Lab -5

Programing in Python

Instructor : AALWAHAB DHULFIQAR

Advisor : Dr. Tejfel Mate

python

What you will learn:

Classes.
OOP objects.
Instance variables.
Class variables.
OOP Methods.

Use your brain to find the answers

```
try:
    print(1/0)
except ZeroDivisionError:
    print("zero")
except ArithmeticError:
    print("arith")
except:
    print("some")
```

```
try:
    print(1/0)
except ArithmeticError:
    print("arith")
except ZeroDivisionError:
    print("zero")
except:
    print("some")
```

Which of the exceptions will you use to protect your code from being interrupted through the use of the keyboard?

```
def foo(x):
    assert x
    return 1/x
try:
    print(foo(0))
except ZeroDivisionError:
    print("zero")
except:
    print("some")
```

Which of the exceptions will be raised through the following unsuccessful evaluation?

What is the name of the most general of all Python exceptions?

Can you name one of your classes just "class"?

Lab: 5.1 (Exceptions)

Scenario

Your task is to write a function able to input integer values and to check if they are within a specified range.

The function should:

- accept three arguments: a prompt, a low acceptable limit, and a high acceptable limit;
- if the user enters a string that is not an integer value, the function should emit the message Error: wrong input, and ask the user to input the value again;
- if the user enters a number which falls outside the specified range, the function should emit the message Error: the value is not within permitted range (min..max) and ask the user to input the value again;
- if the input value is valid, return it as a result.

```
def read_int(prompt, min, max):
    #
    # Write your code here.
    #


v = read_int("Enter a number from -10 to 10: ", -10, 10)

print("The number is:", v)
```

```
Enter a number from -10 to 10: 100
Error: the value is not within permitted range (-10..10)
Enter a number from -10 to 10: asd
Error: wrong input
Enter number from -10 to 10: 1
The number is: 1
```

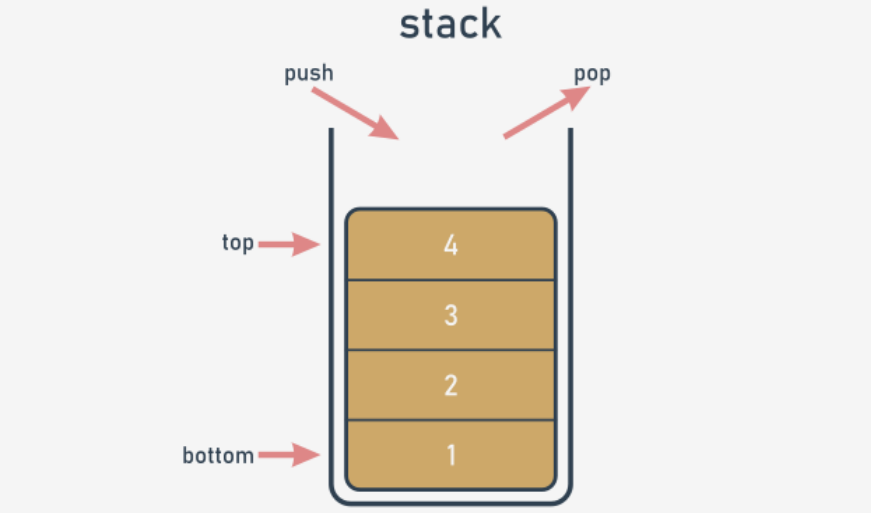
Your first class

```
class TheSimplestClass:
    pass
```

Your first object

```
my_first_object = TheSimplestClass()
```

What is a stack?



The stack - the procedural approach

```
stack = []


def push(val):
    stack.append(val)


def pop():
    val = stack[-1]
    del stack[-1]
    return val


push(3)
push(2)
push(1)

print(pop())
print(pop())
print(pop())
```

The stack - the procedural approach

1. the essential variable (the stack list) is highly vulnerable
2. it may also happen that one day you need more than one stack
3. it may also happen that you need not only push and pop functions

vs. the object-oriented approach

1. the ability to hide (protect) selected values against unauthorized access is called encapsulation
2. when you have a class implementing all the needed stack behaviors, you can produce as many stacks as you want
3. the ability to enrich the stack with new functions comes from inheritance

The stack - the object approach

```
class Stack:  # Defining the Stack class.
    def __init__(self):  # Defining the constructor function.
        print("Hi!")
stack_object = Stack()  # Instantiating the object.
```

add just one property to the new object

```
class Stack:
    def __init__(self):
        self.stack_list = []

stack_object = Stack()
print(len(stack_object.stack_list))
```

When any class component has a name starting with two underscores (__), it becomes private - this means that it can be accessed only from within the class.

You cannot see it from the outside world. This is how Python implements the encapsulation concept.

Run the program to test our assumptions - an AttributeError exception should be raised.

```
class Stack:
    def __init__(self):
        self.__stack_list = []

stack_object = Stack()
print(len(stack_object.__stack_list))
```

The object approach: a stack from scratch

```
class Stack:
    def __init__(self):
        self.__stack_list = []

    def push(self, val):
        self.__stack_list.append(val)

    def pop(self):
        val = self.__stack_list[-1]
        del self.__stack_list[-1]
        return val

stack_object = Stack()

stack_object.push(3)
stack_object.push(2)
stack_object.push(1)

print(stack_object.pop())
print(stack_object.pop())
print(stack_object.pop())
```

The stack - the object approach

```
class Stack:
    def __init__(self):
        self.__stack_list = []

    def push(self, val):
        self.__stack_list.append(val)

    def pop(self):
        val = self.__stack_list[-1]
        del self.__stack_list[-1]
        return val


little_stack = Stack()
another_stack = Stack()
funny_stack = Stack()

little_stack.push(1)
another_stack.push(little_stack.pop() + 1)
funny_stack.push(another_stack.pop() - 2)

print(funny_stack.pop())
```

```
class Stack:
    def __init__(self):
        self.__stack_list = []

    def push(self, val):
        self.__stack_list.append(val)

    def pop(self):
        val = self.__stack_list[-1]
        del self.__stack_list[-1]
        return val


class AddingStack(Stack):
    def __init__(self):
        Stack.__init__(self)
        self.__sum = 0
```

The object approach: a stack from scratch

```
class Stack:
    def __init__(self):
        self.__stack_list = []

    def push(self, val):

self.__stack_list.append(val)

    def pop(self):
        val = self.__stack_list[-1]
        del self.__stack_list[-1]
        return val
```

```
class AddingStack(Stack):
    def __init__(self):
        Stack.__init__(self)
        self.__sum = 0

    def get_sum(self):
        return self.__sum

    def push(self, val):
        self.__sum += val
        Stack.push(self, val)

    def pop(self):
        val = Stack.pop(self)
        self.__sum -= val
        return val
```

```
stack_object = AddingStack()

for i in range(5):
    stack_object.push(i)
print(stack_object.get_sum())

for i in range(5):
    print(stack_object.pop())
```

Lab: 5.2 (OOP)

Objectives

improve the student's skills in defining classes;

using existing classes to create new classes equipped with new functionalities.

Scenario

We've showed you recently how to extend Stack possibilities by defining a new class (i.e., a subclass) which retains all inherited traits and adds some new ones.

Your task is to extend the Stack class behavior in such a way so that the class is able to count all the elements that are pushed and popped (we assume that counting pops is enough). Use the Stack class we've provided in the editor.

Follow the hints:

introduce a property designed to count pop operations and name it in a way which guarantees hiding it;

initialize it to zero inside the constructor;

provide a method which returns the value currently assigned to the counter (name it get_counter()).

```
class Stack:
    def __init__(self):
        self.__stk = []

    def push(self, val):
        self.__stk.append(val)

    def pop(self):
        val = self.__stk[-1]
        del self.__stk[-1]
        return val


class CountingStack(Stack):
    def __init__(self):
    #
    # Fill the constructor with appropriate actions.
    #

    def get_counter(self):
    #
    # Present the counter's current value to the world.
    #

    def pop(self):
    #
    # Do pop and update the counter.
    #


stk = CountingStack()
for i in range(100):
    stk.push(i)
    stk.pop()
print(stk.get_counter())
```

your code outputs 100.

Lab: 5.3 (OOP)

Scenario

As you already know, a stack is a data structure realizing the so-called LIFO (Last In - First Out) model. It's easy and you've already grown perfectly accustomed to it.

Let's taste something new now. A queue is a data model characterized by the term FIFO: First In - Fist Out. Note: a regular queue (line) you know from shops or post offices works exactly in the same way - a customer who came first is served first too.

Your task is to implement the Queue class with two basic operations:

- put(element), which puts an element at end of the queue;
- get(), which takes an element from the front of the queue and returns it as the result (the queue cannot be empty to successfully perform it.)

Follow the hints:

- use a list as your storage (just like we did in stack)
- put() should append elements to the beginning of the list, while get() should remove the elements from the list's end;
- define a new exception named QueueError (choose an exception to derive it from) and raise it when get() tries to operate on an empty list.

```
class QueueError(???):  # Choose base class for the new exception.
    #
    #  Write code here
    #


class Queue:
    def __init__(self):
        #
        # Write code here
        #

    def put(self, elem):
        #
        # Write code here
        #

    def get(self):
        #
        # Write code here
        #


que = Queue()
que.put(1)
que.put("dog")
que.put(False)
try:
    for i in range(4):
        print(que.get())
except:
    print("Queue error")
```

```
1
dog
False
Queue error
```

Instance variables

Variables that they are closely connected to the objects (which are class instances), not to the classes themselves

```
class ExampleClass:
    def __init__(self, val = 1):
        self.first = val

    def set_second(self, val):
        self.second = val


example_object_1 = ExampleClass()
example_object_2 = ExampleClass(2)

example_object_2.set_second(3)

example_object_3 = ExampleClass(4)
example_object_3.third = 5

print(example_object_1.__dict__)
print(example_object_2.__dict__)
print(example_object_3.__dict__)
```

Python objects, when created, are gifted with a small set of predefined properties and methods. Each object has got them, whether you want them or not. One of them is a variable named __dict__ (it's a dictionary).

```
class ExampleClass:
    def __init__(self, val = 1):
        self.__first = val

    def set_second(self, val = 2):
        self.__second = val


example_object_1 = ExampleClass()
example_object_2 = ExampleClass(2)
```

```
example_object_2.set_second(3)

example_object_3 =
ExampleClass(4)
example_object_3.__third = 5


print(example_object_1.__dict__)
print(example_object_2.__dict__)
print(example_object_3.__dict__)
```

Class variables

```
class ExampleClass:
    counter = 0
    def __init__(self, val = 1):
        self.__first = val
        ExampleClass.counter += 1


example_object_1 = ExampleClass()
example_object_2 = ExampleClass(2)
example_object_3 = ExampleClass(4)

print(example_object_1.__dict__, example_object_1.counter)
print(example_object_2.__dict__, example_object_2.counter)
print(example_object_3.__dict__, example_object_3.counter)
```

```
class ExampleClass:
    varia = 1
    def __init__(self, val):
        ExampleClass.varia = val
print(ExampleClass.__dict__)
example_object = ExampleClass(2)
print(ExampleClass.__dict__)
print(example_object.__dict__)
```


```
class ExampleClass:
    __counter = 0
    def __init__(self, val = 1):
        self.__first = val
        ExampleClass.__counter += 1


example_object_1 = ExampleClass()
example_object_2 = ExampleClass(2)
example_object_3 = ExampleClass(4)

print(example_object_1.__dict__,
example_object_1._ExampleClass__counter)
print(example_object_2.__dict__,
example_object_2._ExampleClass__counter)
print(example_object_3.__dict__,
example_object_3._ExampleClass__counter)
```

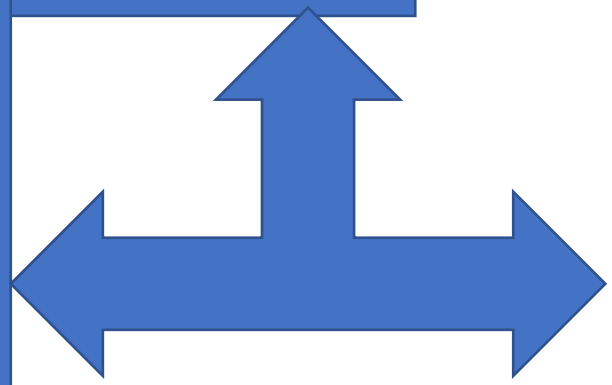
Checking an attribute's existence

```
class ExampleClass:
    def __init__(self, val):
        if val % 2 != 0:
            self.a = 1
        else:
            self.b = 1
example_object = ExampleClass(1)
print(example_object.a)
print(example_object.b)
```

```
1

Traceback (most recent call last):

  File "main.py", line 12, in <module>

    print(example_object.b)

AttributeError: 'ExampleClass' object has no
attribute 'b'
```

```
class ExampleClass:
    attr = 1
print(hasattr(ExampleClass, 'attr'))
print(hasattr(ExampleClass, 'prop'))
```

```
class ExampleClass:
    def __init__(self, val):
        if val % 2 != 0:
            self.a = 1
        else:
            self.b = 1
example_object = ExampleClass(1)
print(example_object.a)

try:
    print(example_object.b)
except AttributeError:
    pass
```



```
class ExampleClass:
    def __init__(self, val):
        if val % 2 != 0:
            self.a = 1
        else:
            self.b = 1


example_object = ExampleClass(1)
print(example_object.a)

if hasattr(example_object, 'b'):
    print(example_object.b)
```

Methods in detail

a method is a function embedded inside a class.

```
class Classy:
    def method(self):
        print("method")
obj = Classy()
obj.method()
```

```
class Classy:
    def method(self, par):
        print("method:", par)


obj = Classy()
obj.method(1)
obj.method(2)
obj.method(3)
```

```
class Classy:
    def other(self):
        print("other")

    def method(self):
        print("method")
        self.other()


obj = Classy()
obj.method()
```

```
method
other
```

__init__

If you name a method like this: __init__, it won't be a regular method - it will be a **constructor**

```
class Classy:
    def __init__(self, value):
        self.var = value
obj_1 = Classy("object")
print(obj_1.var)
```

Note that the constructor:

- cannot return a value
- cannot be invoked directly either from the object or from inside the class

```
class Classy:
    def visible(self):
        print("visible")

    def __hidden(self):
        print("hidden")
obj = Classy()
obj.visible()
try:
    obj.__hidden()
except:
    print("failed")
obj._Classy__hidden()
```

```
visible
failed
hidden
```

__dict__

Each Python class and each Python object is pre-equipped with a set of useful attributes which can be used to examine its capabilities.

```
class Classy:
    varia = 1
    def __init__(self):
        self.var = 2

    def method(self):
        pass

    def __hidden(self):
        pass


obj = Classy()

print(obj.__dict__)
print(Classy.__dict__)
```

__name__

The property contains the name of the class.

```
class Classy:
    pass
print(Classy.__name__)
obj = Classy()
print(type(obj).__name__)
```

__module__

it stores the name of the module which contains the definition of the class.

```
class Classy:
    pass
print(Classy.__module__)
obj = Classy()
print(obj.__module__)
```

__main__ is actually not a module, but the file currently being run.

if __name__ == "__main__"

__bases__

The tuple contains classes (not class names) which are direct superclasses for the class.

```
class SuperOne:
    pass

class SuperTwo:
    pass
class Sub(SuperOne, SuperTwo):
    pass
def printBases(cls):
    print('( ', end='')

    for x in cls.__bases__:
        print(x.__name__, end=' ')
    print(')')
printBases(SuperOne)
printBases(SuperTwo)
printBases(Sub)
```

```
( object )
( object )
( SuperOne SuperTwo )
```

Investigating classes

```
class MyClass:
    pass

obj = MyClass()
obj.a = 1
obj.b = 2
obj.i = 3
obj.ireal = 3.5
obj.integer = 4
obj.z = 5

def incIntsI(obj):
    for name in obj.__dict__.keys():
        if name.startswith('i'):
            val = getattr(obj, name)
            if isinstance(val, int):
                setattr(obj, name, val + 1)

print(obj.__dict__)
incIntsI(obj)
print(obj.__dict__)
```

Lab: 5.4 (OOP, methods and variables)

Scenario

Let's visit a very special place - a plane with the Cartesian coordinate system (you can learn more about this concept here: https://en.wikipedia.org/wiki/Cartesian_coordinate_system).

Each point located on the plane can be described as a pair of coordinates customarily called x and y. We expect that you are able to write a Python class which stores both coordinates as float numbers. Moreover, we want the objects of this class to evaluate the distances between any of the two points situated on the plane.

The task is rather easy if you employ the function named hypot() (available through the math module) which evaluates the length of the hypotenuse of a right triangle (more details here: https://en.wikipedia.org/wiki/Hypotenuse) and here: https://docs.python.org/3.7/library/math.html#trigonometric-functions.

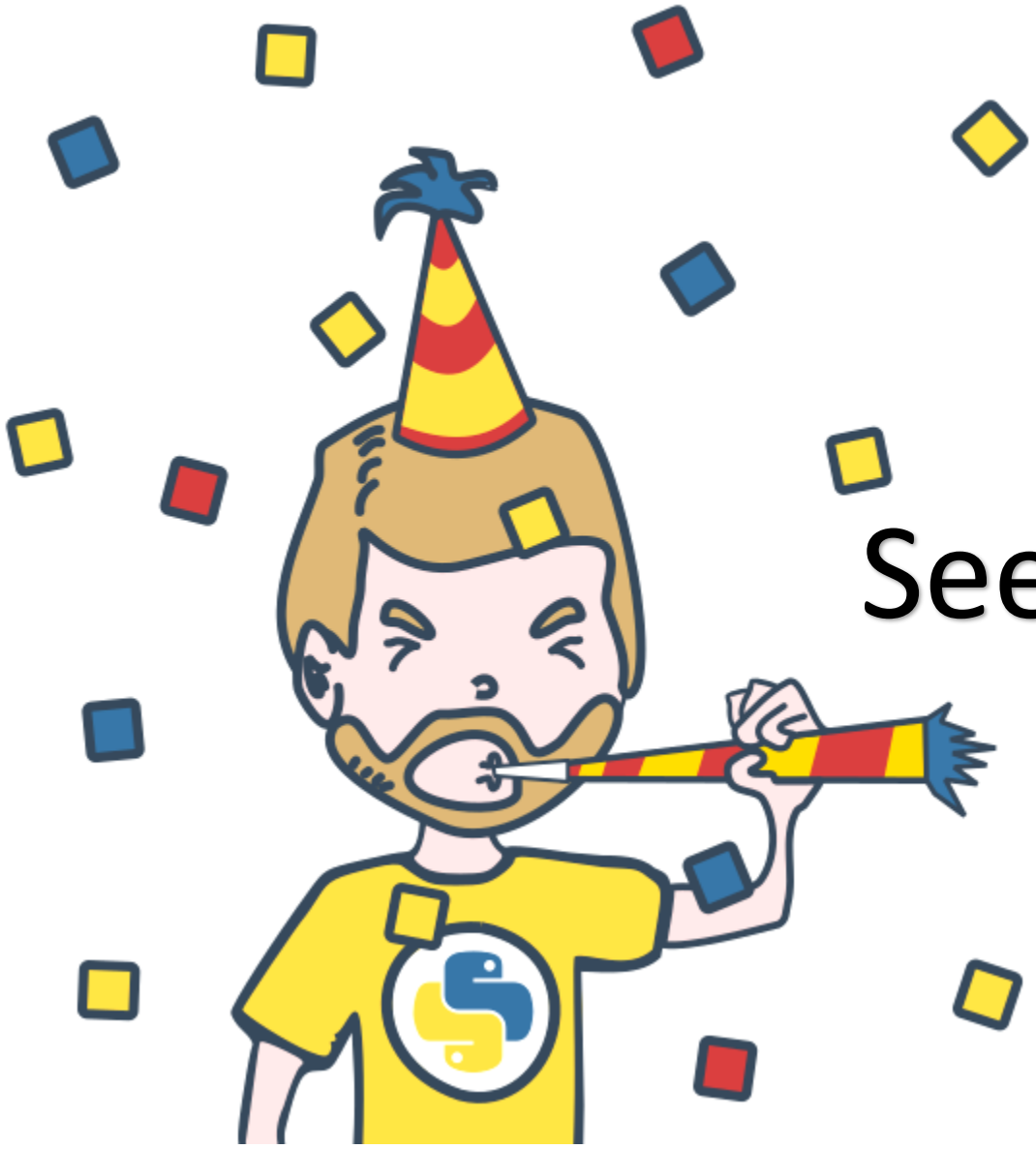
This is how we imagine the class:

- it's called Point;
- its constructor accepts two arguments (x and y respectively), both default to zero;
- all the properties should be private;
- the class contains two parameterless methods called getx() and gety(), which return each of the two coordinates (the coordinates are stored privately, so they cannot be accessed directly from within the object);
- the class provides a method called distance_from_xy(x,y), which calculates and returns the distance between the point stored inside the object and the other point given as a pair of floats;
- the class provides a method called distance_from_point(point), which calculates the distance (like the previous method), but the other point's location is given as another Point class object;

Complete the template we've provided in the editor and run your code and check whether your output looks the same as ours.

```
import math

class Point:
    def __init__(self, x=0.0, y=0.0):
        #
        # Write code here
        #

    def getx(self):
        #
        # Write code here
        #

    def gety(self):
        #
        # Write code here
        #

    def distance_from_xy(self, x, y):
        #
        # Write code here
        #

    def distance_from_point(self, point):
        #
        # Write code here
        #
point1 = Point(0, 0)
point2 = Point(1, 1)
print(point1.distance_from_point(point2))
print(point2.distance_from_xy(2, 0))
```

```
1.4142135623730951
1.4142135623730951
```



See you Next week ☺