Python programming for beginners

Stefan Zhauryd
Instructor

Module 7

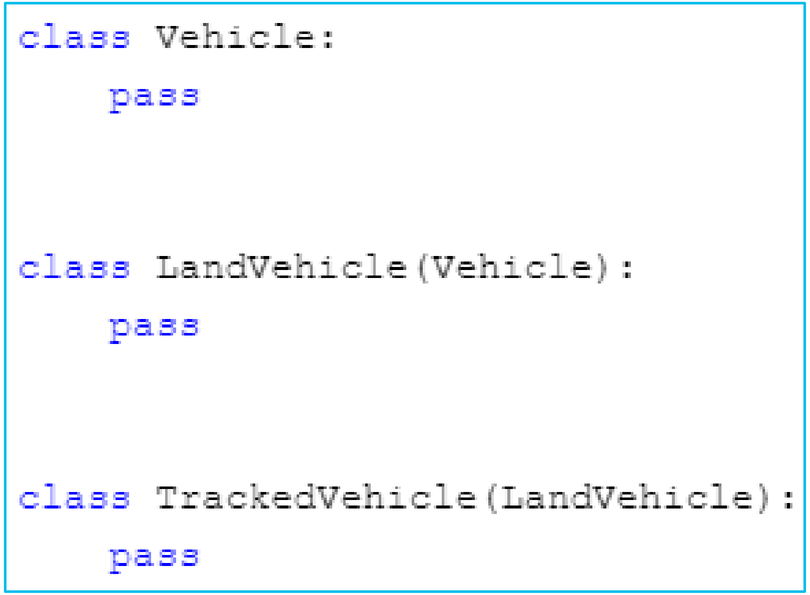
Object-Oriented Programming

In this module, you will learn about:

- Basic concepts of object-oriented programming (OOP)
- The differences between the procedural and object approaches (motivations and profits)
- Classes, objects, properties, and methods;
- **Designing reusable classes and creating objects;**
- **Inheritance and polymorphism;**
- **Exceptions as objects.**

Inheritance - why and how?

Inheritance is a common practice (in object programming) of passing attributes and methods from the superclass (defined and existing) to a newly created class, called the subclass.



```
class Vehicle:
    pass


class LandVehicle(Vehicle):
    pass


class TrackedVehicle(LandVehicle):
    pass
```

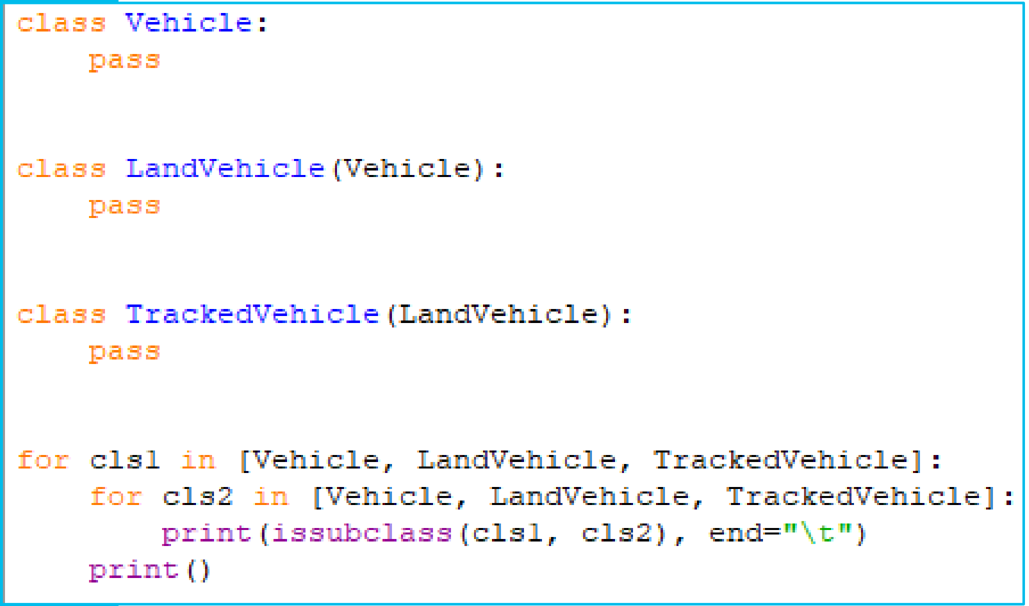
Inheritance: **issubclass()**

Python offers a function which is able to identify a relationship between two classes, and although its diagnosis isn't complex, it can check if a particular class is a subclass of any other class.

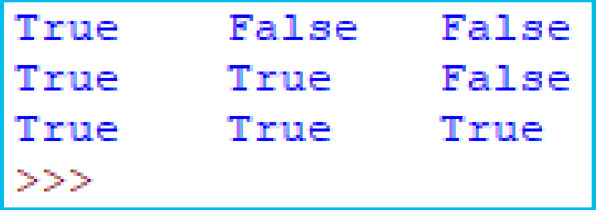
This is how it looks: **issubclass(ClassOne, ClassTwo)**

The function returns True if ClassOne is a subclass of ClassTwo, and False otherwise.

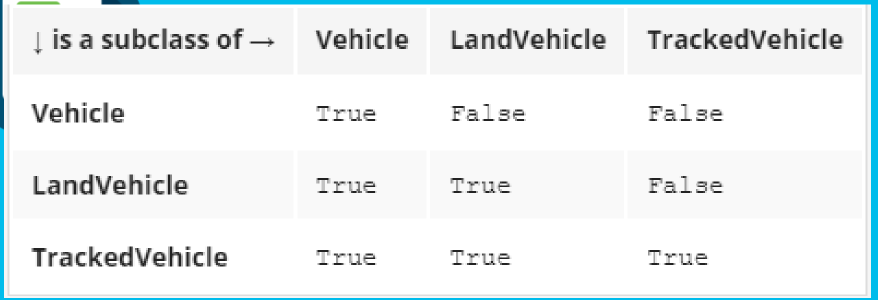
There is one important observation to make: each class is considered to be a subclass of itself.



```
class Vehicle:
    pass


class LandVehicle(Vehicle):
    pass


class TrackedVehicle(LandVehicle):
    pass


for cls1 in [Vehicle, LandVehicle, TrackedVehicle]:
    for cls2 in [Vehicle, LandVehicle, TrackedVehicle]:
        print(issubclass(cls1, cls2), end="\t")
    print()
```

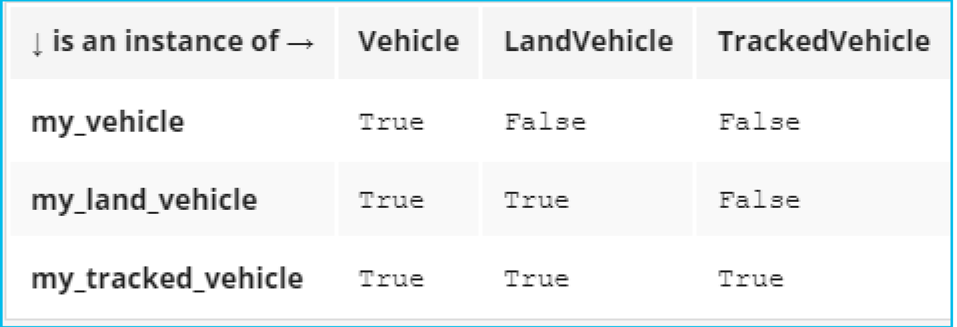


```
True	False	False
True	True	False
True	True	True
>>>
```



↓ is a subclass of →	Vehicle	LandVehicle	TrackedVehicle
Vehicle	True	False	False
LandVehicle	True	True	False
TrackedVehicle	True	True	True

Inheritance: **isinstance()**



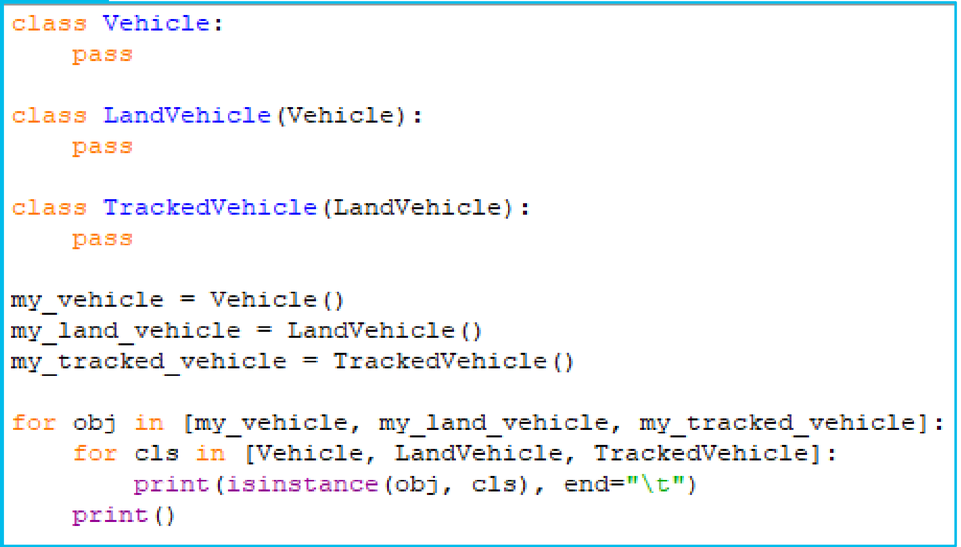
↓ is an instance of →	Vehicle	LandVehicle	TrackedVehicle
my_vehicle	`True`	`False`	`False`
my_land_vehicle	`True`	`True`	`False`
my_tracked_vehicle	`True`	`True`	`True`

Similarly, it can be crucial if the object does have (or doesn't have) certain characteristics. In other words, whether it is an object of a certain class or not.

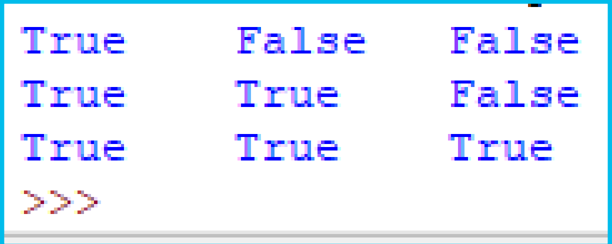
Such a fact could be detected by the function named isinstance():

isinstance(objectName, ClassName)

The functions returns True if the object is an instance of the class, or False otherwise.

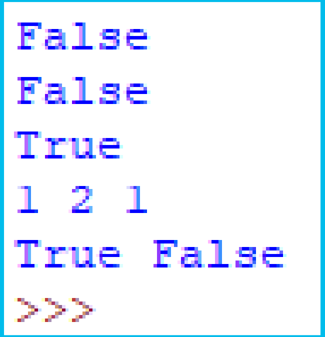


```python
class Vehicle:
    pass

class LandVehicle(Vehicle):
    pass

class TrackedVehicle(LandVehicle):
    pass

my_vehicle = Vehicle()
my_land_vehicle = LandVehicle()
my_tracked_vehicle = TrackedVehicle()

for obj in [my_vehicle, my_land_vehicle, my_tracked_vehicle]:
    for cls in [Vehicle, LandVehicle, TrackedVehicle]:
        print(isinstance(obj, cls), end="\t")
    print()
```



```
True    False   False
True    True    False
True    True    True
>>>
```

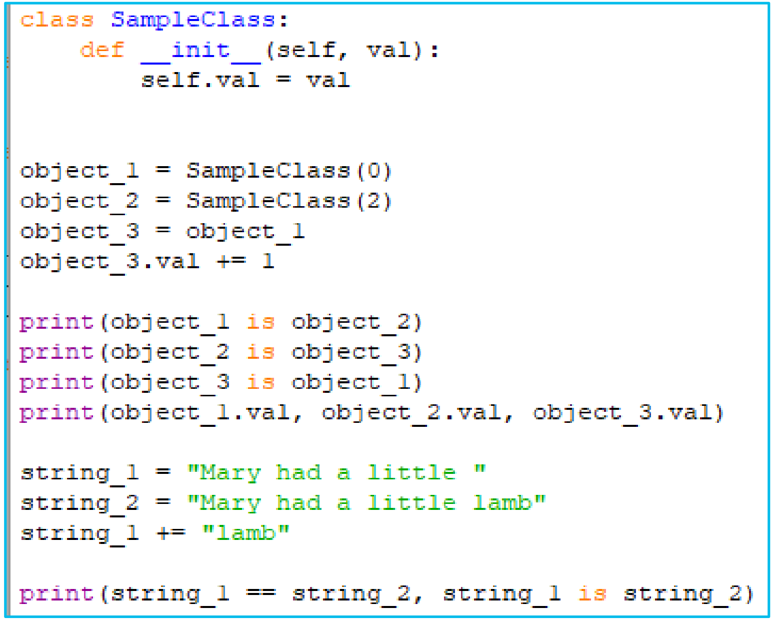
Inheritance: the **is** operator



```
False
False
True
1 2 1
True False
>>>
```

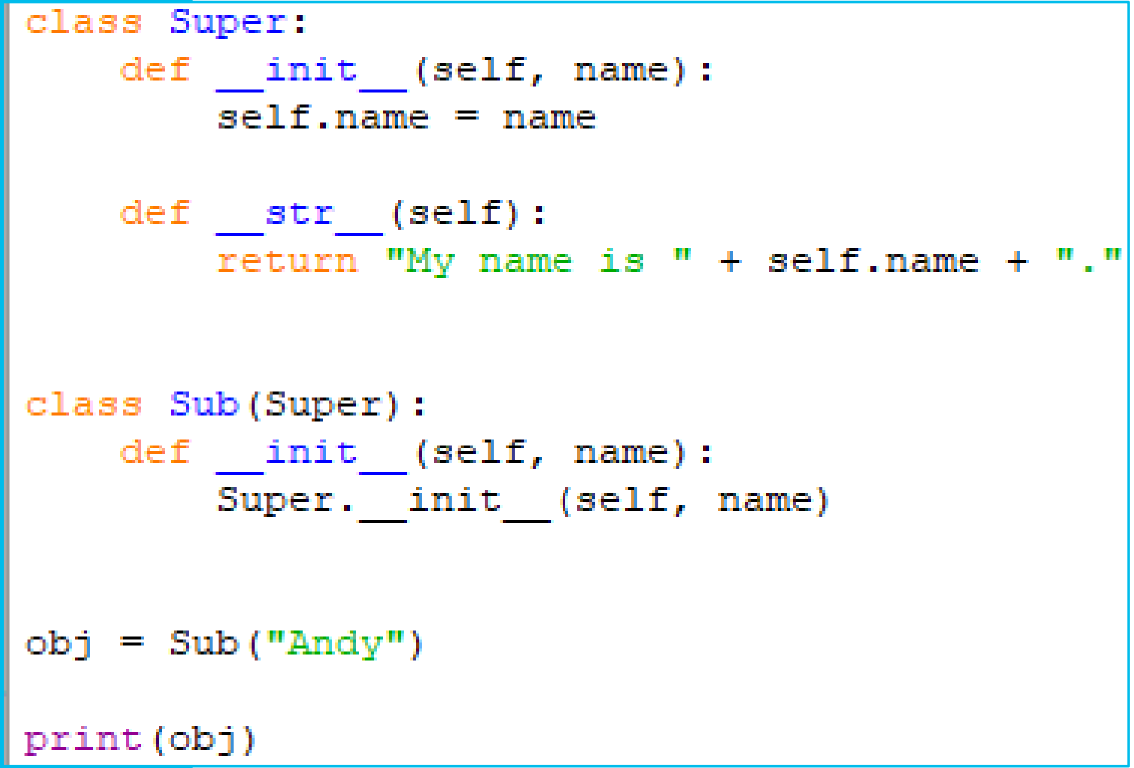
The is operator checks whether two variables (object_one and object_two here) refer to the same object.

Don't forget that variables don't store the objects themselves, but only the handles pointing to the internal Python memory.

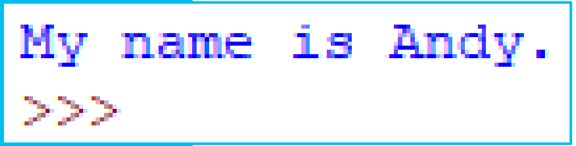


```
class SampleClass:
    def __init__(self, val):
        self.val = val


object_1 = SampleClass(0)
object_2 = SampleClass(2)
object_3 = object_1
object_3.val += 1

print(object_1 is object_2)
print(object_2 is object_3)
print(object_3 is object_1)
print(object_1.val, object_2.val, object_3.val)

string_1 = "Mary had a little "
string_2 = "Mary had a little lamb"
string_1 += "lamb"

print(string_1 == string_2, string_1 is string_2)
```

How Python finds properties and methods

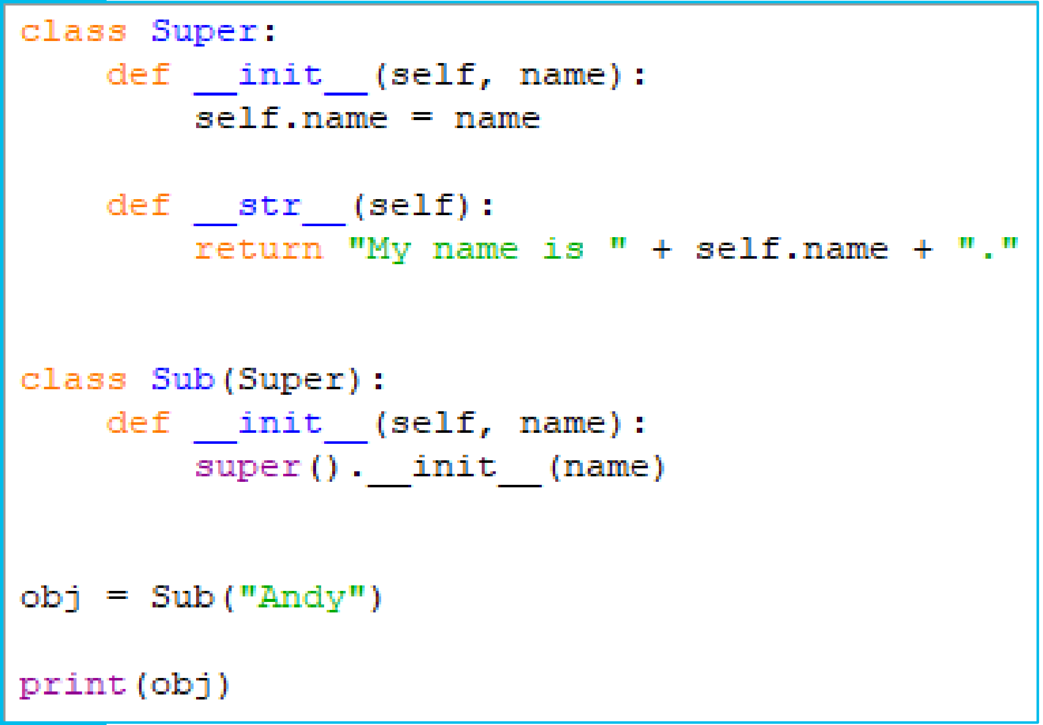


```python
class Super:
    def __init__(self, name):
        self.name = name

    def __str__(self):
        return "My name is " + self.name + "."


class Sub(Super):
    def __init__(self, name):
        Super.__init__(self, name)


obj = Sub("Andy")

print(obj)
```

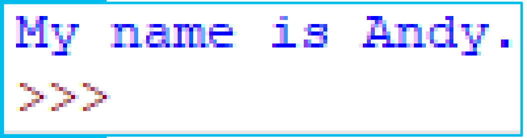


```
My name is Andy.
>>>
```

How Python finds properties and methods: continued

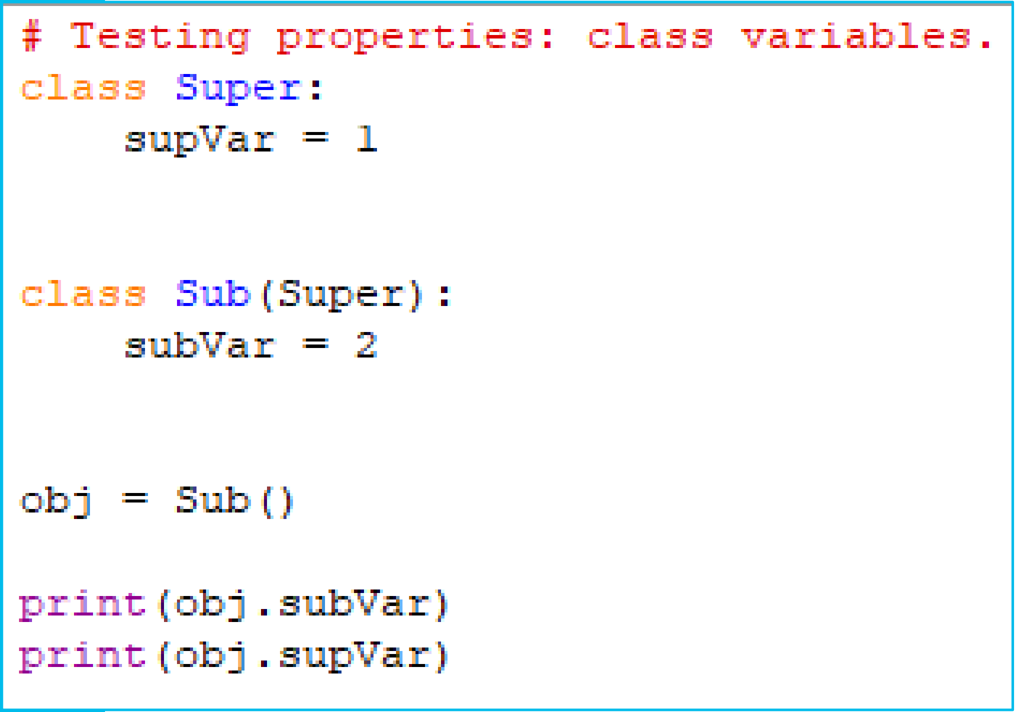


```python
class Super:
    def __init__(self, name):
        self.name = name

    def __str__(self):
        return "My name is " + self.name + "."


class Sub(Super):
    def __init__(self, name):
        super().__init__(name)


obj = Sub("Andy")

print(obj)
```

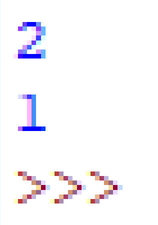


```
My name is Andy.
>>>
```

How Python finds properties and methods: continued class var

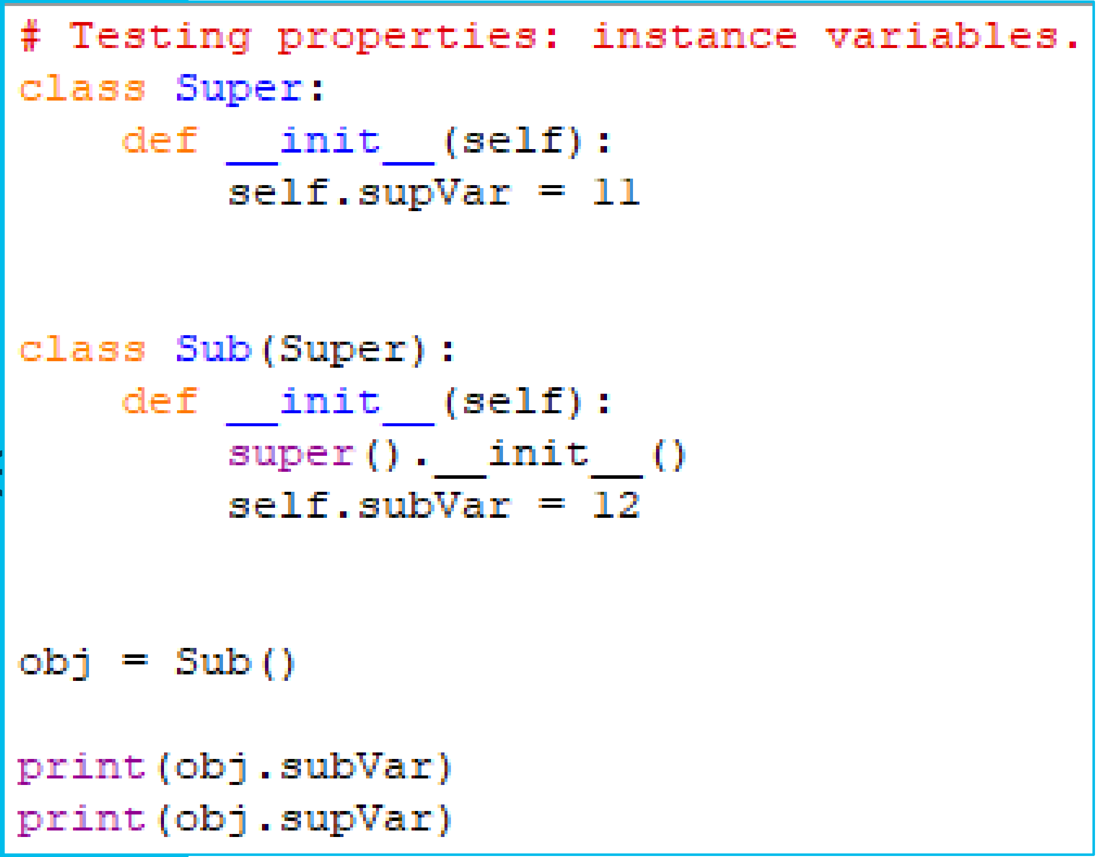


```
# Testing properties: class variables.
class Super:
    supVar = 1


class Sub(Super):
    subVar = 2


obj = Sub()

print(obj.subVar)
print(obj.supVar)
```

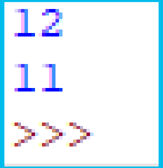


```
2
1
>>>
```

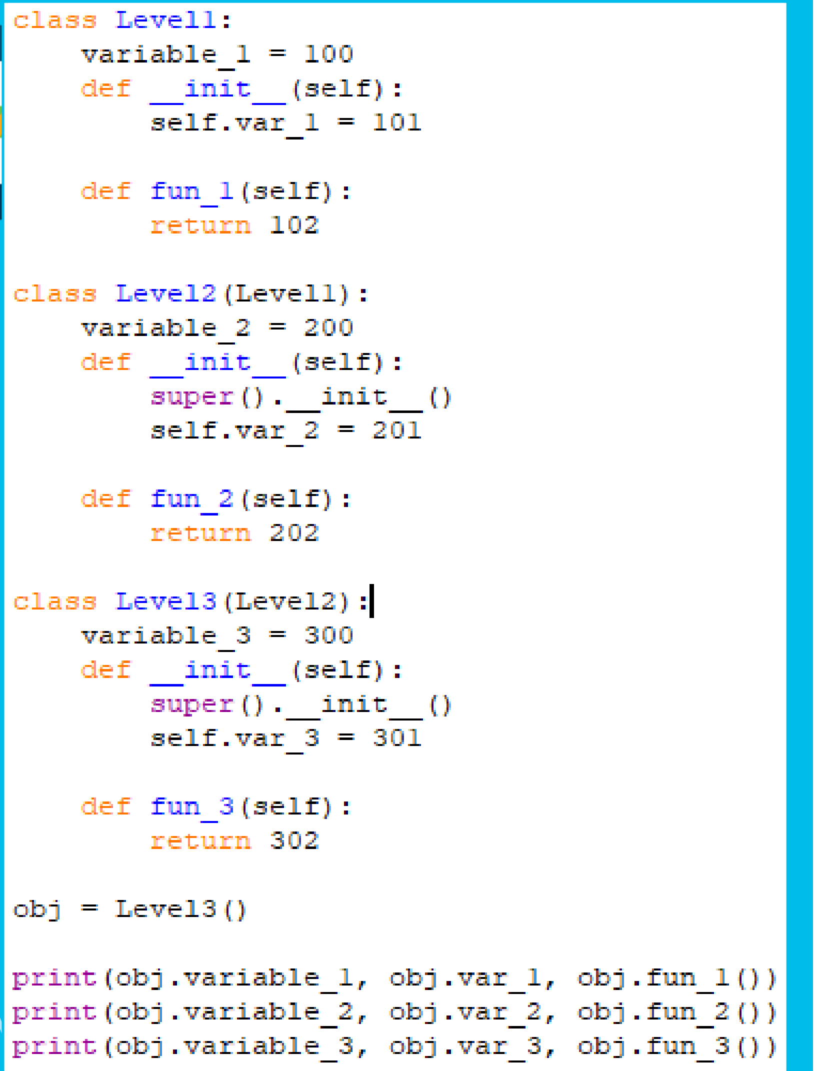
How Python finds properties and methods: continue instance var



```
# Testing properties: instance variables.
class Super:
    def __init__(self):
        self.supVar = 11


class Sub(Super):
    def __init__(self):
        super().__init__()
        self.subVar = 12


obj = Sub()

print(obj.subVar)
print(obj.supVar)
```



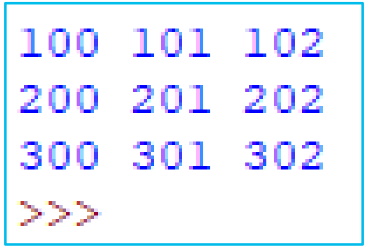
```
12
11
>>>
```



```
class Level1:
    variable_1 = 100
    def __init__(self):
        self.var_1 = 101

    def fun_1(self):
        return 102

class Level2(Level1):
    variable_2 = 200
    def __init__(self):
        super().__init__()
        self.var_2 = 201

    def fun_2(self):
        return 202

class Level3(Level2):
    variable_3 = 300
    def __init__(self):
        super().__init__()
        self.var_3 = 301

    def fun_3(self):
        return 302

obj = Level3()

print(obj.variable_1, obj.var_1, obj.fun_1())
print(obj.variable_2, obj.var_2, obj.fun_2())
print(obj.variable_3, obj.var_3, obj.fun_3())
```

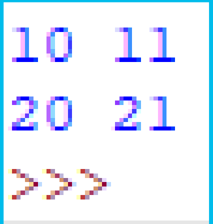
When you try to access any object's entity, Python will try to (in this order):

- find it inside the object itself;
- find it in all classes involved in the object's inheritance line from bottom to top;
- If both of the above fail, an exception (AttributeError) is raised.



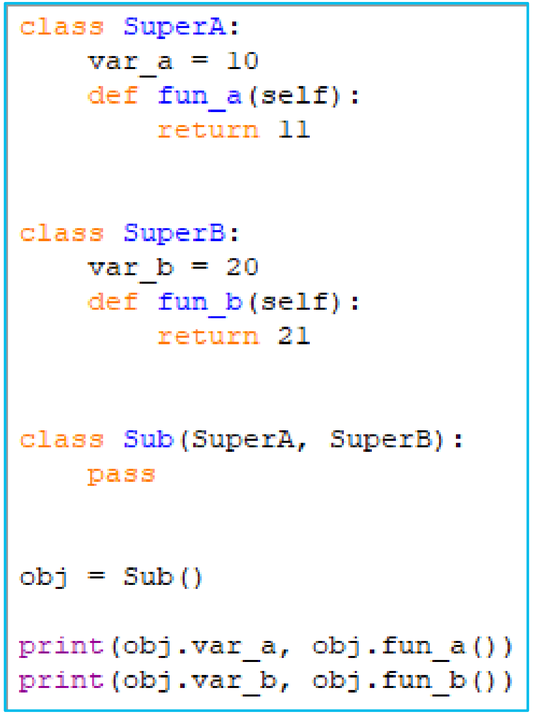
```
100 101 102
200 201 202
300 301 302
>>>
```

How Python finds properties and methods: continued



```
10 11
20 21
>>>
```

Multiple inheritance occurs when a class has more than one superclass. Syntactically, such inheritance is presented as a comma-separated list of superclasses put inside parentheses after the new class name

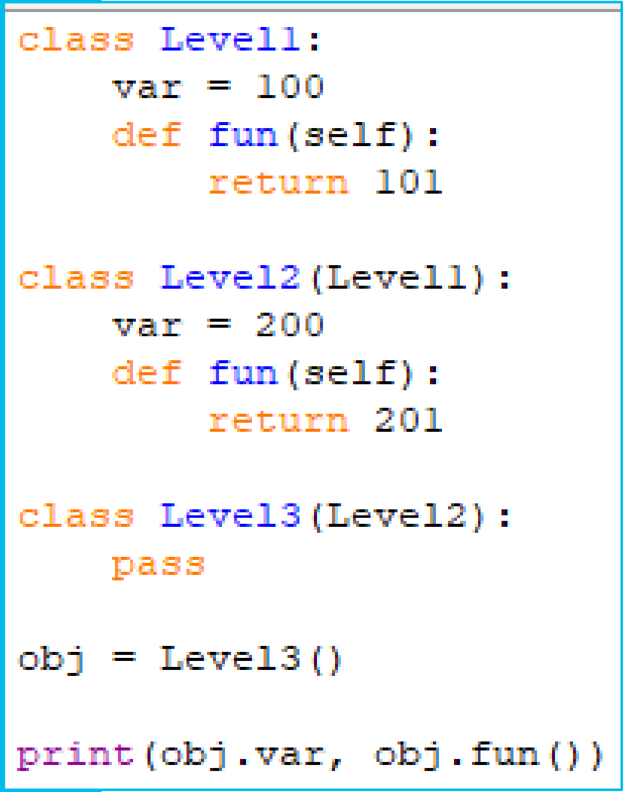


```
class SuperA:
    var_a = 10
    def fun_a(self):
        return 11


class SuperB:
    var_b = 20
    def fun_b(self):
        return 21


class Sub(SuperA, SuperB):
    pass


obj = Sub()

print(obj.var_a, obj.fun_a())
print(obj.var_b, obj.fun_b())
```

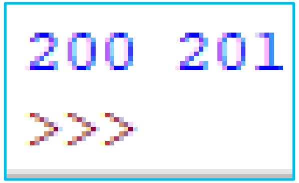
How Python finds properties and methods: continued

The entity defined later (in the inheritance sense) overrides the same entity defined earlier.

Python looks for an entity from bottom to top.



```
class Level1:
    var = 100
    def fun(self):
        return 101

class Level2(Level1):
    var = 200
    def fun(self):
        return 201

class Level3(Level2):
    pass

obj = Level3()

print(obj.var, obj.fun())
```

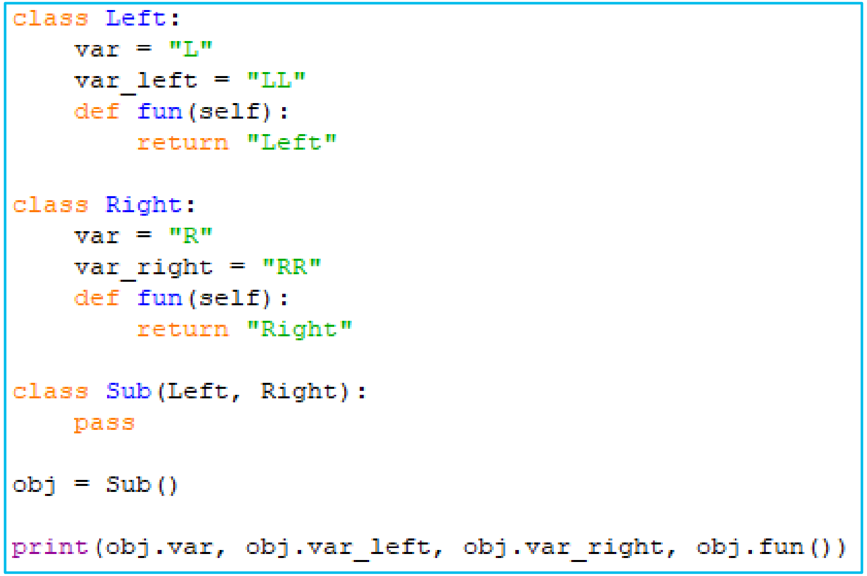


```
200 201
>>>
```

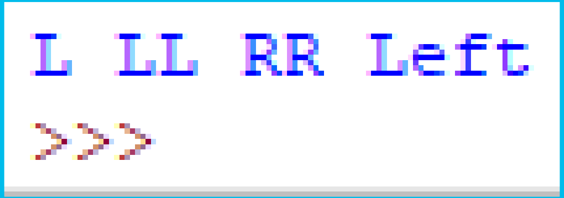
How Python finds properties and methods: continued

We can say that Python looks for object components in the following order:

- inside the object itself;
- in its superclasses, from bottom to top;
- if there is more than one class on a particular inheritance path, Python scans them from left to right.

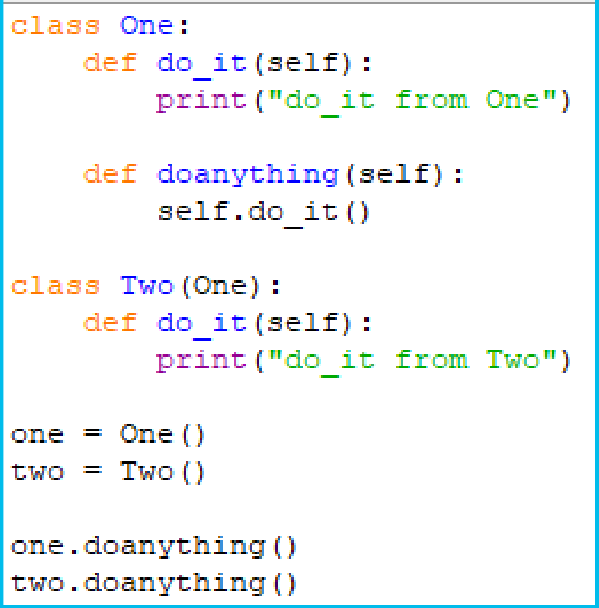


```
class Left:
    var = "L"
    var_left = "LL"
    def fun(self):
        return "Left"

class Right:
    var = "R"
    var_right = "RR"
    def fun(self):
        return "Right"

class Sub(Left, Right):
    pass

obj = Sub()

print(obj.var, obj.var_left, obj.var_right, obj.fun())
```

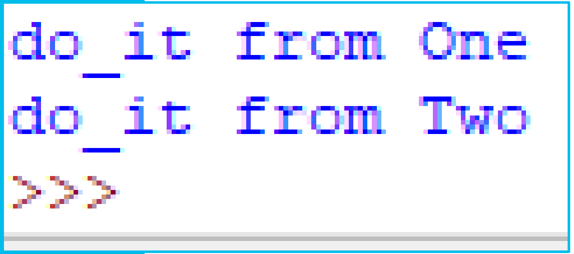


```
L LL RR Left
>>>
```

How to build a hierarchy of classes



```
class One:
    def do_it(self):
        print("do_it from One")

    def doanything(self):
        self.do_it()

class Two(One):
    def do_it(self):
        print("do_it from Two")

one = One()
two = Two()

one.doanything()
two.doanything()
```



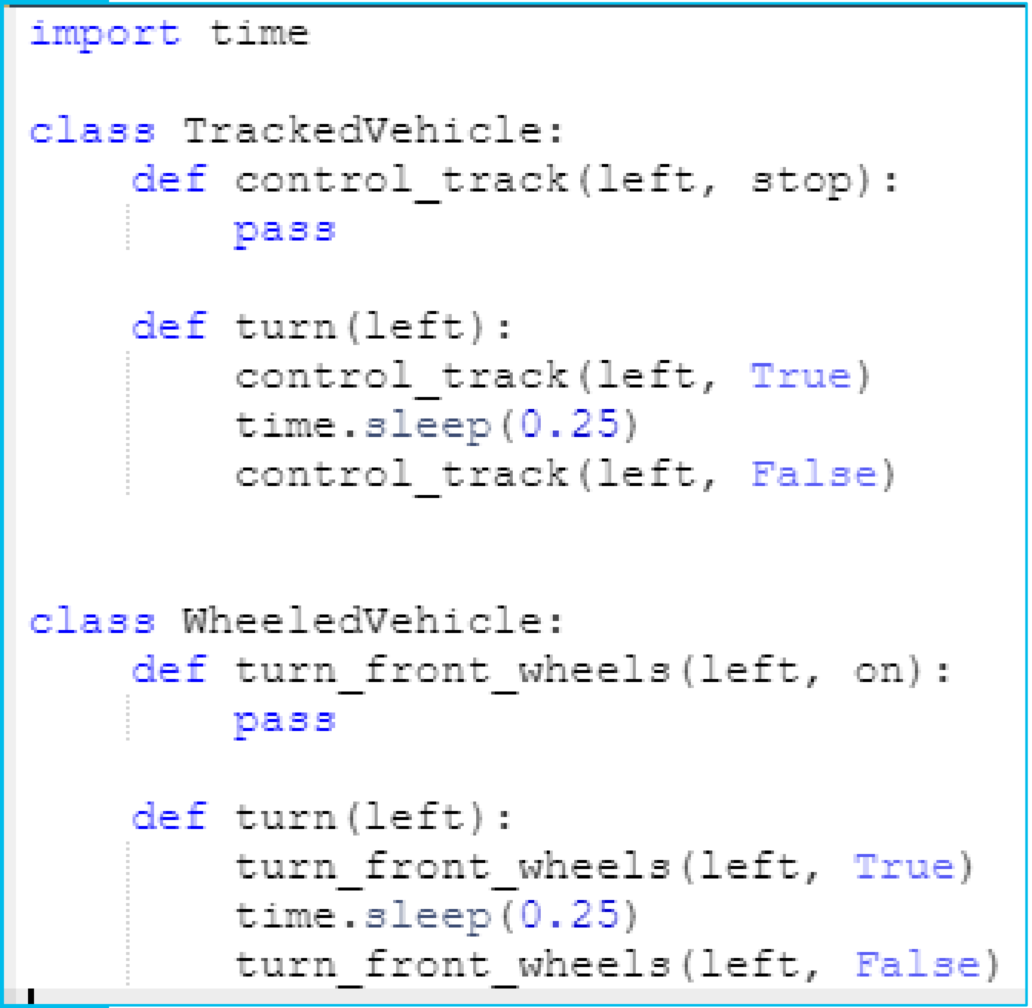
```
do_it from One
do_it from Two
>>>
```

The situation in which the subclass is able to modify its superclass behavior (just like in the example) is called polymorphism. The word comes from Greek (polys: "many, much" and morphe, "form, shape"), which means that one and the same class can take various forms depending on the redefinitions done by any of its subclasses.

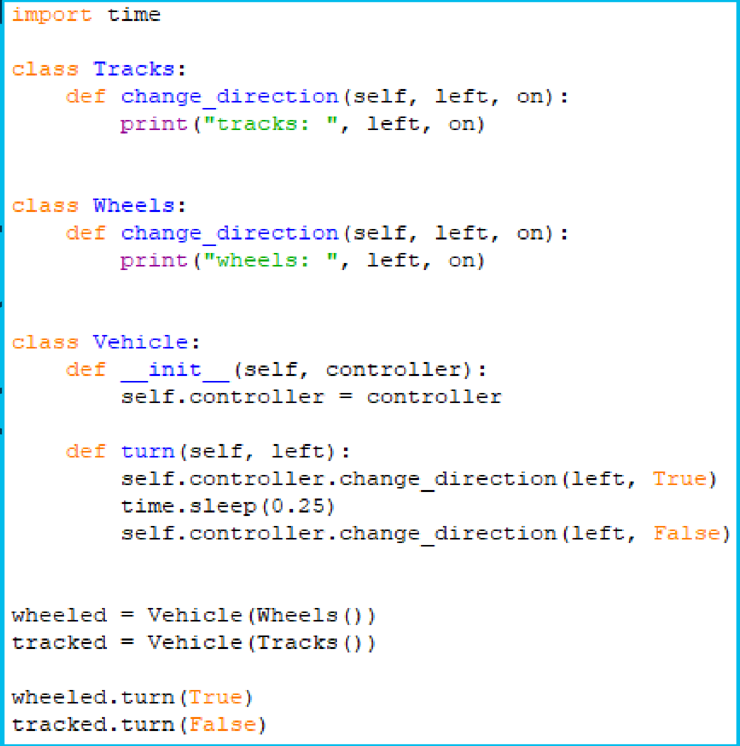
The method, redefined in any of the superclasses, thus changing the behavior of the superclass, is **called virtual.**

In other words, no class is given once and for all. Each class's behavior may be modified at any time by any of its subclasses.

How to build a hierarchy of classes: continued



```python
import time

class TrackedVehicle:
    def control_track(left, stop):
        pass

    def turn(left):
        control_track(left, True)
        time.sleep(0.25)
        control_track(left, False)


class WheeledVehicle:
    def turn_front_wheels(left, on):
        pass

    def turn(left):
        turn_front_wheels(left, True)
        time.sleep(0.25)
        turn_front_wheels(left, False)
```

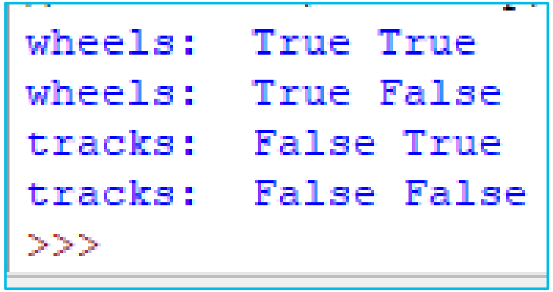


```python
import time

class Tracks:
    def change_direction(self, left, on):
        print("tracks: ", left, on)


class Wheels:
    def change_direction(self, left, on):
        print("wheels: ", left, on)


class Vehicle:
    def __init__(self, controller):
        self.controller = controller

    def turn(self, left):
        self.controller.change_direction(left, True)
        time.sleep(0.25)
        self.controller.change_direction(left, False)


wheeled = Vehicle(Wheels())
tracked = Vehicle(Tracks())

wheeled.turn(True)
tracked.turn(False)
```

Inheritance extends a class's capabilities by adding new components and modifying existing ones; in other words, the complete recipe is contained inside the class itself and all its ancestors; the object takes all the class's belongings and makes use of them;

Composition projects a class as a container able to store and use other objects (derived from other classes) where each of the objects implements a part of a desired class's behavior.

a = int(input())



```
wheels:  True True
wheels:  True False
tracks:  False True
tracks:  False False
>>>
```

Single inheritance vs. multiple inheritance

Don't forget that:

- a single inheritance class is always simpler, safer, and easier to understand and maintain;
- multiple inheritance is always risky, as you have many more opportunities to make a mistake in identifying these parts of the superclasses which will effectively influence the new class;
- multiple inheritance may make overriding extremely tricky; moreover, using the super() function becomes ambiguous;
- multiple inheritance violates the single responsibility principle (more details here: https://en.wikipedia.org/wiki/Single_responsibility_principle) as it makes a new class of two (or more) classes that know nothing about each other;
- we strongly suggest multiple inheritance as the last of all possible solutions - if you really need the many different functionalities offered by different classes, composition may be a better alternative.

Home work 10_2_1

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.)

https://ru.wikipedia.org/wiki/Очередь_(программирование)

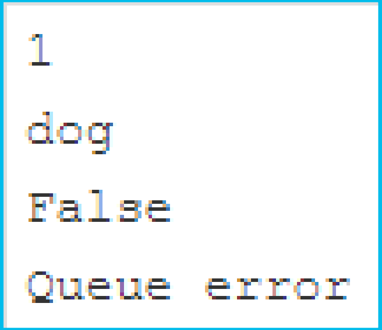
Home work 10_2_1

Follow the hints:

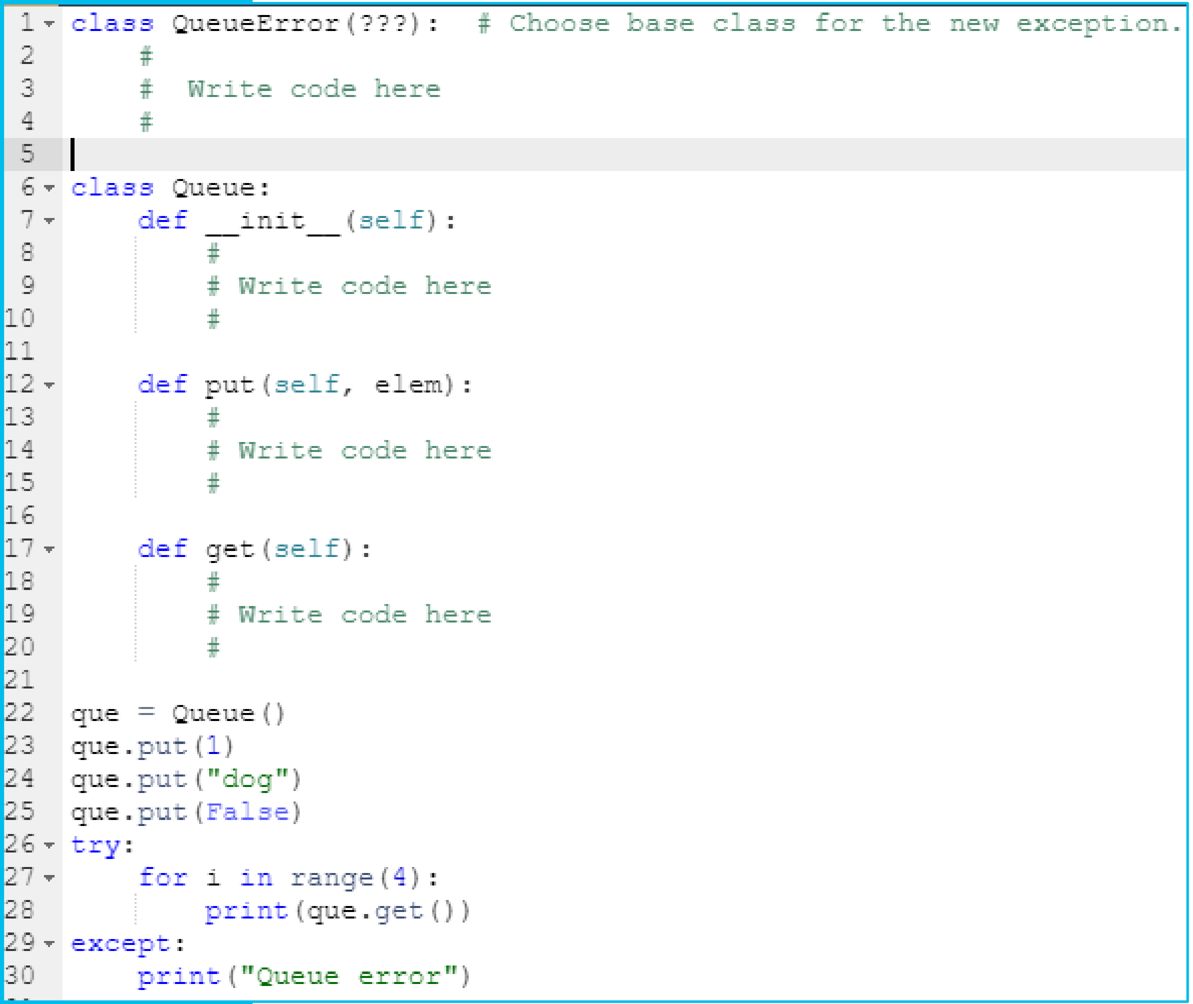
- use a list as your storage (just like we did in stack)
- **put()** should append elements to the beginning of the list, while get() should remove the elements from the list's end;
- *****define a new exception** named **QueueError** (choose an exception to derive it from) and raise it when get() tries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

https://ru.wikipedia.org/wiki/Очередь_(программирование)

Home work 10_2_1 IndexError QueErr as q

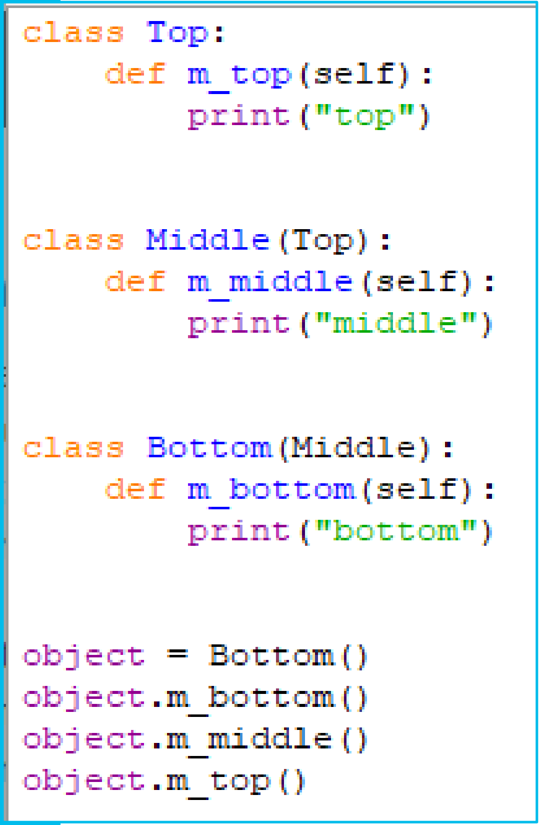


```
1
dog
False
Queue error
```

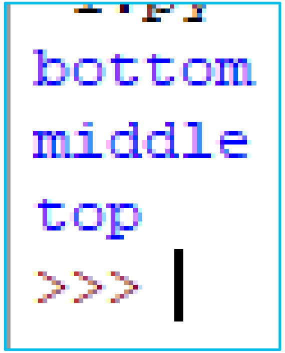


```python
class QueueError(???):   # Choose base class for the new exception.
    #
    #  Write code here
    #

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")
```

What is Method Resolution Order (**MRO**) and why is it that not all inheritances make sense?



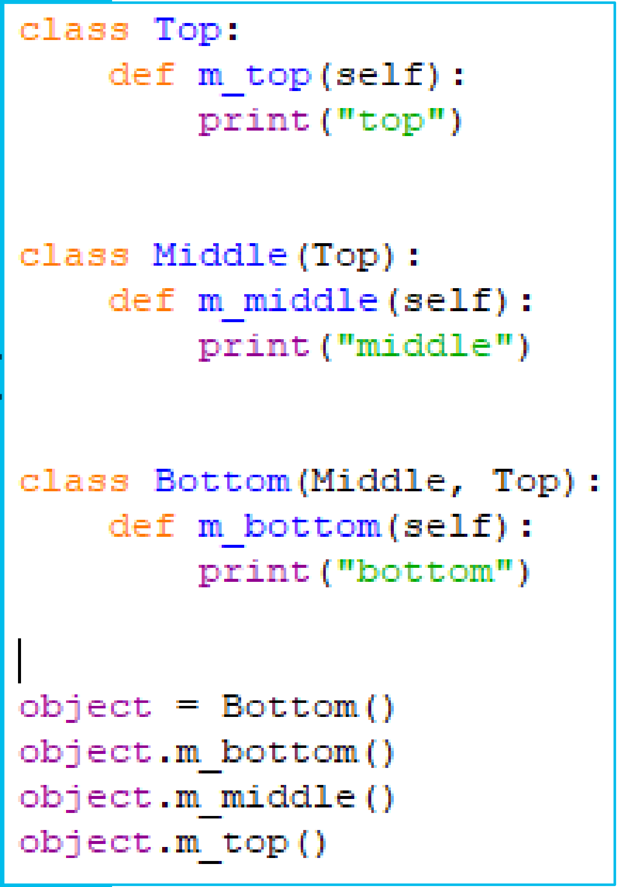
```
class Top:
    def m_top(self):
        print("top")


class Middle(Top):
    def m_middle(self):
        print("middle")


class Bottom(Middle):
    def m_bottom(self):
        print("bottom")


object = Bottom()
object.m_bottom()
object.m_middle()
object.m_top()
```



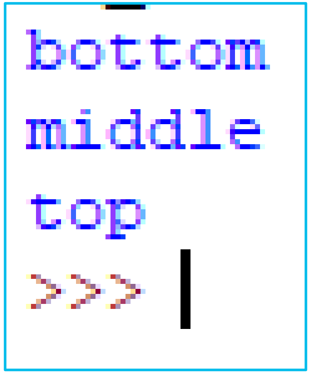
```
bottom
middle
top
>>> 
```

All rights reserved © Confidential

What is Method Resolution Order (MRO) and why is it that not all inheritances make sense?

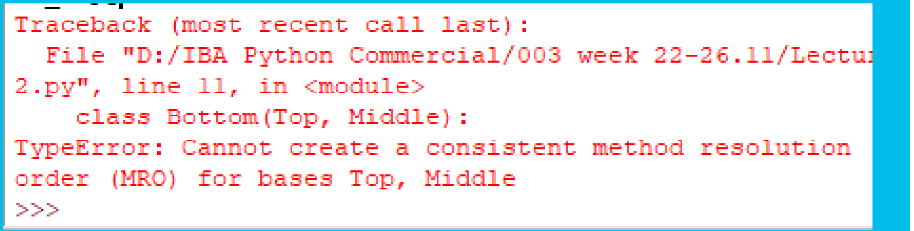


```
class Top:
    def m_top(self):
        print("top")


class Middle(Top):
    def m_middle(self):
        print("middle")


class Bottom(Middle, Top):
    def m_bottom(self):
        print("bottom")


object = Bottom()
object.m_bottom()
object.m_middle()
object.m_top()
```

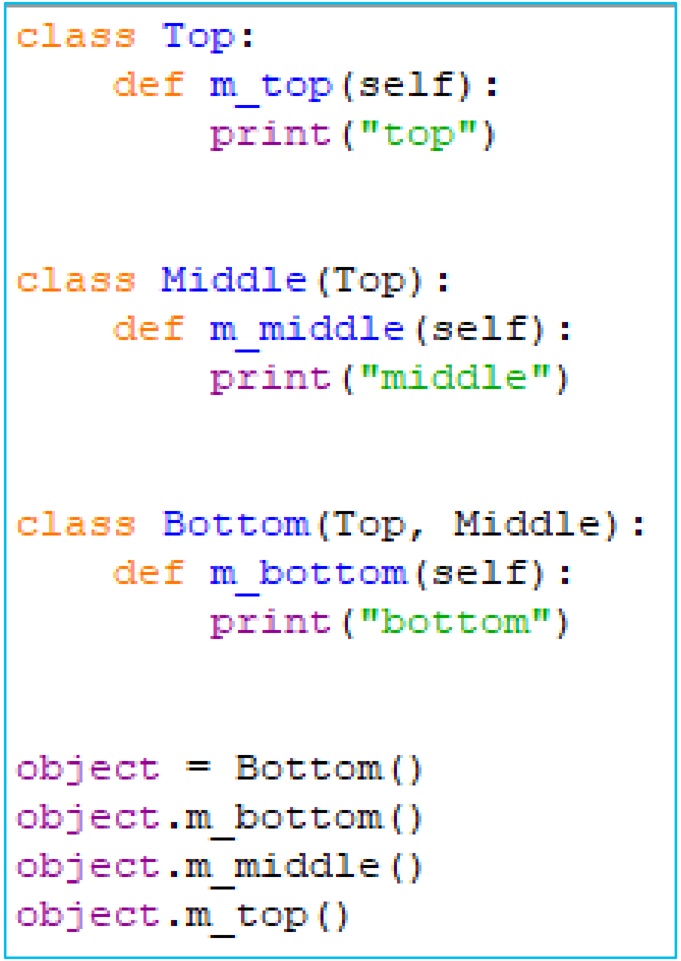


```
bottom
middle
top
>>> 
```

What is Method Resolution Order (MRO) and why is it that not all inheritances make sense?

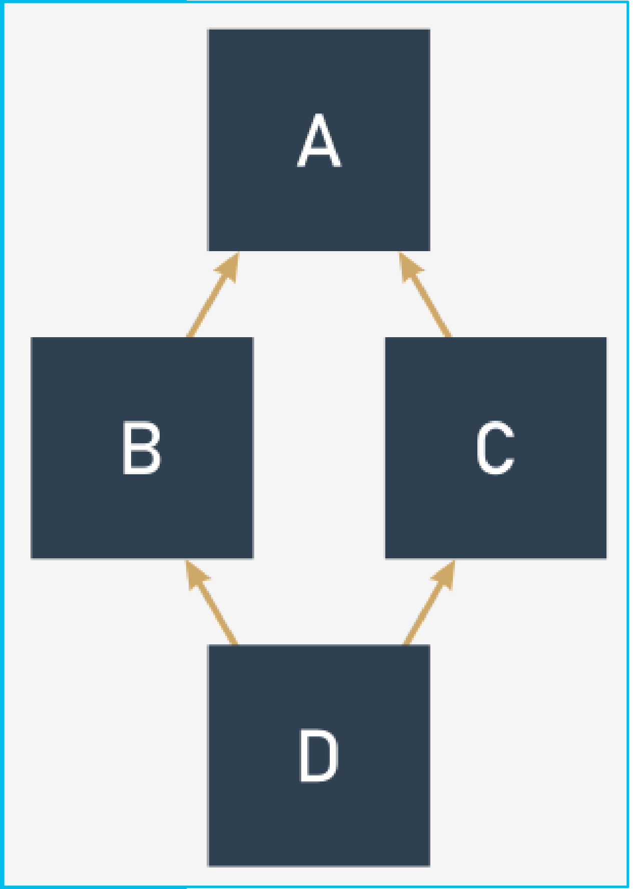


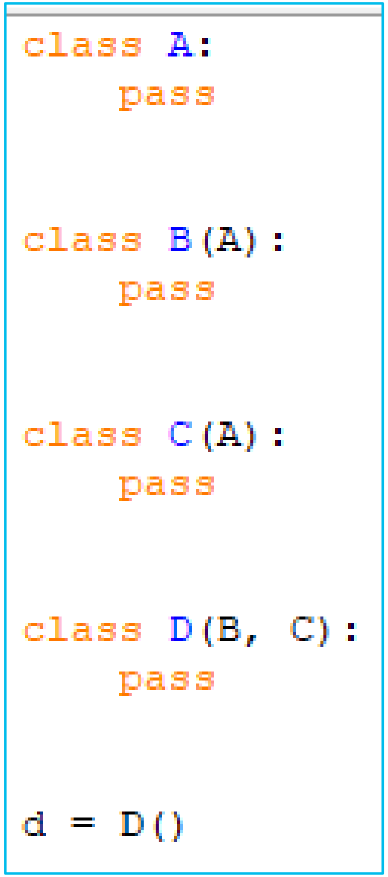
```
Traceback (most recent call last):
  File "D:/IBA Python Commercial/003 week 22-26.11/Lectu
2.py", line 11, in <module>
    class Bottom(Top, Middle):
TypeError: Cannot create a consistent method resolution
order (MRO) for bases Top, Middle
>>>
```



```
class Top:
    def m_top(self):
        print("top")


class Middle(Top):
    def m_middle(self):
        print("middle")


class Bottom(Top, Middle):
    def m_bottom(self):
        print("bottom")


object = Bottom()
object.m_bottom()
object.m_middle()
object.m_top()
```

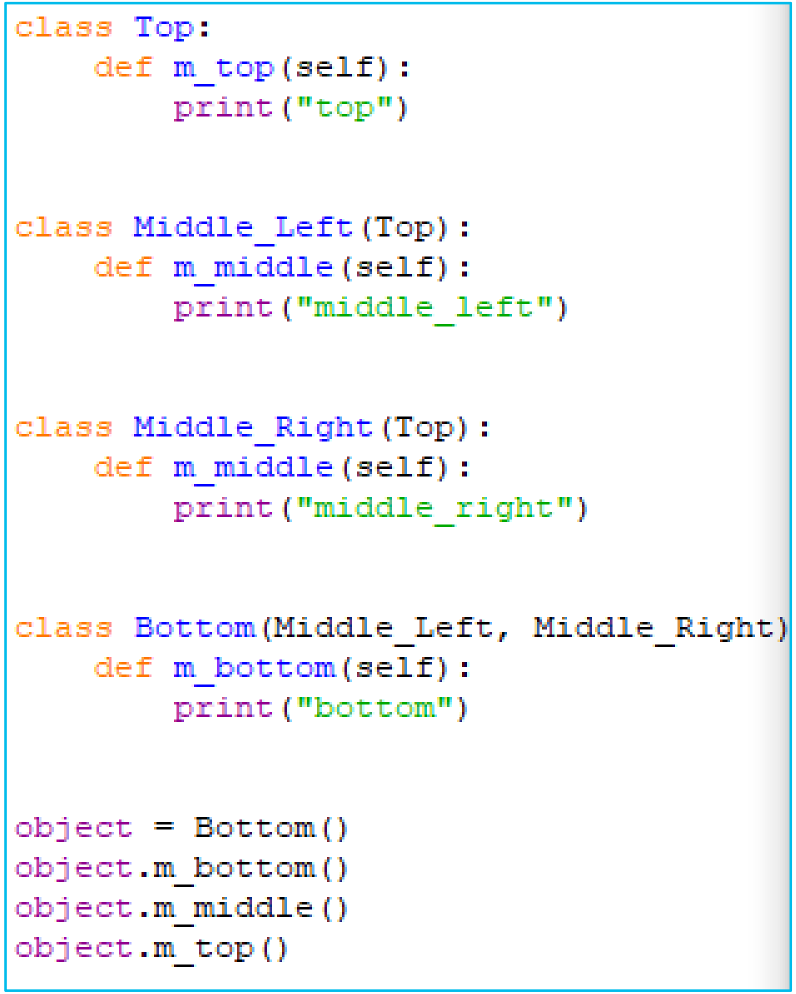
The diamond problem



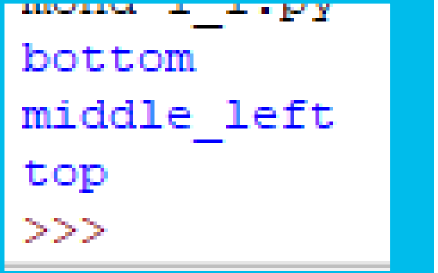


```
class A:
    pass


class B(A):
    pass


class C(A):
    pass


class D(B, C):
    pass


d = D()
```

The diamond problem



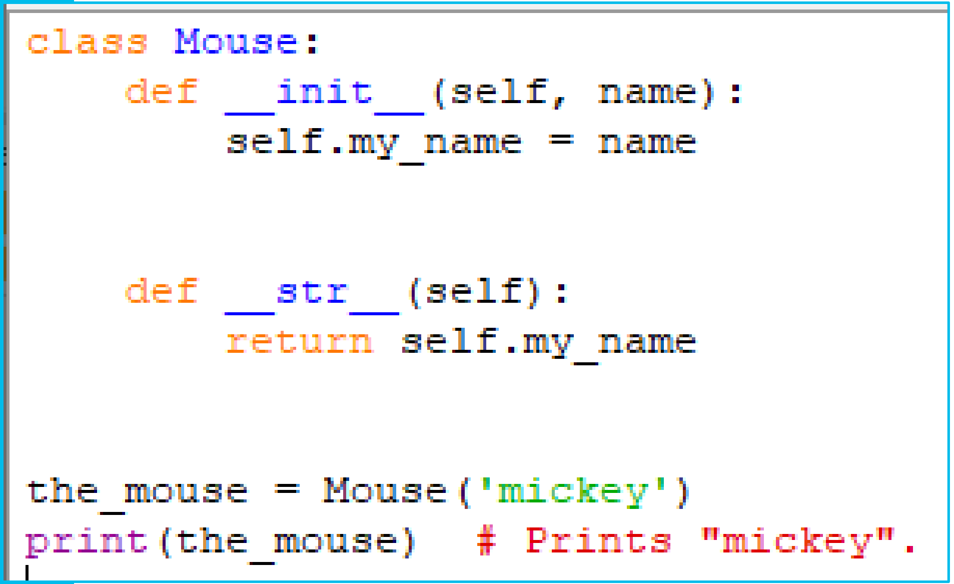
```
class Top:
    def m_top(self):
        print("top")


class Middle_Left(Top):
    def m_middle(self):
        print("middle_left")


class Middle_Right(Top):
    def m_middle(self):
        print("middle_right")


class Bottom(Middle_Left, Middle_Right):
    def m_bottom(self):
        print("bottom")


object = Bottom()
object.m_bottom()
object.m_middle()
object.m_top()
```



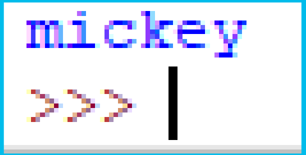
```
bottom
middle_left
top
>>>
```

Key takeaways

1. A **method named __str__()** is responsible for converting an object's contents into a (more or less) readable string. You can redefine it if you want your object to be able to present itself in a more elegant form.



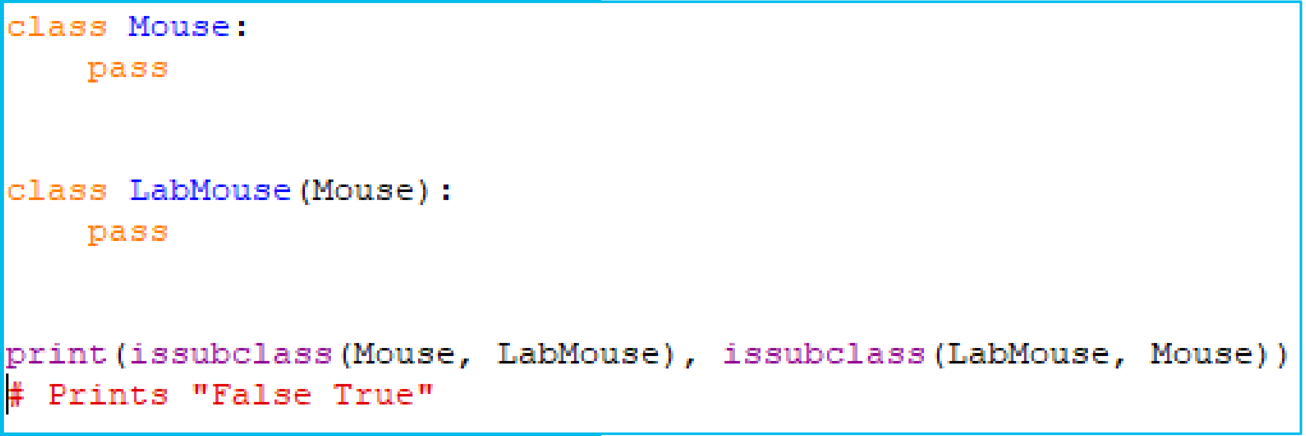
```
class Mouse:
    def __init__(self, name):
        self.my_name = name


    def __str__(self):
        return self.my_name


the_mouse = Mouse('mickey')
print(the_mouse)  # Prints "mickey".
```



```
mickey
>>> 
```

Key takeaways

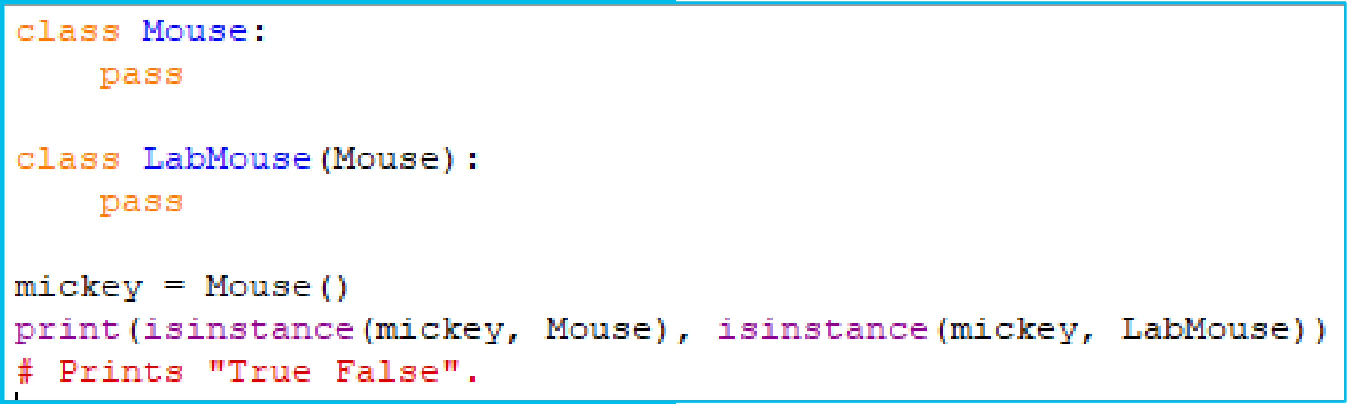
2. A function named **issubclass**(Class_1, Class_2) is able to determine if Class_1 is a subclass of Class_2.



```
class Mouse:
    pass


class LabMouse(Mouse):
    pass


print(issubclass(Mouse, LabMouse), issubclass(LabMouse, Mouse))
# Prints "False True"
```

Key takeaways

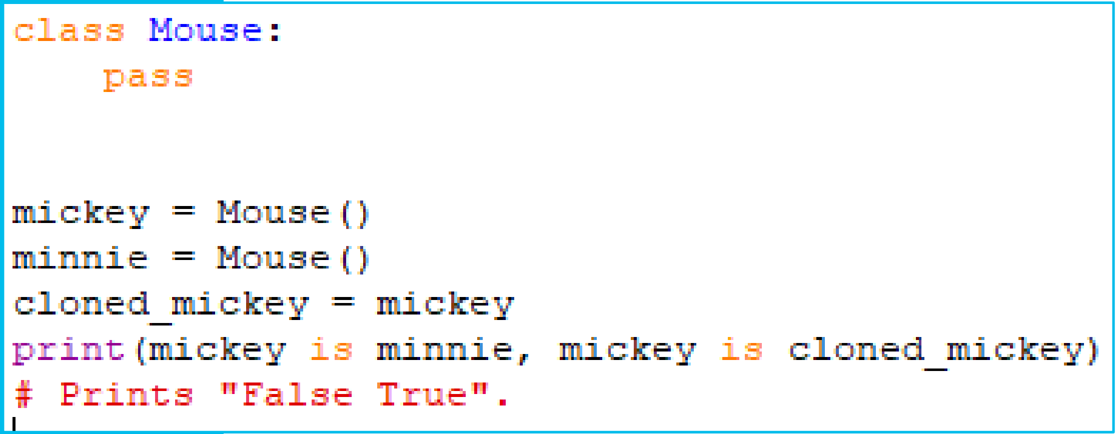
3. A function named **isinstance**(Object, Class) checks if an object comes from an indicated class.



```
class Mouse:
    pass

class LabMouse(Mouse):
    pass

mickey = Mouse()
print(isinstance(mickey, Mouse), isinstance(mickey, LabMouse))
# Prints "True False".
```

Key takeaways

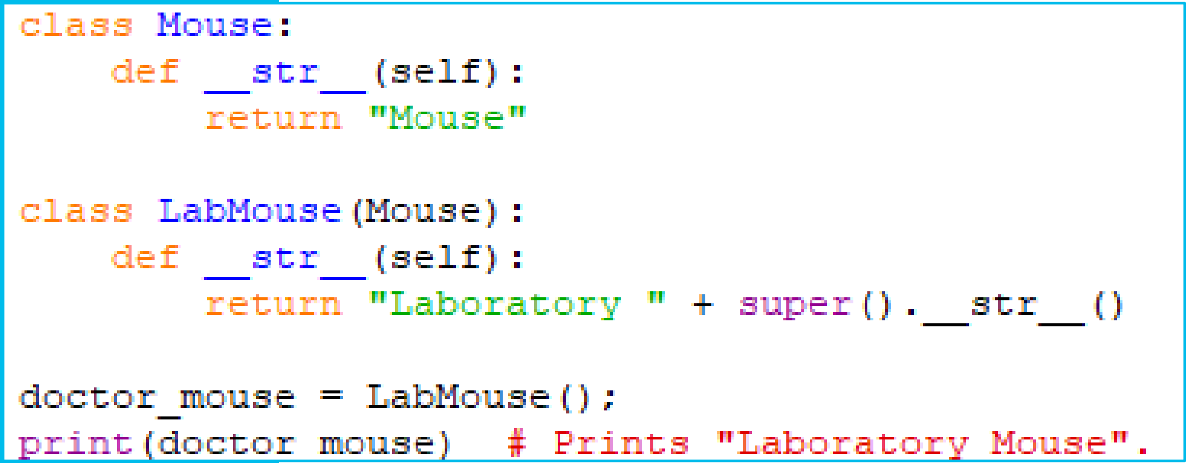
4. A operator called **is** checks if two variables refer to the same object.



```python
class Mouse:
    pass


mickey = Mouse()
minnie = Mouse()
cloned_mickey = mickey
print(mickey is minnie, mickey is cloned_mickey)
# Prints "False True".
```

Key takeaways

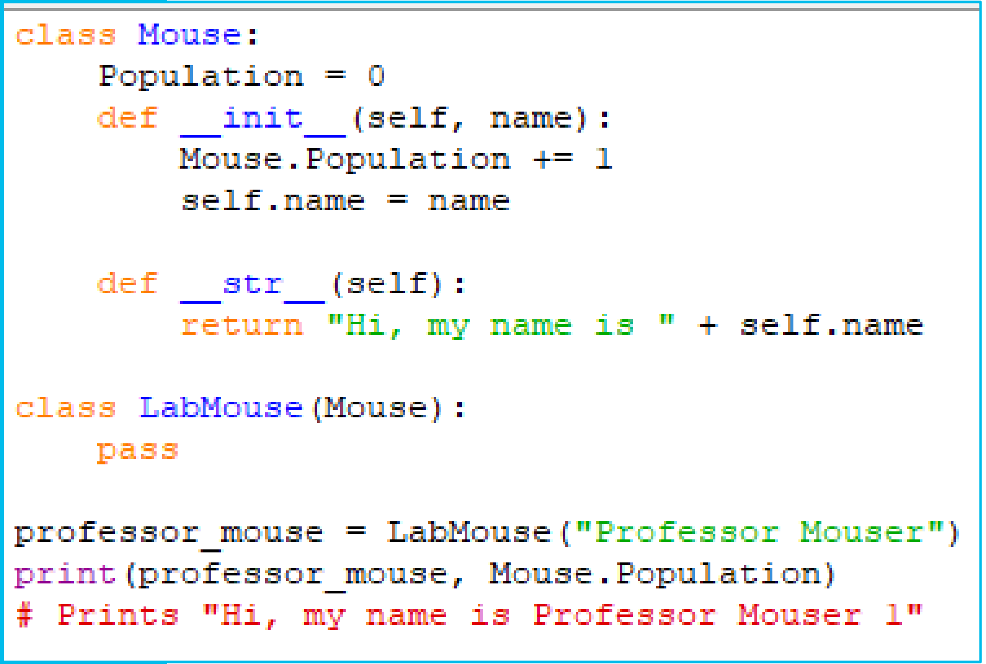
5. A parameterless function named **super()** returns a reference to the nearest superclass of the class.



```
class Mouse:
    def __str__(self):
        return "Mouse"

class LabMouse(Mouse):
    def __str__(self):
        return "Laboratory " + super().__str__()

doctor_mouse = LabMouse();
print(doctor mouse)  # Prints "Laboratory Mouse".
```

Key takeaways

6. Methods as well as instance and **class variables defined in a superclass are automatically inherited by their subclasses.**



```
class Mouse:
    Population = 0
    def __init__(self, name):
        Mouse.Population += 1
        self.name = name

    def __str__(self):
        return "Hi, my name is " + self.name

class LabMouse(Mouse):
    pass

professor_mouse = LabMouse("Professor Mouser")
print(professor_mouse, Mouse.Population)
# Prints "Hi, my name is Professor Mouser 1"
```

Key takeaways

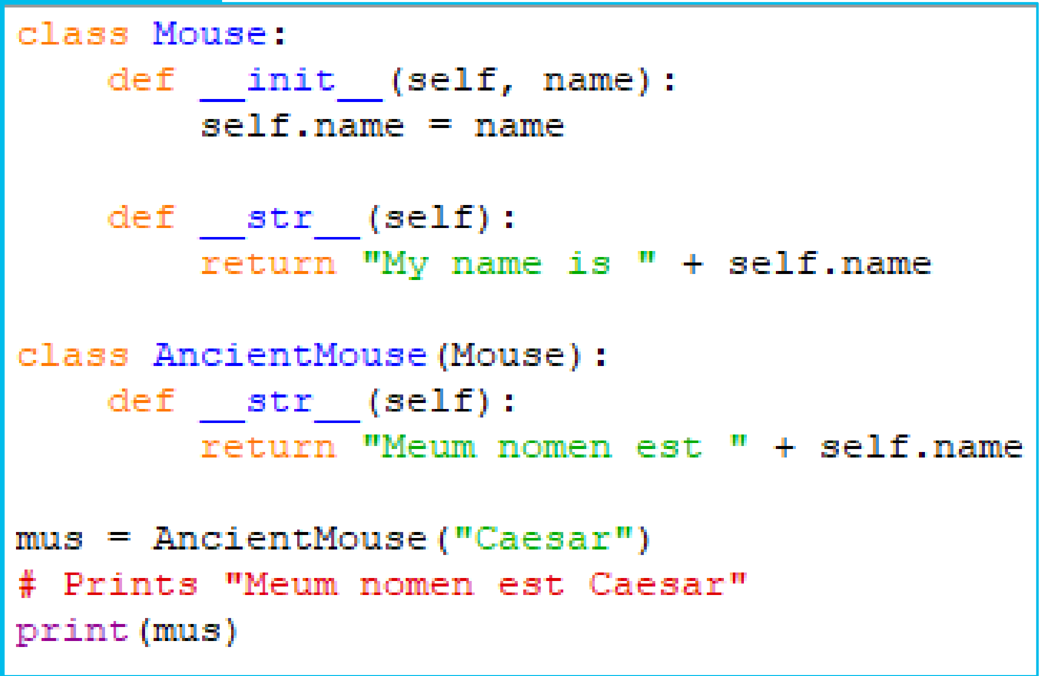
7. In order to find any object/class property, Python looks for it inside:

- the object itself;
- all classes involved in the object's inheritance line from bottom to top;
- if there is more than one class on a particular inheritance path, Python scans them from left to right;

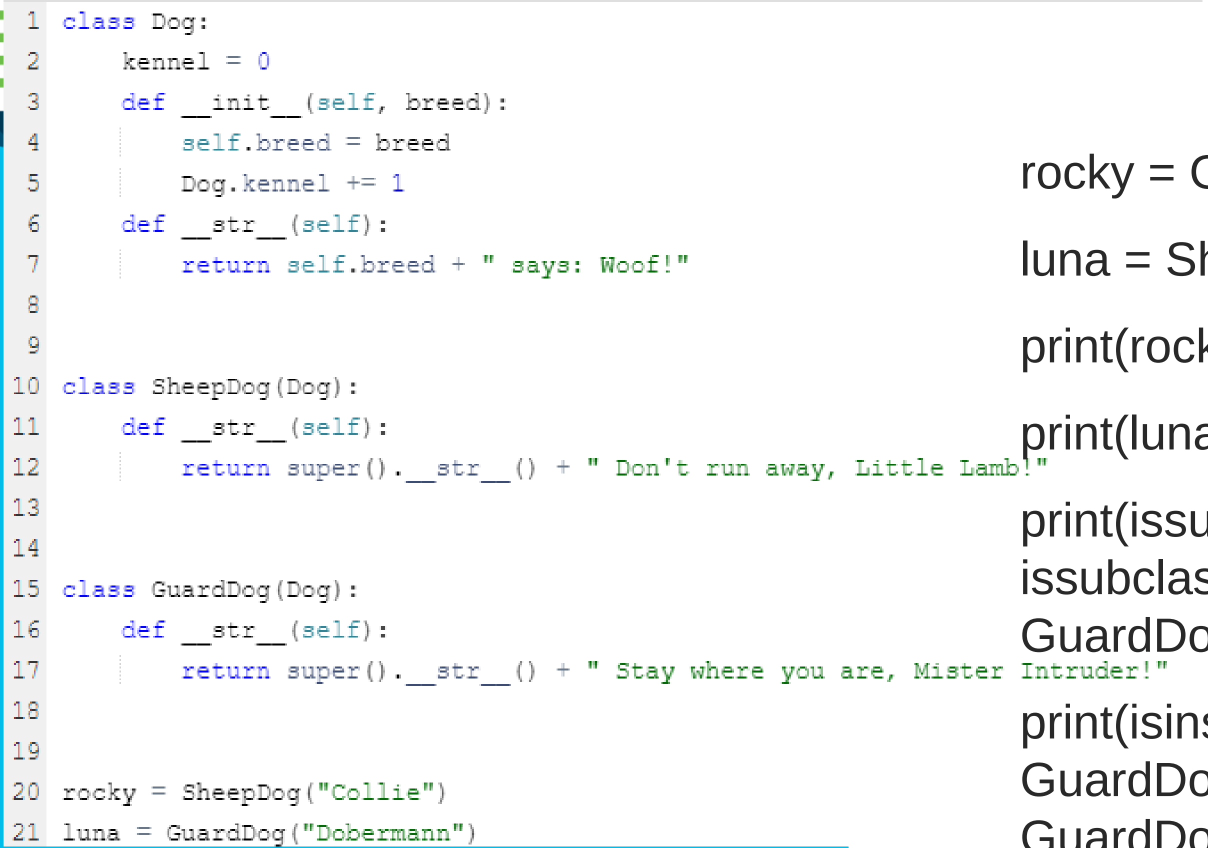
if both of the above fail, the AttributeError exception is raised.

Key takeaways

8. If any of the subclasses defines a method/class variable/instance variable of the same name as existing in the superclass, the new name overrides any of the previous instances of the name.



```python
class Mouse:
    def __init__(self, name):
        self.name = name

    def __str__(self):
        return "My name is " + self.name

class AncientMouse(Mouse):
    def __str__(self):
        return "Meum nomen est " + self.name

mus = AncientMouse("Caesar")
# Prints "Meum nomen est Caesar"
print(mus)
```



```python
class Dog:
    kennel = 0
    def __init__(self, breed):
        self.breed = breed
        Dog.kennel += 1
    def __str__(self):
        return self.breed + " says: Woof!"


class SheepDog(Dog):
    def __str__(self):
        return super().__str__() + " Don't run away, Little Lamb!"


class GuardDog(Dog):
    def __str__(self):
        return super().__str__() + " Stay where you are, Mister Intruder!"


rocky = SheepDog("Collie")
luna = GuardDog("Dobermann")
```

rocky = GuardDog()

luna = SheepDog()

print(rocky)

print(luna)

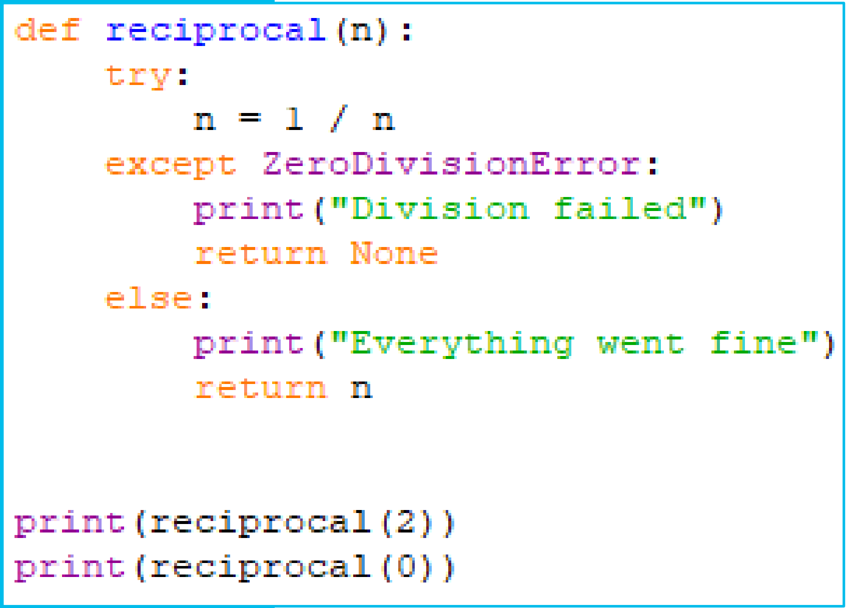
print(issubclass(SheepDog, Dog), issubclass(SheepDog, GuardDog))

print(isinstance(rocky, GuardDog), isinstance(luna, GuardDog))

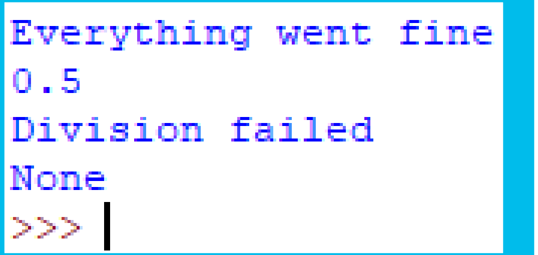
More about exceptions

Exactly one branch can be executed after try: - either the one beginning with except (don't forget that there can be more than one branch of this kind) or the one starting with **else**.

Note: the else: branch has to be located after the last except branch.

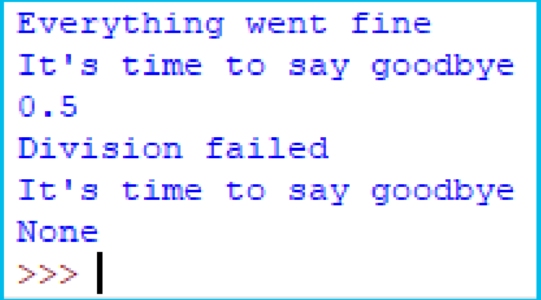


```
def reciprocal(n):
    try:
        n = 1 / n
    except ZeroDivisionError:
        print("Division failed")
        return None
    else:
        print("Everything went fine")
        return n


print(reciprocal(2))
print(reciprocal(0))
```



```
Everything went fine
0.5
Division failed
None
>>> 
```

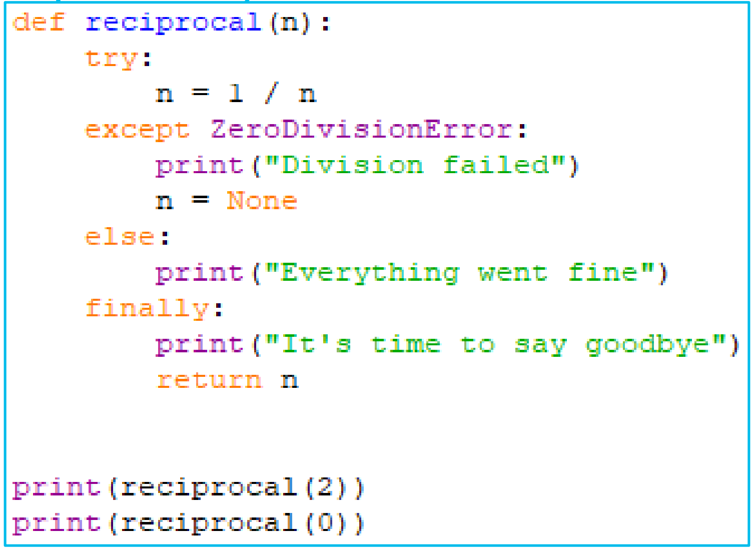
More about exceptions



```
Everything went fine
It's time to say goodbye
0.5
Division failed
It's time to say goodbye
None
>>> 
```

The **finally** keyword (it must be the last branch of the code designed to handle exceptions).

Note: these two variants (else and finally) aren't dependent in any way, and they can coexist or occur independently.

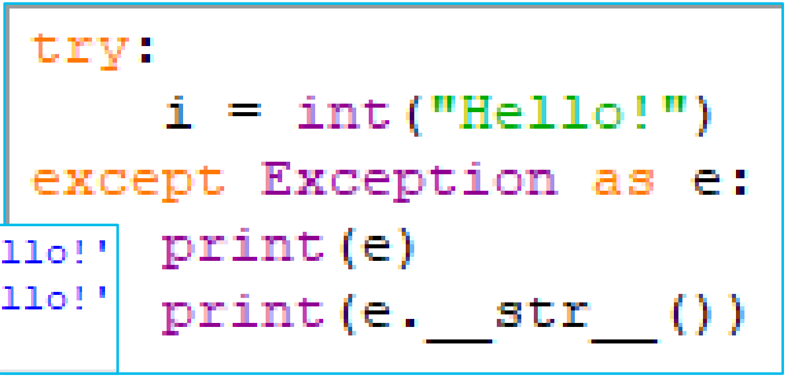


```
def reciprocal(n):
    try:
        n = 1 / n
    except ZeroDivisionError:
        print("Division failed")
        n = None
    else:
        print("Everything went fine")
    finally:
        print("It's time to say goodbye")
        return n


print(reciprocal(2))
print(reciprocal(0))
```

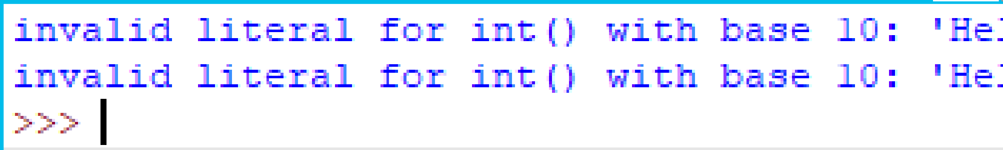
Exceptions are classes

Exceptions are classes.

The except statement is extended, and contains an additional phrase starting with the **as** keyword, followed by an identifier. The identifier is designed to catch the exception object so you can analyze its nature and draw proper conclusions.

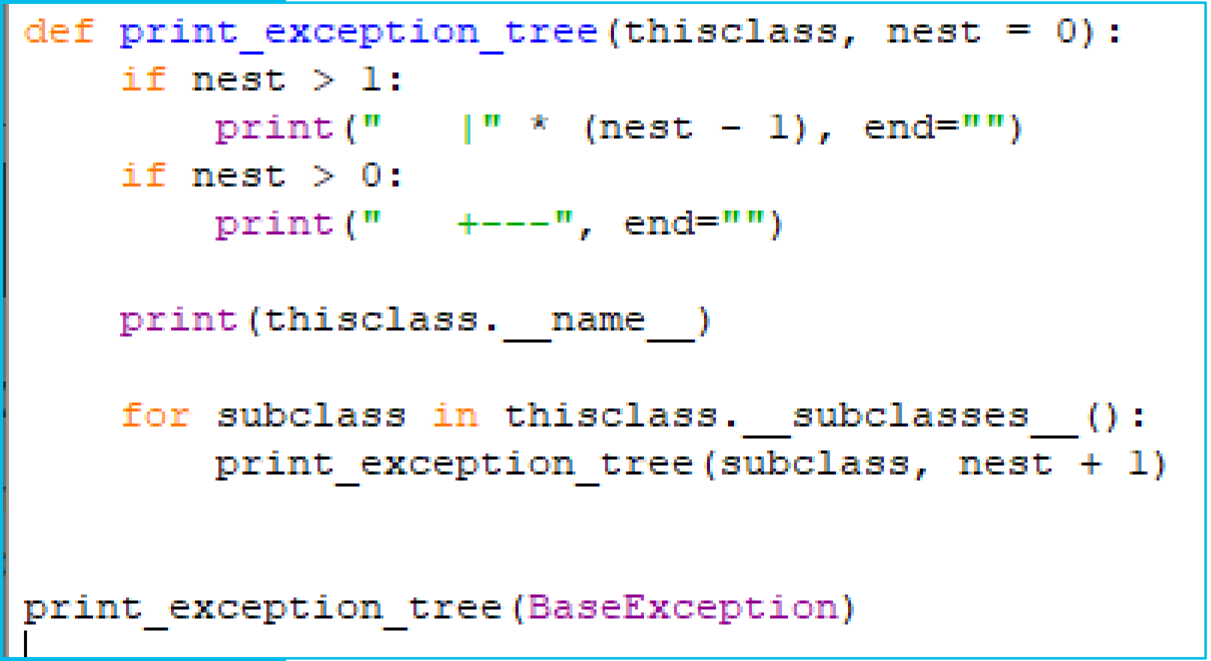


```
try:
    i = int("Hello!")
except Exception as e:
    print(e)
    print(e.__str__())
```



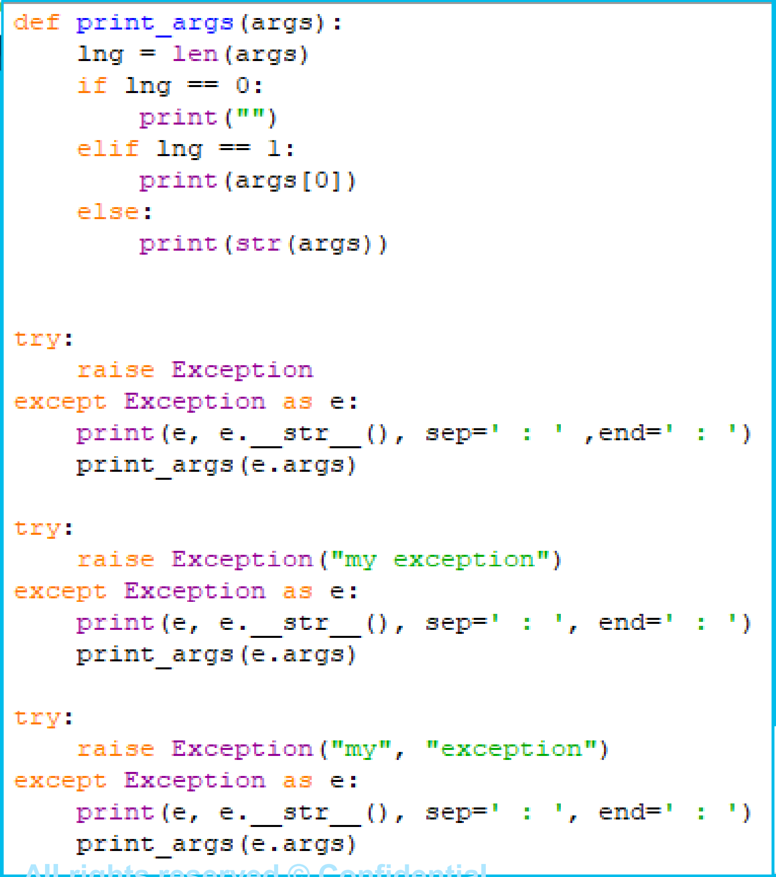
```
invalid literal for int() with base 10: 'Hello!'
invalid literal for int() with base 10: 'Hello!'
>>> |
```

Exceptions are classes



```
def print_exception_tree(thisclass, nest = 0):
    if nest > 1:
        print("   |" * (nest - 1), end="")
    if nest > 0:
        print("   +---", end="")

    print(thisclass.__name__)

    for subclass in thisclass.__subclasses__():
        print_exception_tree(subclass, nest + 1)


print_exception_tree(BaseException)
```

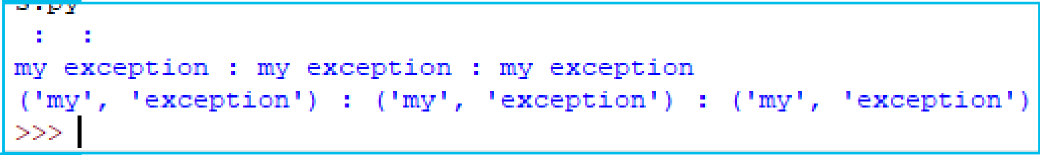
Detailed anatomy



```
def print_args(args):
    lng = len(args)
    if lng == 0:
        print("")
    elif lng == 1:
        print(args[0])
    else:
        print(str(args))


try:
    raise Exception
except Exception as e:
    print(e, e.__str__(), sep=' : ' ,end=' : ')
    print_args(e.args)

try:
    raise Exception("my exception")
except Exception as e:
    print(e, e.__str__(), sep=' : ', end=' : ')
    print_args(e.args)

try:
    raise Exception("my", "exception")
except Exception as e:
    print(e, e.__str__(), sep=' : ', end=' : ')
    print_args(e.args)
```

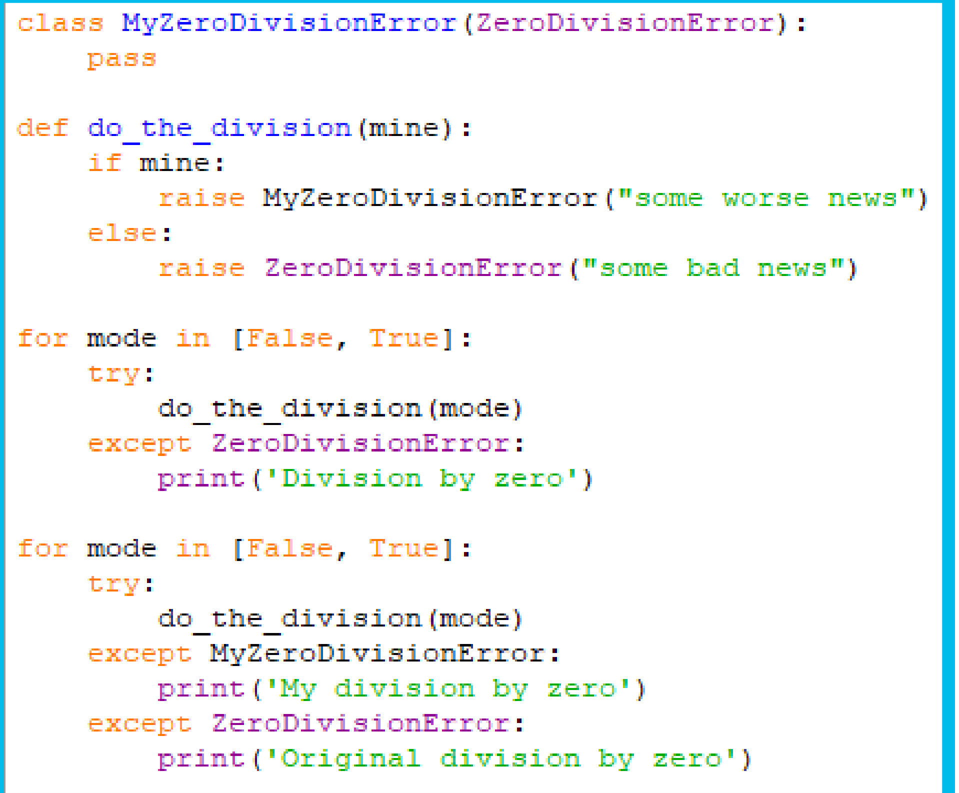
The BaseException class introduces a property named **args**. It's a tuple designed to gather all arguments passed to the class constructor. It is empty if the construct has been invoked without any arguments, or contains just one element when the constructor gets one argument (we don't count the self argument here), and so on.

I've prepared a simple function to print the args property in an elegant way.



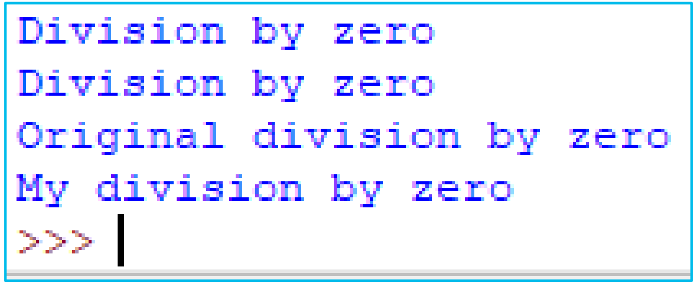
```
s.py
 :  :
my exception : my exception : my exception
('my', 'exception') : ('my', 'exception') : ('my', 'exception')
>>> 
```

How to create your own exception



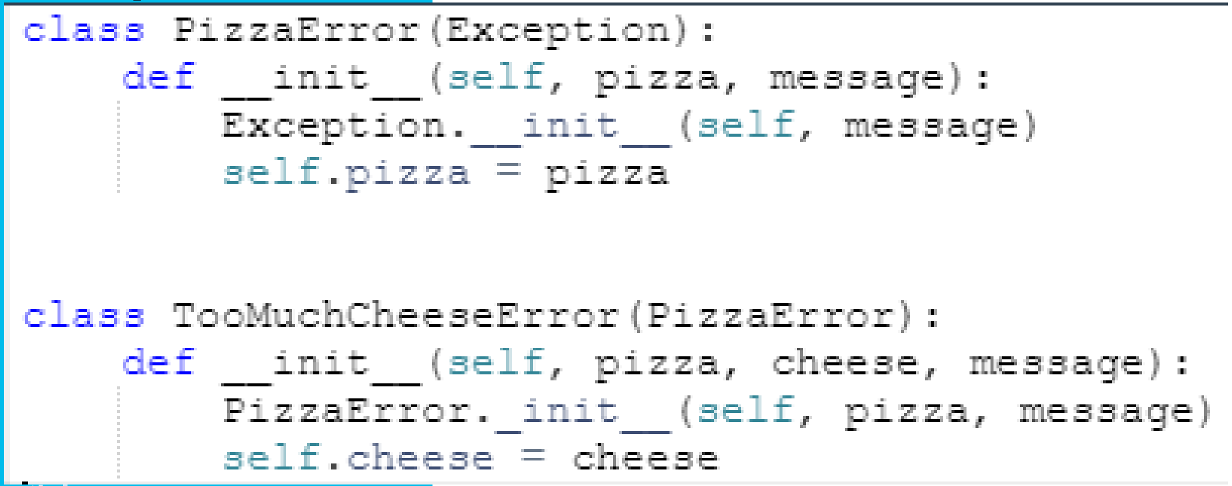
```
class MyZeroDivisionError(ZeroDivisionError):
    pass

def do_the_division(mine):
    if mine:
        raise MyZeroDivisionError("some worse news")
    else:
        raise ZeroDivisionError("some bad news")

for mode in [False, True]:
    try:
        do_the_division(mode)
    except ZeroDivisionError:
        print('Division by zero')

for mode in [False, True]:
    try:
        do_the_division(mode)
    except MyZeroDivisionError:
        print('My division by zero')
    except ZeroDivisionError:
        print('Original division by zero')
```

Note: if you want to create an exception which will be utilized as a specialized case of any built-in exception, derive it from just this one. If you want to build your own hierarchy, and don't want it to be closely connected to Python's exception tree, derive it from any of the top exception classes, like Exception.



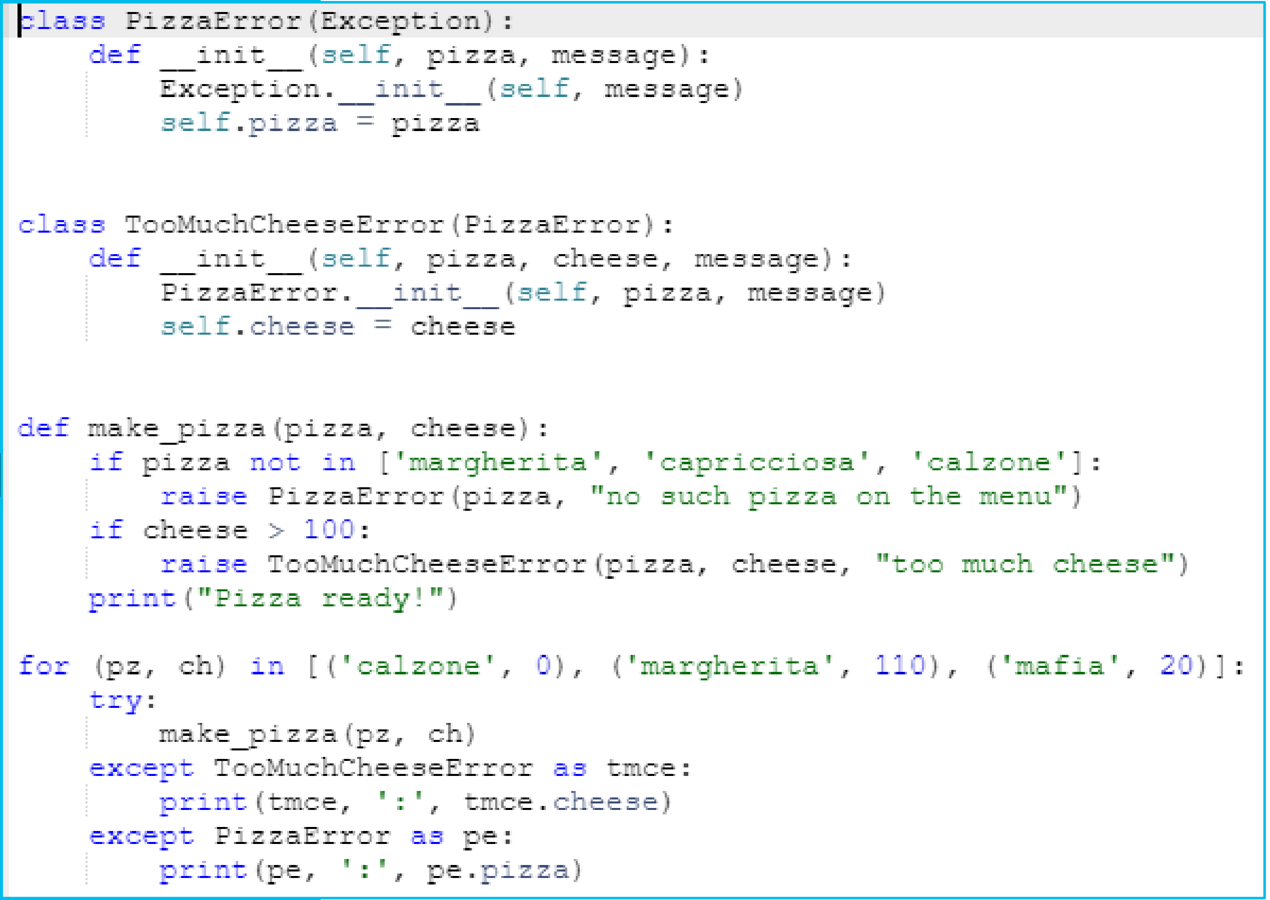
```
Division by zero
Division by zero
Original division by zero
My division by zero
>>> 
```

***How to create your own exception: continued

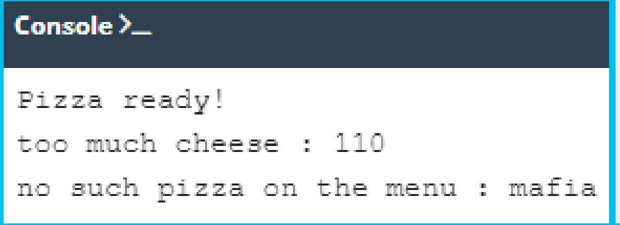


```
class PizzaError(Exception):
    def __init__(self, pizza, message):
        Exception.__init__(self, message)
        self.pizza = pizza


class TooMuchCheeseError(PizzaError):
    def __init__(self, pizza, cheese, message):
        PizzaError._init__(self, pizza, message)
        self.cheese = cheese
```

How to create own exception continued

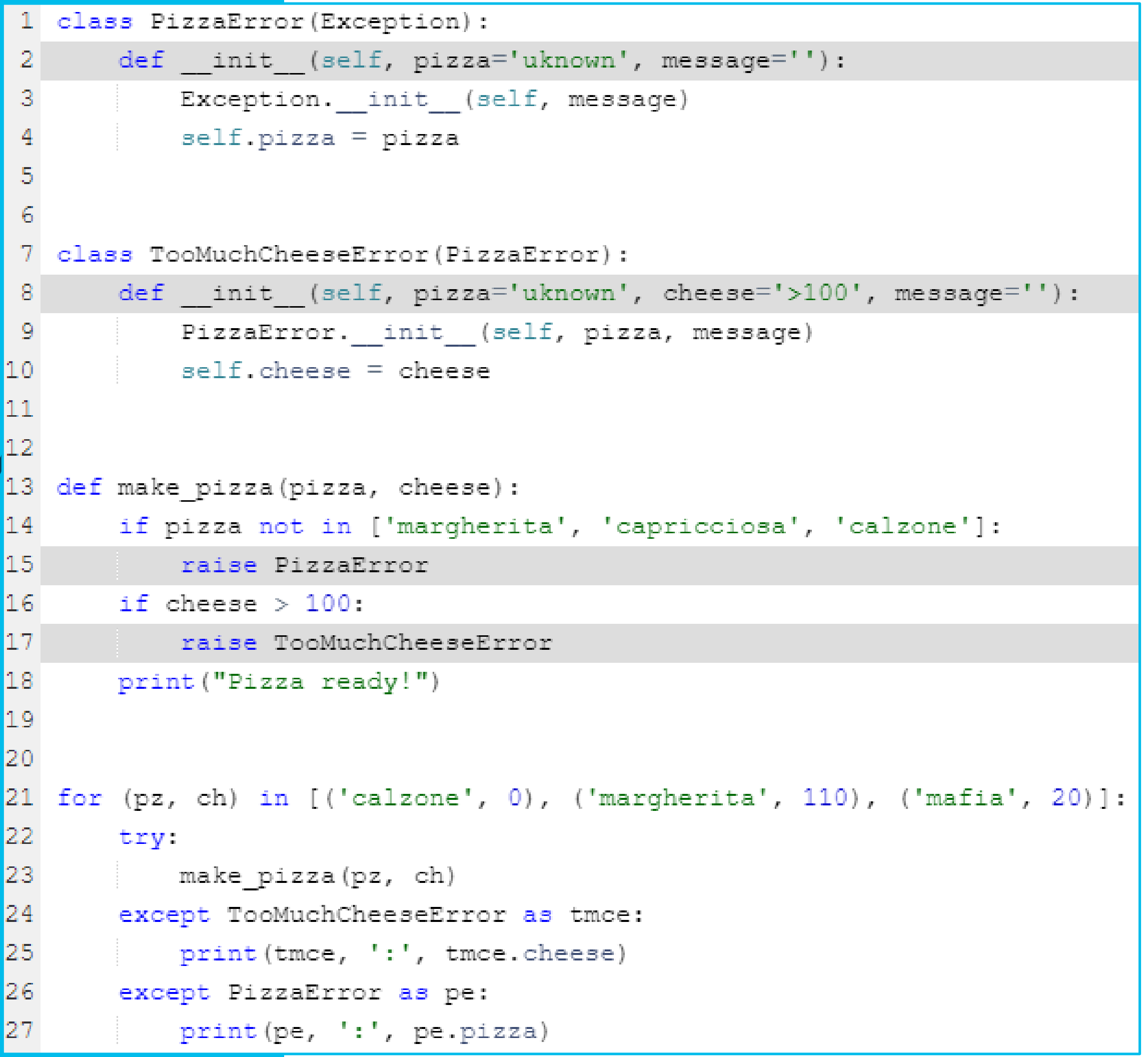


```
class PizzaError(Exception):
    def __init__(self, pizza, message):
        Exception.__init__(self, message)
        self.pizza = pizza


class TooMuchCheeseError(PizzaError):
    def __init__(self, pizza, cheese, message):
        PizzaError.__init__(self, pizza, message)
        self.cheese = cheese


def make_pizza(pizza, cheese):
    if pizza not in ['margherita', 'capricciosa', 'calzone']:
        raise PizzaError(pizza, "no such pizza on the menu")
    if cheese > 100:
        raise TooMuchCheeseError(pizza, cheese, "too much cheese")
    print("Pizza ready!")

for (pz, ch) in [('calzone', 0), ('margherita', 110), ('mafia', 20)]:
    try:
        make_pizza(pz, ch)
    except TooMuchCheeseError as tmce:
        print(tmce, ':', tmce.cheese)
    except PizzaError as pe:
        print(pe, ':', pe.pizza)
```

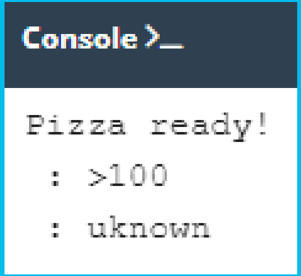


```
Pizza ready!
too much cheese : 110
no such pizza on the menu : mafia
```

How to create y
own exception:
continued

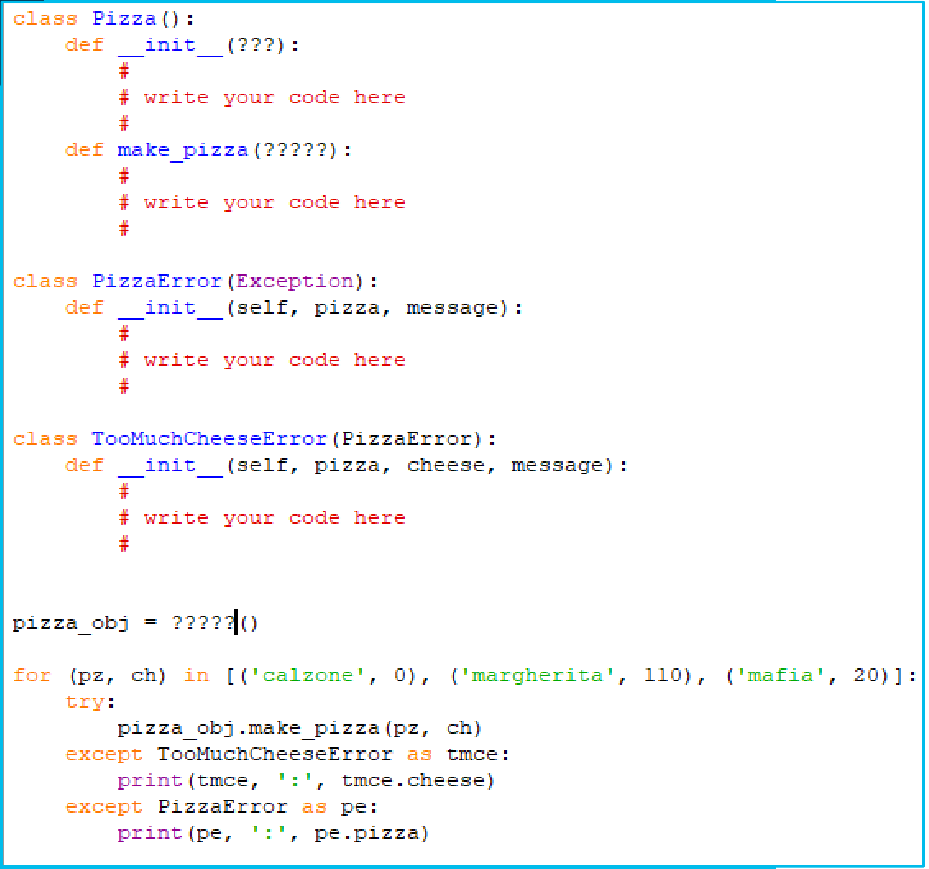


```python
class PizzaError(Exception):
    def __init__(self, pizza='uknown', message=''):
        Exception.__init__(self, message)
        self.pizza = pizza


class TooMuchCheeseError(PizzaError):
    def __init__(self, pizza='uknown', cheese='>100', message=''):
        PizzaError.__init__(self, pizza, message)
        self.cheese = cheese


def make_pizza(pizza, cheese):
    if pizza not in ['margherita', 'capricciosa', 'calzone']:
        raise PizzaError
    if cheese > 100:
        raise TooMuchCheeseError
    print("Pizza ready!")


for (pz, ch) in [('calzone', 0), ('margherita', 110), ('mafia', 20)]:
    try:
        make_pizza(pz, ch)
    except TooMuchCheeseError as tmce:
        print(tmce, ':', tmce.cheese)
    except PizzaError as pe:
        print(pe, ':', pe.pizza)
```



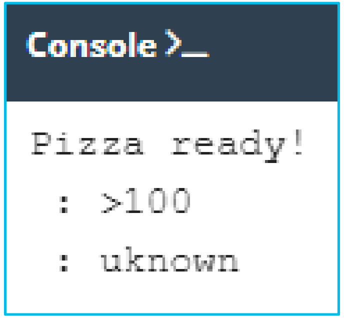
```
Pizza ready!
 : >100
 : uknown
```

More practice

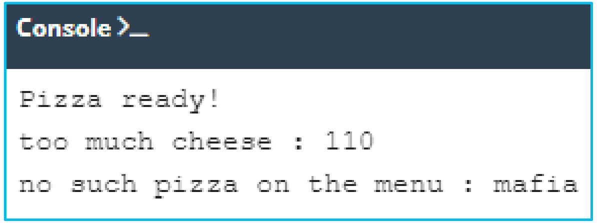


```python
class Pizza():
    def __init__(???):
        #
        # write your code here
        #
    def make_pizza(?????):
        #
        # write your code here
        #

class PizzaError(Exception):
    def __init__(self, pizza, message):
        #
        # write your code here
        #

class TooMuchCheeseError(PizzaError):
    def __init__(self, pizza, cheese, message):
        #
        # write your code here
        #


pizza_obj = ?????()

for (pz, ch) in [('calzone', 0), ('margherita', 110), ('mafia', 20)]:
    try:
        pizza_obj.make_pizza(pz, ch)
    except TooMuchCheeseError as tmce:
        print(tmce, ':', tmce.cheese)
    except PizzaError as pe:
        print(pe, ':', pe.pizza)
```

- create class Pizza
- list of pizza - private in constructor
- write method like make_a_pizza

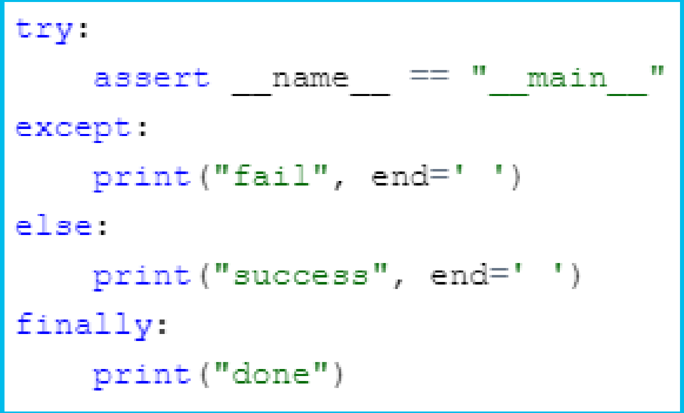


```
Pizza ready!
 : >100
 : uknown
```



```
Pizza ready!
too much cheese : 110
no such pizza on the menu : mafia
```

Key takeaways



```
try:
    assert __name__ == "__main__"
except:
    print("fail", end=' ')
else:
    print("success", end=' ')
finally:
    print("done")
```

1. **The else:** branch of the try statement is executed when there has been no exception during the execution of the try: block.

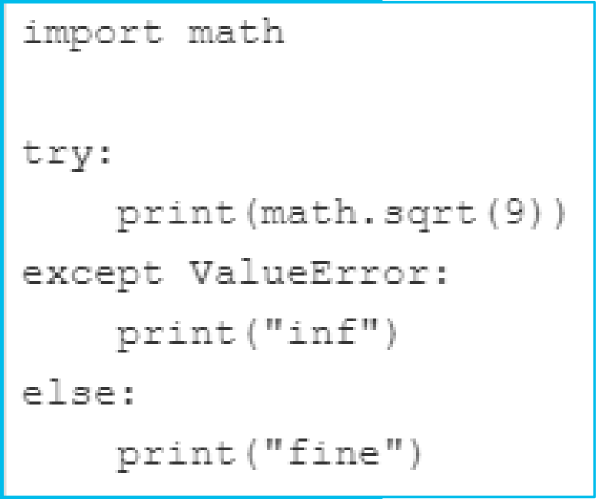
2**. The finally:** branch of the try statement is always executed.

3. The syntax except

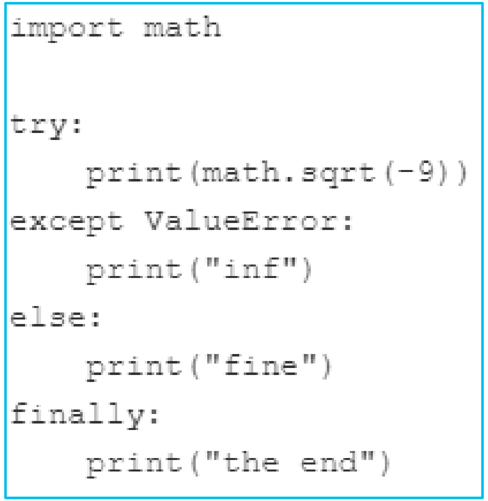
Exception_Name as exc_obj: lets you intercept an object carrying information about a pending exception. The object's property named **args (a tuple) stores all arguments passed to the object's constructor.**

4. The exception classes can be extended to enrich them with new capabilities, or to adopt their traits to newly defined exceptions. For example:

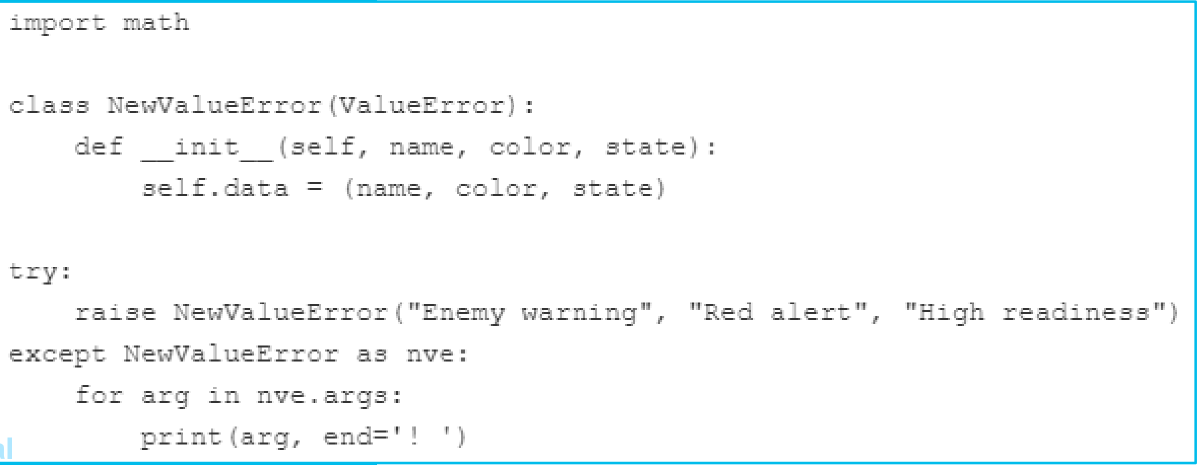
Examples



```
import math


try:
    print(math.sqrt(9))
except ValueError:
    print("inf")
else:
    print("fine")
```



```
import math


try:
    print(math.sqrt(-9))
except ValueError:
    print("inf")
else:
    print("fine")
finally:
    print("the end")
```



```
import math

class NewValueError(ValueError):
    def __init__(self, name, color, state):
        self.data = (name, color, state)

try:
    raise NewValueError("Enemy warning", "Red alert", "High readiness")
except NewValueError as nve:
    for arg in nve.args:
        print(arg, end='! ')
```

ЗАДАНИЯ

1) Прорешать всю классную работу
2) Выполнить все домашние задания

Почитать:

1) Byte of Python - стр.108-120

Крайний срок сдачи 17/10 в 21:00 (можно раньше, но не позже)

https://docs.python-guide.org/writing/structure/

ЗАДАНИЯ

Название файлов, которые вы отправляете мне в telegram:

Vasia_Pupkin_class_work_L10_P2.py

Без классной работы домашнее не принимается на проверку

Vasia_Pupkin_L10_2_1_que.py

Формат сообщения которое вы присылаете мне

(после полного выполнения домашнего задания, только один раз) в Telegram:

Добрый день/вечер. Я Вася Пупкин, и это мои домашние задания к лекции 10 часть 2 про ООП и исключения.

Крайний срок сдачи 17/10 в 21:00 (можно раньше, но не позже)

https://docs.github.com/articles/using-pull-requests

Q&A

Create your possibilities. Bye bye.