**Python 5**

**Excercise Answers**

**1. Learn and show your understanding of map, reduce, filter**

from functools import reduce

nums = range(1, 20)

#filter()

even = list(filter(lambda x : x%2 == 0, nums))

# odd = filter(lambda x : x%2 != 0, nums)

print(even)

# print(list(odd))

def func(n):

return n+2

doub = list(map(func, even))

#map()

doub\_lam = list(map(lambda x : x+2, even))

print(doub\_lam)

#reduce()

cummulative = reduce(lambda a,b : a+b, doub\_lam)

print(cummulative)

**2. Python functions are said to be "first class objects". What is the meaning of this? Show various cases where this is true**

functions are allowed to paased as parameter in python

we can add functions in iterables

Example

def say\_hello(name):

return f"Hello {name}"

def be\_awesome(name):

return f"Yo {name}, together we are the awesomest!"

mylist = [say\_hello, be\_awesome]

# print(mylist[0]("Subhashini"))

def greet\_subha(greeter\_func):

return greeter\_func("Subhashini")

print(greet\_subha(say\_hello))

print(greet\_subha(be\_awesome))

**3. Write a lambda function and how is it different from a regular function?**

In Python, an anonymous function is a [function](https://www.programiz.com/python-programming/function) that is defined without a name. While normal functions are defined using the def keyword in Python, anonymous functions are defined using the lambda keyword. Hence, anonymous functions are also called lambda functions.

### Syntax of Lambda Function in python

Obj = lambda arguments: expression

Lambda functions can have any number of arguments but only one expression.

Example

double = lambda x: x \* 2

def double(x):

return x \* 2

4. Read about "higher order functions" and show a simple implementation to explain their usage

A function that is having another function as an argument or a function that returns another function as a return in the output is called the High order function.

def shout(text):

return text.upper()

def whisper(text):

return text.lower()

def greeting(func):

greet = func("Hello all...")

return greet

print(greeting(shout))

print(greeting(whisper))

def create\_adder(x):

def adder(y):

return x+y

return adder

add = create\_adder(10)

print(add(11))

**5. Write a simple Python decorator that would print the time taken by any function**

import functools

# from symbol import decorator

import time

def decorator\_timer(func):

""" Print the run time of the decorated function"""

@functools.wraps(func)

def wrapper\_timer(\*args, \*\*kwargs):

start = time.time()

value = func(\*args, \*\*kwargs)

end = time.time()

run\_time = end - start

print(f"Finished in {run\_time:.4f} secs")

return value

return wrapper\_timer

@decorator\_timer

def waste\_some\_time(num\_times):

for \_ in range(num\_times):

sum([i\*\*2 for i in range(500)])

waste\_some\_time(2)

**6. How does Generics work in Python? Learn and show your understanding by implementing a snippet that uses Generics**

reference: https://mypy.readthedocs.io/en/stable/generics.html

The aim of generics are to:

* Allow functions, methods and classes to work with arguments of any type whilst maintaining the information on the relationships between things, such as arguments and return values.
* Better define how types can mix

## **Defining generic classes**

The built-in collection classes are generic classes. Generic types have one or more type parameters, which can be arbitrary types. For example, dict[int, str] has the type parameters int and str, and list[int] has a type parameter int.

Programs can also define new generic classes. Here is a very simple generic class that represents a stack:

from typing import TypeVar, Generic

T = TypeVar('T')

class Stack(Generic[T]):

def \_\_init\_\_(self) -> None:

# Create an empty list with items of type T

self.items: list[T] = []

def push(self, item: T) -> None:

self.items.append(item)

def pop(self) -> T:

return self.items.pop()

def empty(self) -> bool:

return not self.items

The Stack class can be used to represent a stack of any type: Stack[int], Stack[tuple[int, str]], etc.

## **Generic functions**

Generic type variables can also be used to define generic functions:

from typing import TypeVar, Sequence

T = TypeVar('T') # Declare type variable

def first(seq: Sequence[T]) -> T: # Generic function

return seq[0]

As with generic classes, the type variable can be replaced with any type. That means first can be used with any sequence type, and the return type is derived from the sequence item type. For example:

# Assume first defined as above.

s = first('foo') # s has type str.

n = first([1, 2, 3]) # n has type int.

Note also that a single definition of a type variable (such as T above) can be used in multiple generic functions or classes. In this example we use the same type variable in two generic functions:

from typing import TypeVar, Sequence

T = TypeVar('T') # Declare type variable

def first(seq: Sequence[T]) -> T:

return seq[0]

def last(seq: Sequence[T]) -> T:

return seq[-1]

**7. Implement an example that uses overloading in Python. How is overloading different in Python from other languages and why?**

**Polymorphism**

the same [function](https://www.edureka.co/blog/python-functions) name can be used for different types. This makes programming more intuitive and easier.

**Achived through**

* Duck typing
* Method overloading
* Operator overloading
* Method overriding

**Duck typing**

The idea behind this principle is that the code itself does not care about whether an object is a duck, but instead it does only care about whether it quacks.

generaly python is a dynamic programming language unlike other languages we cant define the type of the variable

>>>a = 5

>>>a = 10.5

>>>a = “subhashini”

class Pycharm:

def execute():

print(“compile”)

print(“execute”)

class MyEditor:

def execute():

print(“spell check”)

print(“convention check”)

print(“compile”)

print(“execute”)

class Laptop:

def code(self, ide):

ide.execute()

ide1 = Pycharm()

ide2 = Myeditor()

lap1 = Laptop(ide1)

lap1 = Laptop(ide2)

here python not concern about ide1, ide2 are which object, which type it is.

**Operator Overloading**

same method name with num of args and type of args are different

in python >>>a + b which calls the magic function \_\_add\_\_() for each and every class(int.\_\_add\_\_(), float.\_\_add\_\_(), str.\_\_add\_\_())

there is no \_\_add\_\_() for user defined classes.

So overload the ‘+’ operator for adding two two objects of user defined classes.

Example

class Student:

def \_\_init\_\_(self, m1, m2):

self.m1 = m1

self.m2 = m2

def \_\_add\_\_(self, other):

m1 = self.m1 + other.m1

m2 = self.m2 + other.m2

s3 = Student(m1, m2)

return s3

s1 = Student(55, 56)

s2 = Student(62, 67)

**Method overloading**

it is not supported by python.

class Addition:

# first sum for 2 params

def my\_sum(self, a, b):

return a + b

# second overloaded sum for 3 params

def my\_sum(self, a, b, c):

return a + b + c

obj = Addition()

print(obj.my\_sum(3, 4))

print(obj.my\_sum(3, 4, 5))

Traceback (most recent call last):

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

TypeError: my\_sum() missing 1 required positional argument: 'c'

12

In python, unlike other languages, you cannot perform method overloading by using the same method name. Why?

Everything is an object in python, classes, and even methods. Say you have an object Addition, which is a class (everything in python is an object, so the class Addition is an object too). It has an attribute called - my\_sum. It's the only attribute that the class Addition can have with that name.

The second function replaces the first one, and that is why the only valid function that you can call is my\_sum that takes 3 parameters as input.

Achive method overloading by using None

#method overloading

class Addition:

def my\_sum(self, a = None, b = None, c = None):

s = 0

if a != None and b != None and c != None:

s = a + b + c

elif a != None and b != None:

s = a + b

return s

obj = Addition()

print(obj.my\_sum(10, 2))

print(obj.my\_sum(10,11,22))

**Method Overriding**

used in inheritance. While both the parent and child using the same methods

#method overriding

class A:

def show(self):

print("this is parent")

class B(A):

def show(self):

print("am child") #overrided

ob = B()

ob.show()

ob1 = A()

ob1.show()

we can also override build in magic methods

**8. What is the difference between a Python script, package and a module?**

A script is a Python file that’s intended to be run directly. When you run it, it should do something. This means that scripts will often contain code written outside the scope of any classes or functions.

A module is a Python file that’s intended to be imported into scripts or other modules. It often defines members like classes, functions, and variables intended to be used in other files that import it.

A package is a collection of related modules that work together to provide certain functionality. These modules are contained within a folder and can be imported just like any other modules. This folder will often contain a special \_\_init\_\_ file that tells Python it’s a package, potentially containing more modules nested within subfolders

A library is an umbrella term that loosely means “a bundle of code.” These can have tens or even hundreds of individual modules that can provide a wide range of functionality. [Matplotlib](https://realpython.com/python-matplotlib-guide/) is a plotting library. The Python Standard Library contains hundreds of modules for performing common tasks, like [sending emails](https://realpython.com/python-send-email/) or reading JSON data. What’s special about the Standard Library is that it comes bundled with your installation of Python, so you can use its modules without having to download them from anywhere.