Object Oriented Programming

Nearly all of the programs and techniques you have used until now fall under the procedural style of programming. Admittedly, you have made use of some build-in objects, but when referring to them, it's just mentioned the absolute minimum.

The procedural style of programming was the dominant approach to software developments for decades of IT, and it is still in use today. Moreover, it isn't going to disappear in the future, as it works very well for specific types of projects (generally, not very complex ones and not large ones, but there are lots of exceptions to that rule).

The object approach is quite young (much younger than the procedural approach) and is particularly useful when applied to big and complex projects carried out by large teams consisting of many developers.

This kind of understanding of a project's structure makes many important tasks easier, for example, dividing the project into small, independent parts, and independent development of different project elements.

Python is a universal tool for both object and procedural programming, meaning that it may be successfully utilized in both spheres.

Furthermore, you can create lots of useful applications, even if you know nothing about classes and objects, but you have to keep in mind that some of the problems (for example graphical user interface handling) may require a strict object approach.

Fortunately, object programming is relatively simple.

Procedural vs Object-oriented Approach

In the procedural approach, it's possible to distinguish two different and completely separate worlds: the world of **data** and the world of **code**.

The world of **data** is populated with **variables** of different kinds, while the world of **code** is inhabited by code grouped into **modules** and **functions**.

Functions are able to use (and abuse as well) data, but not vice versa. (For example, when a sine function gets a bank account balance as a parameter).

Data cannot use functions, but they can use methods, functions which are invoked from within the data, not beside them.

The **object** approach suggests a completely different way of thinking. The data and the code are enclosed together in the same world, divided into **classes**.

Every **class** is like a recipe which can be used when you want to create a useful object (this is where the name of approach comes from). You may produce as many objects as you need to solve your problem.

Every object has a set of traits (properties or attributes) and is able to perform a set of activities (methods).

The recipes may be modified if they are inadequate for specific purposes and, in effect, new classes may be created. These new classes inherit properties and methods from the originals, and usually add some new ones, creating new, more specific tools.

Objects are incarnations of ideas expressed in classes, like a cheesecake on your plate is an incarnation of the idea expressed in a recipe printed in a cookbook.

The objects interact with each other, exchanging data or activating their methods. A properly constructed class (and thus, its objects) are able to protect the sensible data and hide if from unauthorized modifications.

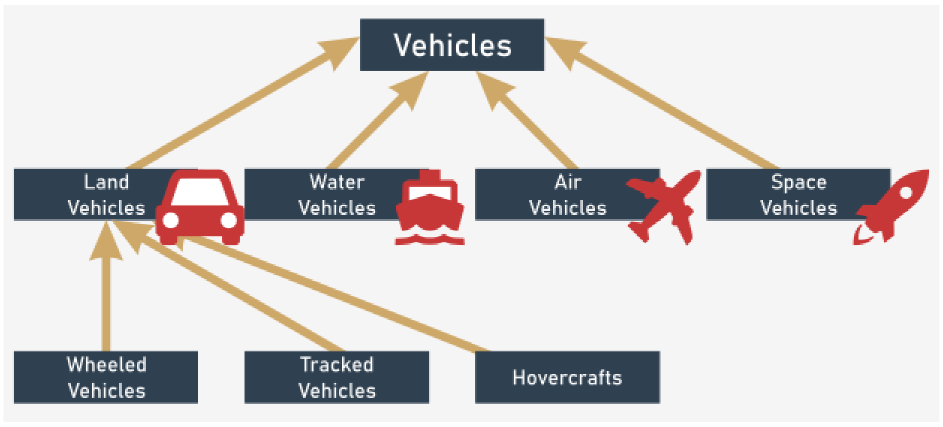
There is no clear border between data and code, they live as one in objects.

All these concepts are not as abstract as you may first suspect. On the contrary, they all are taken from real-life experiences, and therefore are extremely useful in computer programing. They don't create artificial life, they reflect real facts, relationships and circumstances.

Class hierarchies

Class here that we are concerned with is like a category, as a result of precisely defined similarities.

Image source: cisco/Python institute



Let's define "vehicles", object that moves? Well, dogs move too, so we need a better definition.

"Vehicles are artificially created entities used for transportation, moved by forces of nature, and directed (driven) by humans." Based on this definition, dogs aren't vehicles.

The *vehicles* class is very broad, so we have to define some more **specialized classes** (subclasses). The *vehicles* class will be a superclass for them all.

Note that the hierarchy grows from top to bottom. The most general, and the widest class is always at the top (the superclass, *vehicles* in this case) while its descendants are located below (the subclasses).

In the example above. The superclass $vehicles$ has four subclasses:

- Land vehicles
- Water vehicles
- Air vehicles
- Space vehicles

In this example, we'll discuss the $land\ vehicles$ subclass only. If you wish, you can continue with the remaining classes.

As you can see, $land\ vehicles$ are further divided into $wheeled\ vehicles$, $tracked\ vehicles$ and $hovercrafts$. The hierarchy we've created is illustrated by the figure.

Note the direction of the arrows, they always point to the superclass. The top-level class is an exception, it doesn't have its own superclass.

Another example will be animals, we say that all $animals$ (our top-level class) can be divided into 5 subclasses:

- Mammals
- Reptiles
- Birds
- Fish
- Amphibians

Let’s dig deeper into $Mammals$. We have identified the following subclasses, $wild\ animals$ and $domesticated\ mammals$.

Try to extend the hierarchy anyway you want, and find the right place for humans.

What is an object?

A class (among other definitions) is a **set of objects**. An object is **a being belonging to a class**.

An object is an incarnation of the requirements, traits, and qualities assigned to a specific class. This may sound simple, but note the following important circumstances. Classes form a hierarchy.

This may mean that an object belonging to a specific class belongs to all the superclasses at the same time. It may also mean that any object belonging to a superclass may not belong to any of its subclasses.

For example: any personal car is an object belonging to the *wheeled vehicles* class. It also means that the same car belongs to all superclasses of its home class, therefore it is a member of the *vehicles* class too. Your dog (or your cat) is an object included in the domesticated $mammal$ class, which means that it is included in the $animals$ class as well.

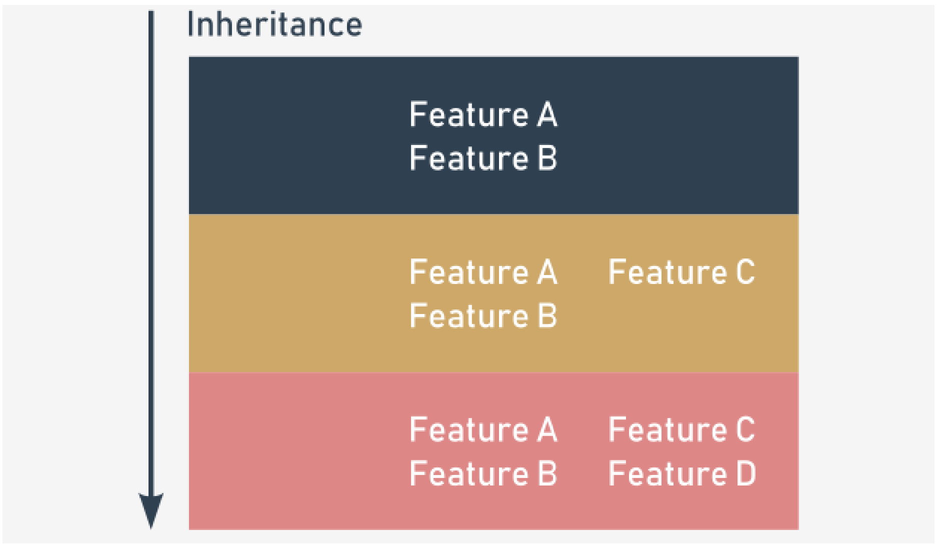
Each subclass is more specialized (or more specific) than its superclass. Conversely, each superclass is more general (more abstract) than any of its subclasses.

Note that we’ve presumed that a class may only have one superclass, which is not always true, but we’ll discuss this issue more a bit later.

Inheritance

One of the fundamental concepts of object programming is **inheritance**. Any object bound to a specific level of a class hierarchy **inherits ALL** the traits (as well as requirements and qualities) defined inside any of the **superclasses**. This means that traits in a superclass is a **subset** of those in a subclass.

Image source: cisco/Python Institute



What does an object have?

The object programming convention assumes that every existing object may be equipped with three groups of attributes.

- An object has a **name** that uniquely identifies it within its home namespace (although there may be some anonymous objects too)

- An object has a **set of individual properties** which makes it original unique or outstanding (although it's possible that some objects may have no properties at all)
- An object has a **set of abilities to perform specific activities**, able to change the object itself, or some of the other objects.

There is a hint (although this doesn't always work) which can help you identify any of the three spheres above. Whenever you describe an object and you use:

- A noun, you probably define the object's name
- An adjective, you probably define the object's property
- A verb, you probably define the object's activity.

Two example phrases should serve as a good example:

- A red Ferrari went quickly
  Object name: Ferrari
  Home Class: Wheeled vehicles
  Property: color (red)
  Activity: go (quickly)
- Rudolph is a small kitten who sleeps all day (no coloring because I'm lazy)
  Object name: Rudolph
  Home class: kitten
  Property: size (small)
  Activity: sleep (all day)

Writing your own class

Object programming is the art of defining and expanding classes. A class is a model of a very specific part of reality, reflecting properties and activities found in the real world.

The classes defined at the beginning are too general and imprecise to cover the largest possible number of real cases.

There's no obstacle to defining new, more precise subclasses. They will inherit everything from their superclass, so the work that went into its creation isn't wasted.

The new class may add new properties and new activities, and therefore may be more useful in specific applications. Obviously, it may be used as a superclass for any number of newly created subclasses.

The process doesn't need to have an end. You can create as many classes as you need.

The class you define has nothing to do with the object: **the existence of a class does not mean that any of the compatible objects will automatically be created**. The class itself isn't able to create an object, you have you create it yourself, and Python allows you to do this.

```
class sample:
    pass
```

Here is the simplest class, but its rather poor, it has neither properties nor activities. It's empty.

The definition begins with the keyword *class*. The keyword is followed by an identifier which will **name** the class (don't confuse it with the object's name, these are two different things).

Next, you add a **colon**, as classes, like functions, form their own nested block. The content inside the block defines all the class's properties and activities.

The *pass* keyword fills the class with **nothing**. It doesn't contain any methods or properties.

The newly defined class becomes a tool that is able to create new objects. The tool has to be used explicitly, on demand.

Imagine that you want to create one (exactly one) object of the *sample* class.

To do this, you need to assign a variable to store the newly created object of that class, and create an object at the same time.

```
some_object = sample()
```

Note these:

- the class name tries to pretend that it's a function, we'll discuss it soon.
- the newly created object is equipped with everything the class brings; as this class is completely empty, the object is empty, too.

The act of creating an object of the selected class is also called an **instantiation** (as the object becomes an instance of the class).

Stack [Stack (abstract data type) - Wikipedia]

A stack is a **data structure** developed to store data in a very specific way.

Imagine a stack of coins. You aren't able to put a coin anywhere else but on the top of the stack. Similarly, you can't get a coin off the stack from any place other than the top of the stack. If you want to get the coin that lies on the bottom, you have to remove all the coins from the higher levels.

A stack's behavior is described as **L**ast **I**n **F**irst **O**ut (LIFO) or **F**irst **I**n **L**ast **O**ut (FILO).

A stack is an object with **two elementary operations**, *conventionally* named **push** (put new element on the top) and **pop** (take an element away from the top).

This data structure is used very often in many classical algorithms, for example DFS (**D**epth **F**irst **S**earch). It can also be used to solve many problems too, like the Josephus problem.

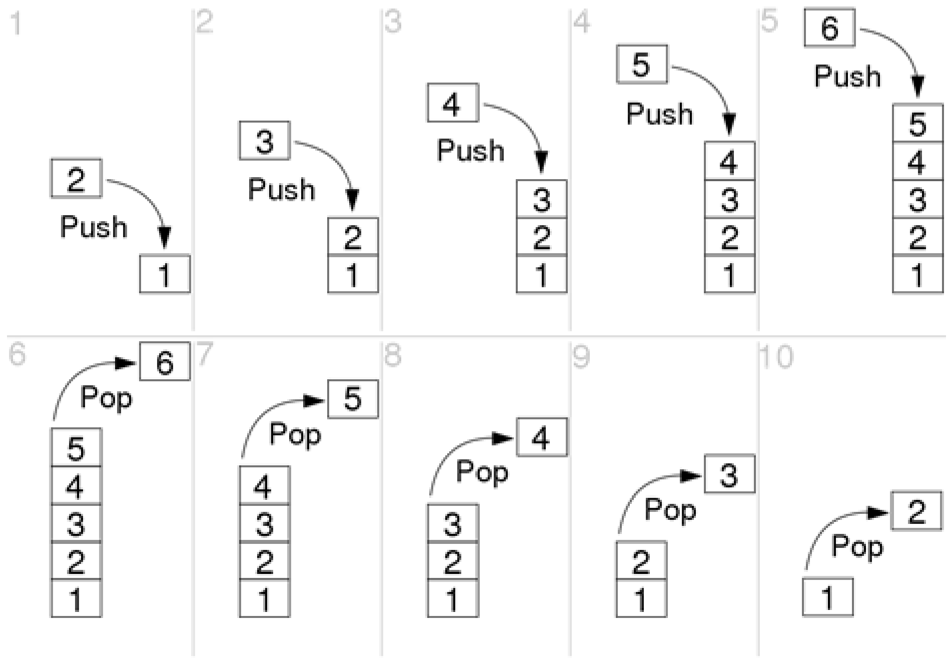


Illustration of a stack (image source: Wikipedia Stack (abstract data type) - Wikipedia)

```
stack = []
def push (val):
    stack.append(val)
def pop ():
    del stack[-1]
def last ():
    return stack[-1]
```

The code right here simulates a stack.

Stack-the PROCEDURAL approach vs the OBJECT-ORIENTED approach

The procedural stack is ready. Of course, there are some weaknesses.

1. The essential variable (the *stack* list) is highly **vulnerable**. Anyone can modify it in an uncontrollable way. It may happen as a result of carelessness.
2. It may also happen that one day you may need more than one stack. You'll have to create another list for the stack's storage, and probably other *push* and *pop* functions too.

The objective approach delivers solutions for each of the problems above.

1. It has the ability to hide (protect) selected values against unauthorized access. It's called encapsulation (Encapsulation (computer programming) - Wikipedia). The encapsulated values can be neither accessed nor modified if you want to use them exclusively.
2. When you have a class implementing all the needed stack behaviours, you can produce as many stacks as you want, you needn't copy or replicate any part of the code.

It also has the ability to enrich the stack with new functions comes from inheritance, you can create a new class (a subclass) which inherits all the existing traits from the superclass, and adds some new ones.

Stack-OOP

First of all, before writing a *class* ourself, you have to know about a function called *constructor*. Its name **must always be** *__init__*, it has to have **at least one** parameter. The obligatory parameter is usually named *self*, it's only a convention, but you should follow it - it simplifies the process of reading and understanding your code.

This code here creates an empty list named *stack_list* under **each** class *Stack*.

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

stack = Stack()
print(len(stack.stack_list)) # 0
```

The dotted notation is used here, just like when invoking methods. This is the general convention for accessing an object's properties, you need to name the object, put a dot (.) after it, and specify the desired property's name. Note that you shouldn't put any paratheses there, it's a property after all, not a method.

Adding two underscores before the property makes it become private.

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

stack = Stack()
print(len(stack.__stack_list)) # AttributeError
```

You can't see it from the outside world. This is how Python implements the **encapsulation** concept.

Public components cannot start with two (or more) underscores. There is one more requirement, the name **must not have no more than one** trailing underscore.

Here is a sample of $stack$, in object-oriented programming.

```
class Stack:
    def __init__ (self):
        self.__stack_list = []
    def push (self, val):
        self.__stack_list.append(val)
    def pop (self):
        del self.__stack_list[-1]
    def top (self):
        return self.__stack_list[-1]

stack = Stack()
```

All methods have to have the $self$ parameter. It allows the method to access entities (properties and activities/methods) carried out by the actual object. You cannot omit it. Every time Python invokes a method, it implicitly sends the current object as the first argument.

The way that methods are invoked is illustrated below:

The first stage delivers the object as a whole ($self$), next you get to the $self.\_\_stack\_list$ list, then you perform the third and last step $self.\_\_stack\_list.append(val)$.

```
class Stack:
    ……

s1 = Stack()
s2 = Stack()
```

Here two stacks are created, note that they work independently.

Here you can see a new class $AddingStack$, the third line in it ($Stack.\_init\_(self)$) creates a Stack and uses it.

```python
class Stack:
    def __init__ (self):
        self.__stack_list = []
    def push (self, val):
        self.__stack_list.append(val)
    def pop (self):
        del self.__stack_list[-1]
    def top (self):
        return self.__stack_list[-1]

class AddingStack (Stack):
    def __init__(self):
        Stack.__init__(self)
        self.__sum = 0
    def push (self, val):
        self.__sum += val
        Stack.push(self, val)
    def pop (self):
        self.__sum -= Stack.top(self)
        Stack.pop()
    def get_sum (self):
        return self.__sum

s1 = Stack()
s2 = Stack()
```

Queue [Queue (abstract data type) - Wikipedia]

Queue is a FIFO data structure, read the Wikipedia page for further understanding, the code is left as an exercise.

__dict__

This method can let you have a peer at the class.

Instance variables

In general, a class can be equipped with two different kinds of data to form a class's properties.

One kind of class property exists when and only when it is explicitly created and added to an object. As you already know, this can be done during the object's initialization, performed by the constructor.

Some important consequences of the above approach:

- Different objects of the same class may possess different sets of properties.
- There must be a way to safely check if a specific object owns the property you want

to utilize (unless you want to provoke an exception)
- Each object carries its own set of properties - they don't interfere with one another in any way.

Such variables (properties) are called instance variables.

The word *instance* suggests that they are closely connect to the objects (which are class instances), not to the class themselves.

There is one conclusion that should be stated here: modifying an instance variable of any object has no impact on all the remaining objects. Instance variables are perfectly isolated from each other.

Class variables

A class variable is a property which exists in just one copy and is stored outside any object.

Note: no instance variable exists if there is no object in the class; a class variable exists in one copy even if there are no objects in the class.

```python
class Example:
    counter = 0
    def __init__ (self):
        Example.counter += 1

a = Example()
b = Example()

print(a.counter) # 2
```

Class variables aren't shown in an object's *__dict__*, but you can always try to look into the variable of the same name, but at the class level.

A class variable always presents the same value in all class instances (objects).

Check if a certain attribute exists in a class

Python provides a function which is able to safely check if any object/class contains a specified property. The function is named *hasattr*, and expects two arguments to be passed to it:

1. The class/object being checked
2. The name of the property whose existence has to be reported (string)

```python
class Example:
    def __init__ (self, val):
        if val % 2 == 0:
            self.lol = 1;
        else:
            self.bruh = 1;

a = Example(1)

if hasattr (a, "bruh"):
    print("bruh exists") # bruh exists
if hasattr (a, "lol"):
    print("lol exists")
```
The *hasattr* can also check for class variables.

Methods in class

A method is a function embedded inside a class.

There is one fundamental requirement, a method is obliged to have at least one parameter (and again, it's the parameter *self*).

The *self* parameter is used to obtain access to the object's instance and class variables. It is also used to invoke other object/class methods from inside the class.

```
class Example:
    def dunno (self):
        print("hihi")

    def method (self):
        print("Method")
        self.dunno()

a = Example()
a.method()
# Method
# hihi
```

As mentioned beforehand, the __*init*__ method isn't just a regular method, it will be a constructor.

The following code demonstrates hidden methods and how to activate them:

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

    def __hidden(self):
        print("hidden")

obj = Example()
obj.visible()

try:
    obj.__hidden()
except:
    print("failed") # failed

obj._Example__hidden() # hidden
```

__name__

Another built-in property besides _*dict*_ is _*name*_, which is a string.

The property contains the name of the class.

```python
class Example:
    pass

print(Example.__name__) # Example
obj = Example()
print(type(obj).__name__) # Example
print(obj.__name__) # AttributeError
```

__module__

It returns the name of the module which contains the definition of the class.

```python
class Classy:
    pass

print(Classy.__module__) # __main__
obj = Classy()
print(obj.__module__) # __main__
```

__base__

Returns a tuple, contains classes (not class names) which are direct superclasses for the class. The order is the same as that used inside the class definition.

I'll show you only a very basic example, as I want to highlight how inheritance works.

Note that only classes have this attribute, objects don't.

```python
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) # ( object )
printBases(SuperTwo) # ( object )
printBases(Sub) # ( SuperOne SuperTwo )
```

Reflection and introspection

All these means allow the Python programmer to perform two important activities specific to many objective languages.

- **Introspection**, which is the ability of a program to examine the type or properties of an object at runtime.
- **Reflection**, which goes a step further, and is the ability of a program to manipulate the values, properties and/or functions of an object at runtime.

In other words, you don't have to know a complete class/object definition to manipulate the object, as the object and/or its class contain the metadata allowing you to recognize its features during program execution.

Some attributes worth mentioning/restating

__dict__, *__name__*, *__module__*, *__bases__*, *getattr*(), *isinstance*(), *setattr*().

They are pretty straightforward so I'm not going to dig deep into them here.

Another method for a class to introduce itself is like this.

```python
class Example:
    pass
a = Example()
print(a) # <__main__.Example object at 0x7fdf8a4f1ef0>
```

If you run the code on your computer, you'll see something very similar, although the hexadecimal number will be different, as it's just an internal object identifier used by Python.

When Python needs any class/object to be presented as a string (putting an object as an argument in the *print*() function invocation fits this condition) it tries to invoke a method named *__str__*() from the object and to use the string it returns.

The default *__str__*() returns the previous string, not so good right? You can change it just by defining your own method of the name.

```python
class Example:
    def __init__ (self):
        self.first_name = "Potter"
        self.last_name = "Harry"
    def __str__ (self):
        return self.last_name + ' ' + self.first_name
a = Example()
print(a) # Harry Potter
```

Inheritance

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

In other words, inheritance is a way of **building a new class, not from scratch, but by using an already defined repertoire of traits**. The new class inherits (and this is the key) all the already existing equipment, but is able to add some new ones if needed.

Thanks to that, it is possible to build more specialized (and more concrete) classes using some sets of predefined general rules and behavious.

```python
class Vehicle:
    pass

class LandVehicle(Vehicle):
    pass

class TrackedVehicle(LandVehicle):
    pass
```

The above code is a very simple example of two-level inheritance.

The $Vehicle$ class is the superclass for both the $LandVehicle$ and $TrackedVehicle$ classes.

The $LandVehicle$ class is a superclass for $TrackedVehicle$.

The $LandVehicle$ class and $TrackedVehicle$ class are subclasses of $Vehicle$.

The $TrackedVehicle$ class is a subclass of $LandVehicle$.

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:

$$issubclass(ClassOne, ClassTwo)$$

It returns a Boolean value.

```python
import yaml

class Vehicle:
    pass

class LandVehicle(Vehicle):
    pass

class TrackedVehicle(LandVehicle):
    pass

for x in [Vehicle, LandVehicle, TrackedVehicle]:
    for y in [Vehicle, LandVehicle, TrackedVehicle]:
        print(issubclass(x, y), end="\t")
    print()

# True  False   False
# True  True    False
# True  True    True
```

And I believe you can understand the code here.

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

The function $isinstance()$ detects whether the object is an instance of the class, and returns a Boolean answer.

The *is* operator

It refers directly to objects, it is used as below:

```
object_one is obejct_two
```

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.

Assigning a value of an object variable to another variable doesn't copy the object, but only its handle. This is why an operator like *is* may be very useful in particular circumstances.

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

one = SampleClass(0)
two = SampleClass(2)
three = one
three.val += 1

print(one is two) # False
print(two is three) # False
print(three is one) # True
print(one.val, two.val, three.val) # 1 2 1
```

three is one is true as in line 7 ($three = one$) we assigned three as one.

```
from re import A
a = "Mary had a little "
b = "Mary had a little lamb"
a += "lamb"

print(a == b, a is b) # True False
```

This shows that even if the objects have same content, they are still different objects.

How Python finds properties and methods

Consider the code here:

```
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("ML7")

print(obj) # My name is ML7.
```

The Sub class defines its own constructor, which invokes the one from the superclass. How it's done is $Super.\_\_init\_\_(self, name)$. We have then instantiated one object of class Sub and printed it.

Note: As there is no $\_\_str\_\_()$ method within the Sub class, the printed string is to be produced within the $Super$ class. This means that the $\_\_str\_\_()$ method has been inherited by the Sub class.

Here is another way to do the same thing

```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("ML7")

print(obj) # My name is ML7.
```

In this example, we make use of the $super()$ function, which accesses the superclass without needing to know its name.

Note: You can use this mechanism not only to invoke the superclass constructor, but also to get access to any of the resources available inside the superclass.

```python
class Super:
    supVar = 1

class Sub(Super):
    subVar = 2

obj = Sub()

print(obj.subVar) # 2
print(obj.supVar) # 1
```

As you can see, the $Super$ class defines one class variable named $supVar$, and the Sub class defines a variable named $subVar$. Both these variables are visible inside the Sub class.

The same effect can be observed with **instance variables**.

```python
class Super:
    def __init__ (self):
        self.supVar = 11

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

```
obj = Sub()

print(obj.subVar) # 12
print(obj.supVar) # 11
```

The Sub class constructor creates an instance variable named $subVar$, while the $Super$ constructor does the same with a variable named $supVar$. As previously, both variables are accessible from within the object of class Sub.

Note: The existence of the $supVar$ variable is obviously conditioned by the $Super$ class constructor invocation. Omitting it would result in the absence of the variable in the created object.

It's now possible to formulate a **general statement** describing Python's behaviour.

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

1. Find it inside the object itself.
2. Find in in all classes involved in the object’s inheritance line from bottom to top.

If both of the above fails, an exception ($AttributeError$) is raised.

The first condition may need some additional attention. As you know, all objects deriving from a particular class may have different sets of attributes, and some of the attributes may be added to the object a long time after the object's creation.

Multiple inheritance

Multiple inheritance occurs when a class has more than one superclass. Here is an example:

```
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()) # 10 11
print(obj.var_b, obj.fun_b()) # 20 21
```

The Sub class has two superclasses: $SuperA$ and $SuperB$. This means that the Sub class inherits all the attributes offered by both $SuperA$ and $SuperB$.

Let’s analyse this example here

```
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
```

Both $Level1$ and $Level2$ classes define a method named $fun()$ and a property named var. Does this mean that the $Level3$ class object will be able to access two copies of each entity? Not at all.

The entity defined later (in the inheritance sense) overrides the same entity defined earlier. That's why the output is 200 201.

This feature can be intentionally used to modify default (or previously defined) class behaviours when any of its classes needs to act in a different way to its ancestor.

We can also say that Python looks for an entity from bottom to top.

But what if two classes are at the same level?

```
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
```

The Sub class inherits goods from two superclasses, $Left$ and $Right$.

There is no doubt that the class variable var_right comes from the $Right$ class, and var_left comes from $Left$ respectively.

As we can see from the output, we can conclude that **Python looks for object components** in the following order:

1. Inside the object itself.
2. In its superclasses, from bottom to top.

3. If there is more than one class on a particular inheritance path, Python scans them from left to right.

Now we swap $Left$ and $Right$.

```
...
class Sub (Right, Left):
    pass
...
```

The output is now:

```
R LL RR Right
```

More examples

```
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() # do_it from One
two.doanything() # do_it from Two
```

I believe that $one.doanything()$ is straightforward for you.

Our attention is on the second one, the second invocation will launch $do\_it()$ in the form existing inside the Two class, regardless of the fact that the invocation takes place within the One class.

The situation in which the subclass is able to **modify** its superclass behaviour 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 behaviour of the superclasses, is called **virtual**.

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

Single inheritance vs multiple inheritance

- 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** (Single-responsibility principle - Wikipedia) as it makes a new class of two (or more) classes that know nothing about each other.
- You are strongly suggested multiple inheritance as the last of all possible solutions. Composition may be a better alternative.

Composition

Inheritance is not the only way of constructing adaptable classes. You can achieve the same goals (not always, but very often) by using a technique named **composition**.

Composition is the process of composing an object using other different objects.

It can be said that:

- **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 behaviour.

Confused? Let's simplify this a bit.

Inheritance is an **IS-A** relationship.

Composition is an **HAS-A** relationship.

Here is a detailed explanation I found on YouTube (OOP Principles: Composition vs Inheritance)

Below is a sample code for composition.

```python
class robot ():
    def move (self):
        print("moveeeee!")
    def meow (self):
        print("meowwwww!")

class play_ball_game ():
    def chase_the_ball (self):
        print("playing ball games!")

class dog_robot ():
    def __init__ (self):
        self.o1 = robot()
        self.o2 = play_ball_game()

    def bark (self):
        print("woof!")

    def move (self):
        return self.o1.move()
    def chase_the_ball (self):
        return self.o2.chase_the_ball()

a = dog_robot()
a.move() # moveeeee!
a.bark() # woof!
a.chase_the_ball() # playing ball games!
# Note that we didn't "import" the meow function from class robot
```

Method Resolution Order (MRO)

MRO, in general, is a way (you can call it a **strategy**) in which a particular programming language scans through the upper part of a class's hierarchy in order to find the method it currently needs.

It's worth emphasizing that different languages use slightly (or even completely) different MROs. Python is a unique creature in this respect, however, and its customs are a bit specific.

Here is an example code,

```python
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() # bottom
object.m_middle() # middle
object.m_top() # top
```

And here is another example code

```python
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")
```

```python
object = Bottom()
object.m_bottom() # bottom
object.m_middle() # middle
object.m_top() # top
```

Found the difference?

It’s this line $class\ Bottom(Middle, Top)$:.

In this exotic way, we've turned a very simple code with a clear single-inheritance path into a mysterious multiple-inheritance riddle.

“Is it **valid**?” you may ask. **Yes**, it is.

“How is that possible?” you should ask now.

As you can see, the order in which the two superclasses have been listed between parenthesis is compliant with the code's structure: the $Middle$ class precedes the Top class, just like in the real inheritance path.

Despite its oddity, the sample is correct and works as expected, but it has to be stated that this notation doesn’t bring any new functionality or additional meaning.

Let’s modify the code once again.

```python
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()
```

I’ve swapped $class\ Bottom(Middle, Top)$ to $class\ Bottom(Top, Middle)$, try run the code. Now it shows this:

$TypeError: Cannot\ create\ a\ consistent\ method\ resolution\ order\ (MRO)\ for\ bases\ Top, Middle$

The order we tried to force (Top, Middle) is incompatible with the inheritance path which is derived from the code's structure. Python won't like it.

Python's MRO cannot be bent or violated, not just because that's the way Python works, but also because it’s a rule you have to obey.

The diamond problem

The second example of the spectrum of issues that can possibly arise from multiple inheritance is illustrated by a classic problem named the diamond problem. The name reflects the shape of the inheritance diagram, take a look at the picture:

- There is an upmost superclass A.
- There are two subclasses derived from A: B and C.
- The bottommost subclass named D, derived from B and C (or C and B, as these two variants mean different things in Python)

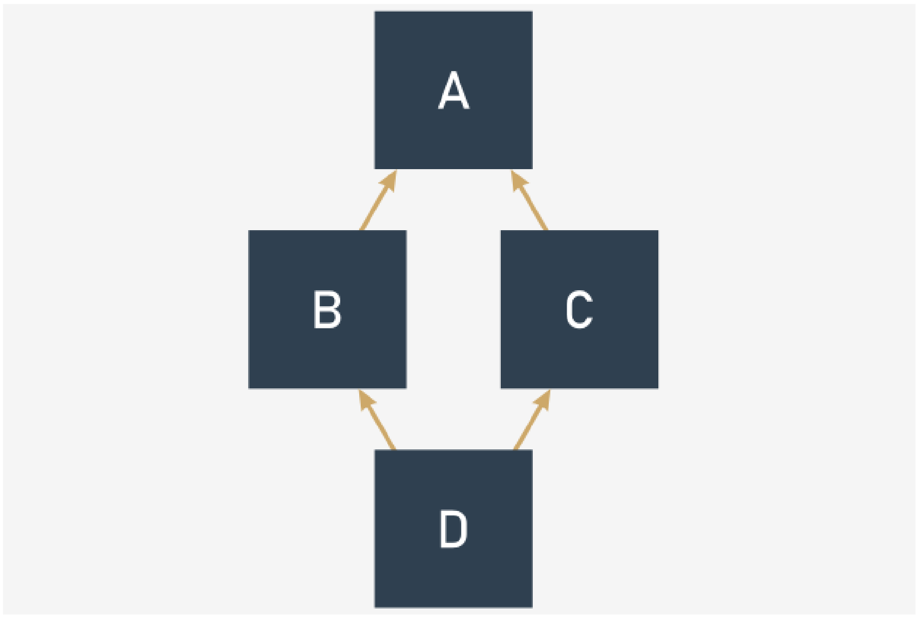


Image source: cisco/Python Institute

Here is the illustrated code

```
class A:
    pass
class B(A):
    pass
class C(A):
    pass
class D(B, C):
    pass
obj = D()
```

To make it more understandable, here is another sample.

```
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() # bottom
object.m_middle() # middle_left
object.m_top() # top
```

Note that both $Middle$ classes define a method of the same name, $m\_middle()$.

It introduces a small uncertainty to our sample, although I'm absolutely sure that you can answer the following question: which of the two $m\_middle()$ methods will actually be invoked when the following line is executed?

Well, as mentioned before, if two classes are at the same inheritance level, Python will process them from left to right. Therefore $middle\_left$ was printed.

As you can see, diamonds may bring some problems into your life – both the real ones and those offered by Python.

USE COMPOSITION WHEN YOU CAN, AND USE INHERITANCE WHEN YOU MUST.

More about exceptions

Discussing object programming offers a very good opportunity to return to exceptions. The objective nature of Python's exceptions makes them a very flexible tool, able to fit to specific needs, even those you don't yet know about.

Before we dive into the objective face of exceptions, I want to show you some syntactical and semantic aspects of the way in which Python treats the try-except block, as it offers a little more than what we have presented so far.

$else$ block

The first feature we want discuss here is an additional, possible branch that can be placed inside (or rather, directly behind) the $try - except$ block, it's the part of the code starting with $else$.

Here is an example, I think you can understand what $else$ block do with it.

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

print(func(2))
# Everything went fine
# 0.5
print(func(0))
# Division failed
# None
```

A code labelled in this way is executed when (and only when) no exception has been raised inside the try: part.

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

$finally$ block

Another extension to the $try - except$ block.

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

print(func(2))
# Everything went fine
# Bye!
# 0.5
print(func(0))
# Division failed
# Bye!
# None
```

The *finally* block is **always** executed (it finalizes the *try* – *except* block execution, hence its name), no matter what happened earlier, even when raising an exception, no matter whether this has been handled or not.

Exceptions are classes

All the previous examples were content with detecting a specific kind of exception and responding to it in an appropriate way. Now we're going to delve deeper, and look inside the exception itself.

When an exception is raised, an object of the class is instantiated, and goes through all levels of program execution, looking for the *except* branch that is prepared to deal with it.

Such an object carries some useful information which can help you to precisely identify all aspects of the pending situation. To achieve that goal, Python offers a special variant of the exception clause.

```
try:
    x = int("Hello!")
except Exception as e:
    print(e) # invalid literal for int() with base 10: 'Hello!'
    print(e.__str__()) # invalid literal for int() with base 10: 'Hello!'
```

As you can see, 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 analyse its nature and draw proper conclusions.

Note: the identifier's scope covers its *except* branch **only**, and doesn't go any further.

The example presents a very simple way of utilizing the received object, printing it out. (The output is produced by the object's __*str*__(), as you can see.) It contains a brief message describing the reason.

The same message will be printed if there is no fitting *except* block in the code, and Python is forced to handle it alone.

```
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)
```

This program dumps all predefined exception classes in the form of a tree-like printout. (Run it yourself, the output will not be provided due to the fact that it's too lengthy).

As a tree is a perfect example of a **recursive data structure**, a recursion seems to be the best tool to traverse through it. The *print_exception_tree*() function takes two arguments:

1. A point inside the tree from which we start traversing the tree;
2. A nesting level (we'll use it to build a simplified drawing of the tree's branches)

And yes, as you might know, this is an algorithm called **D**epth **F**irst **S**earch (DFS).

Let's start from the tree's root - the root of Python's exception classes is the *BaseException* class (it's a superclass of all other exceptions).

For each of the encountered classes, perform the same set of operations:

1. Print its name, taken from the *__name__* property;
2. Iterate through the list of subclasses delivered by the *__subclasses__*() method, and recursively invoke the *print_exception_tree*() function, incrementing the nesting level respectively.

Detailed anatomy of exceptions

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).

```
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 = ' : ') # my exception : my
exception :
    print_args(e.args) # my exception

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

Here is an illustration on how *args* can be used.

How you create your own exception

The exceptions hierarchy is neither closed nor finished, and you can always extend it if you want or need to create your own world populated with your own exceptions.

It may be useful when you create a complex module which detects errors and raises exceptions, and you want the exceptions to be easily distinguishable from any others brought by Python.

This is done by defining **your own**, new exceptions as **subclasses** derived from predefined ones.

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 $Exception$.

Imagine that you've created a brand-new arithmetic, ruled by your own laws and theorems. It's clear that division has been redefined, too, and has to behave in a different way than routine dividing. It's also clear that this new division should raise its own exception, different from the built-in $ZeroDivisionError$, but it's reasonable to assume that in some circumstances, you (or your arithmetic's user) may want to treat all zero divisions in the same way.

Here is an example

```python
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')

# Division by zero
# Division by zero
# Original division by zero
# Division by zero
```

- We've defined our own exception, named $MyZeroDivisionError$, derived from the built-in $ZeroDivisionError$. As you can see, we've decided not to add any new

components to the class. In effect, an exception of this class can be - depending on the desired point of view - treated like a plain $ZeroDivisionError$, or considered separately.
- The $do\_the\_division()$ function raises either a $MyZeroDivisionError$ or $ZeroDivisionError$ exception, depending on the argument's value.

The function is invoked four times in total, while the first two invocations are handled using only one $except$ branch (the more general one) and the last two ones with two different branches, able to distinguish the exceptions (don't forget: the order of the branches makes a fundamental difference!)

```python
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
```

Here is an example of an exception structure.

You can start building it by defining a **general exception** as a new base class for any other specialized exception. Like $PizzaError$ in the above example.

A more specific problem (like an excess of cheese) can require a **more specific** exception. It's possible to derive the new class from the already defined $PizzaError$ class, $TooMuchCheeseError$ here in the example.

The $TooMuchCheeseError$ exception needs more information than the regular $PizzaError$ exception, so we add it to the constructor - the name $cheese$ is then stored for further processing.

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

class TooMuchCheeseError(PizzaError):
    def __init__(self, pizza = "unknown", 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!
# too much cheese : 110
# no such pizza on the menu : mafia
# no such pizza on the menu : mafia
```

Here is a “complete” version of the example above.

Note these:

- removing the branch starting with $except\ TooMuchCheeseError$ will cause all appearing exceptions to be classified as $PizzaError$;
- removing the branch starting with $except\ PizzaError$ will cause the $TooMuchCheeseError$ exceptions to remain unhandled, and will cause the program to terminate.