person = {`
	`  "name": name,`
	`  "age": age,`
	`  "gender": gender`
	`  }`
	
	`  if extra_info:`
	`  Person.update(extra_info)`
	`  return Person`

Order of passing arguments: *Positional -> Args -> Keyworded -> Kwargs*

File Handling

- use secondary memory, to keep data even when program terminated

- Everything in memory is a sequence of bytes or byte array.

File Access Modes

Read only	r
read binary	rb
Read and Write	r+
Write Only	w
write binary	wb
Write and Read	w+
Append Only	a
Append and Read	a+

Working with Files

Opening a file	`file = open("sample.txt", "r")`
Closing a file	`file.close()`

Writing to a file	`file = open("sample.txt", "w+")`
	`file.writelines(["1\n", "2\n"]) # write from list`
	`file.write("hellow world") # write from string`
	`file.close()`

Reading from a file	`file = open("sample.txt", "r+")`
	`file.read() # read everything as string`
	`file.readline()# read line by line`
	`file.readlines() # read all lines as list`
	`file.close()`

Reading large files	`file = open("sample2.txt", "r+")`
	`buffer = file.readline()`
	`while buffer: # n lines = 1 block of memory`
	`  print(buffer, end = "")`

	`buffer = file.readline()` `file.close()`
moving reading/writing cursor	`file.seek(3)# moving 3 characters ahead`

smart way of working with files.
“with” statement simplifies exception handling by encapsulating common preparation and cleanup tasks."
	`with open("sample3.txt", "r+") as file:`
	`  print(file.read(5))`
	`  file.seek(0)`
	`  print(file.read())`

Modules

- collection of python files that contains re-usable functions, which can be imported to other files.
- Collection of such modules is known as package

Multiple import statements

importing entire module	`>> import math` `>> math.sqrt(10)`
Importing using different Alias names	`>> import math as m` `>> m.sqrt(10)`
importing only required functions/classes	`>> from math import sqrt` `>> sqrt(10)`
import everything within module	`>> from math import *` `>> sqrt(10)`

Exception Handling

- Python known error with cause are known as exceptions
- program gets terminated as an exception occurs.

Try-Except

mechanism of handle exceptions	`try:`
	` # code that may cause exception`
	`except:`
	` # what to do when exception occurs`

handling specific exceptions	`try:`
	`  return a / 0`
	`  print(5 + 4)`
	`  # any amount of code`
	`except ZeroDivisionError:`
	`  print("WHY ARE YOU DIVIDING BY ZERO?")`

`finally` block runs even if an exception occurs or not and free all the allocated resources if any.	`try:`
	`  print("I am trying!")`
	`  1/0`
	`except:`
	`  print("Except")`
	`finally:`
	`  print("FINALLLYYY!!")`

Custom Exceptions

raising custom exceptions	`raise Exception("custom exception")`
creating custom exceptions: Just inherit the base class of `Exception` and add req functionalities	`class MyCustomException(Exception):` `  pass`

Types of functions

Linear	ax + by + c = 0
Exponential	e^x, 2^x, etc
Logarithmic	log(x)

Continuous function

- Function that changes without any jump. eg: tan(x) is discontinuous

Limits

(i)	The left hand limit (LHL) at $x = a$: $\lim_{x \to a^-} f(x)$	Describes the behaviour of $f(x)$ to the immediate left of $x = a$
(ii)	The right hand limit (RHL) at $x = a$: $\lim_{x \to a^+} f(x)$	Describes the behaviour of $f(x)$ to the immediate right of $x = a$
(iii)	The value of $f(x)$ at $x = a$: $f(a)$	Gives the precise value that $f(x)$ takes at $x = a$

Fig - 10

Test of continuity: *If the limit exist for all input in the domain of function, then it is continuous.*

Tangents

- a straight line or plane that touches a curve or curved surface at a point.
- Finding tangents:
    - find slope
    - find point of contact

Derivatives

- Derivative of a function is also a function
- $\frac{d}{dx}f(x) = f`(x)$
- Some standard formulas
-

Quadratic	x^2	2x
Linear	mx + c	m
Exponential	e^x	e^x
Logarithmic	log(x)	1/x

Rules for calculating derivative of functions

monomial derivative	x^n	nx^{n-1}
linearity rule	af(x) + bf(y)	af'(x) + bf'(y)
Product rule	f(x)g(x)	f(x)g'(x) + f'(x)g(x)
Chain rule	f(g(x))	f'(g(x))g'(x)