Python programming for beginners

Stefan Zhauryd
Instructor

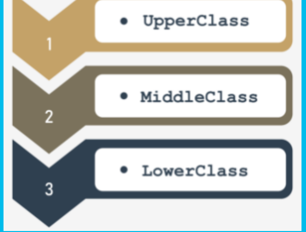
Module 7

Object-Oriented Programming

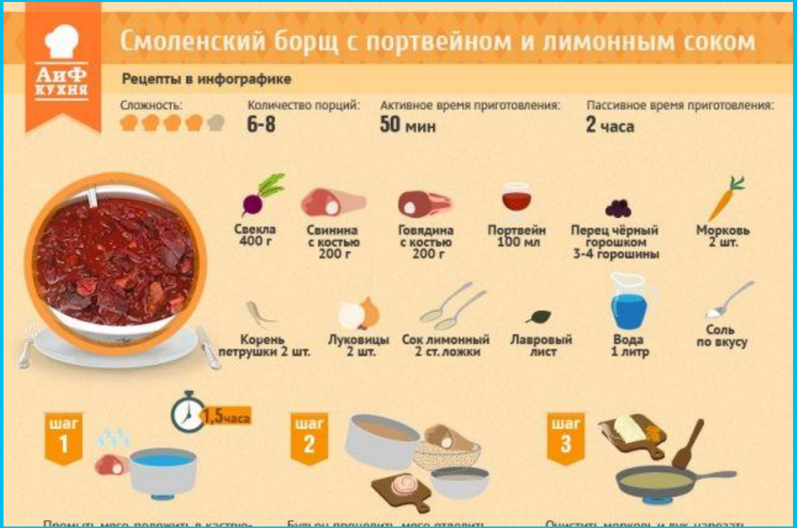
In this module, you will learn about:

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

The basic concepts of the object-oriented approach
type()
class int



Procedural vs. the object-oriented approach


Смоленский борщ с портвейном и лимонным соком

АиФ КУХНЯ

Рецепты в инфографике

Сложность:

Количество порций: 6-8

Активное время приготовления: 50 мин

Пассивное время приготовления: 2 часа

Свекла	400 г
Свинина с костью	200 г
Говядина с костью	200 г
Портвейн	100 мл
Перец чёрный горошком	3-4 горошины
Морковь	2 шт.
Корень петрушки	2 шт.
Луковицы	2 шт.
Сок лимонный	2 ст. ложки
Лавровый лист	
Вода	1 литр
Соль	по вкусу

шаг 1

1,5часа

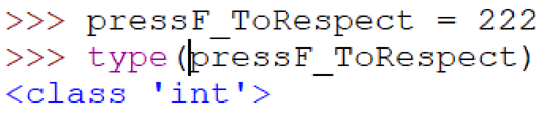
Промыть мясо положить в кастрю-

шаг 2

Бульон процедить, мясо отделить

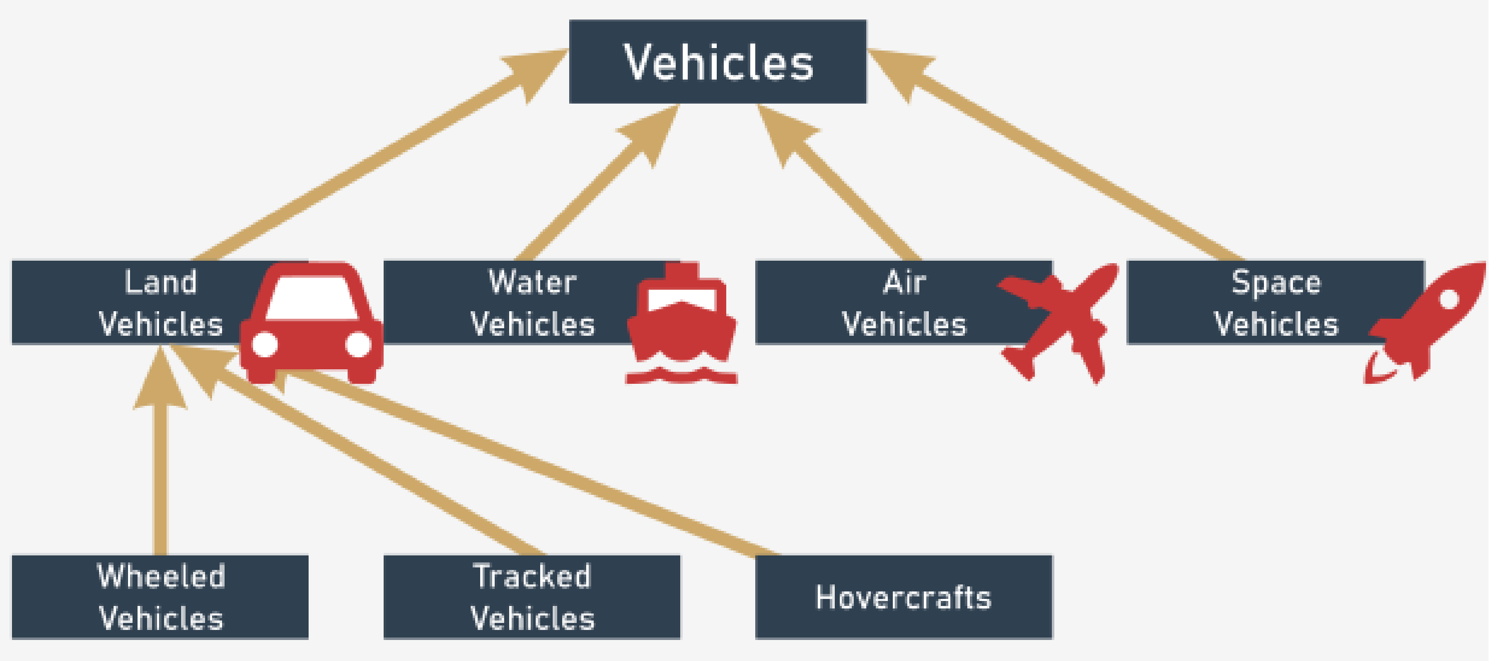
шаг 3

Очистить морковь и лук, нарезать

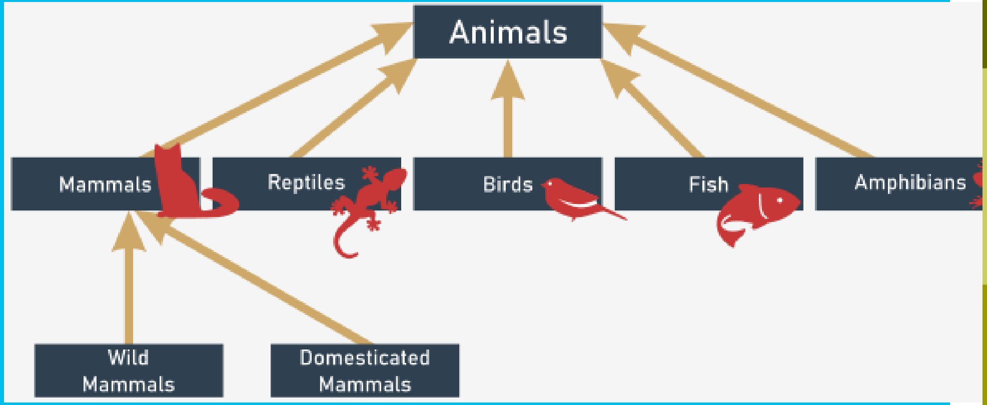


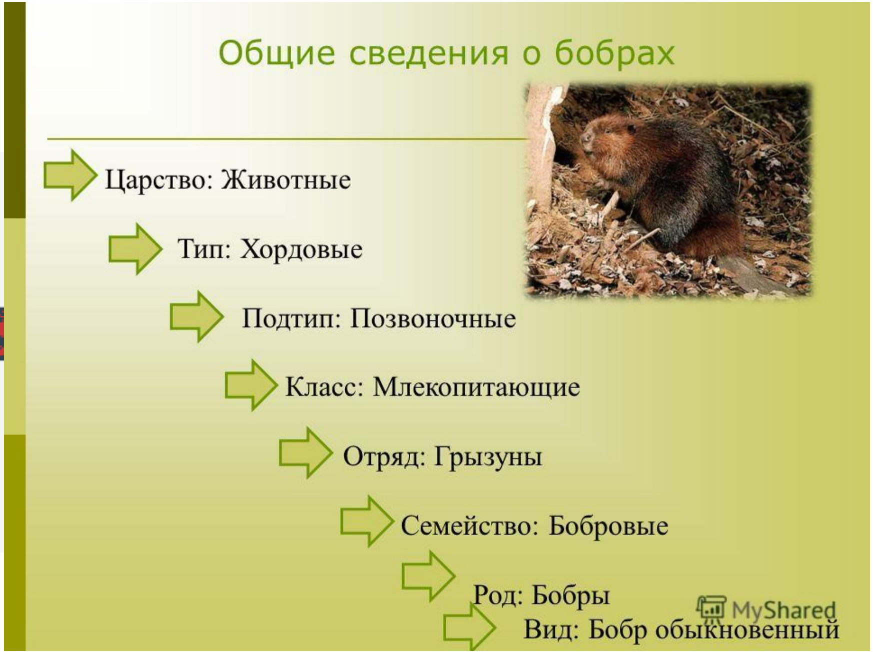
```
>>> pressF_ToRespect = 222
>>> type(pressF_ToRespect)
<class 'int'>
```

Class hierarchies



Class hierarchies: continued

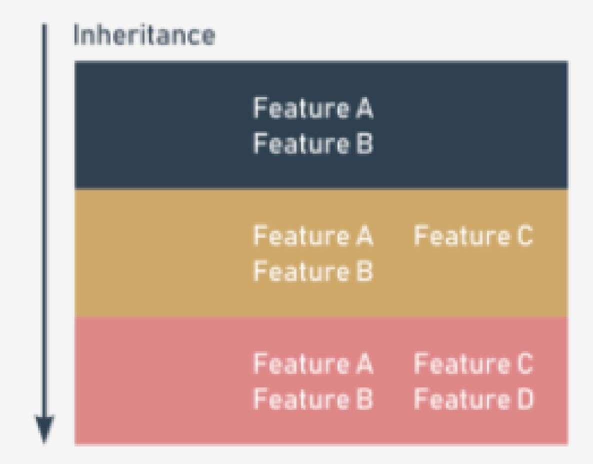




Общие сведения о бобрах

- Царство: Животные
- Тип: Хордовые
- Подтип: Позвоночные
- Класс: Млекопитающие
- Отряд: Грызуны
- Семейство: Бобровые
- Род: Бобры
- Вид: Бобр обыкновенный

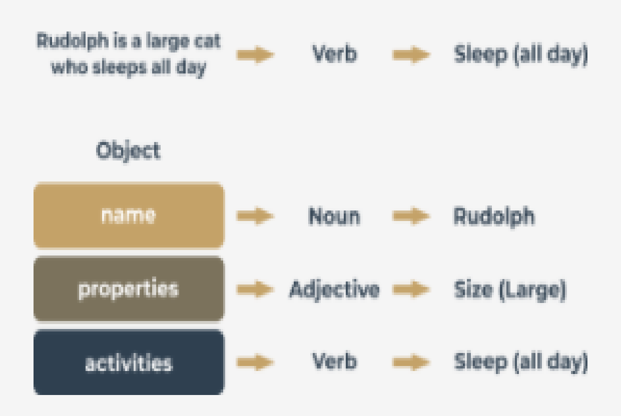
What is an object?
A = ‘hi’
Inheritance



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.

Any object bound to a specific level of a class hierarchy inherits all the traits (as well as the requirements and qualities) defined inside any of the superclasses.



Rudolph is a large cat who sleeps all day → Verb → Sleep (all day)

Object

name → Noun → Rudolph

properties → Adjective → Size (Large)

activities → Verb → Sleep (all day)

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 make 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.

Easier:

a noun – you probably define the object's name;

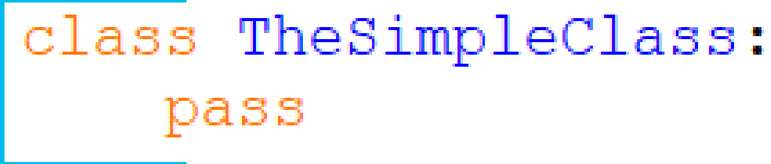
an adjective – you probably define the object's property;

a verb – you probably define the object's activity.

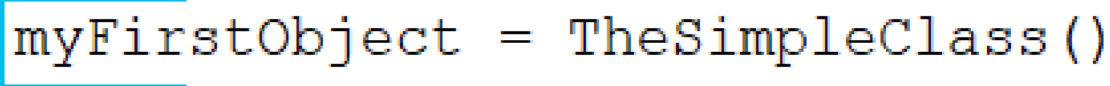
Your first class

Your first object

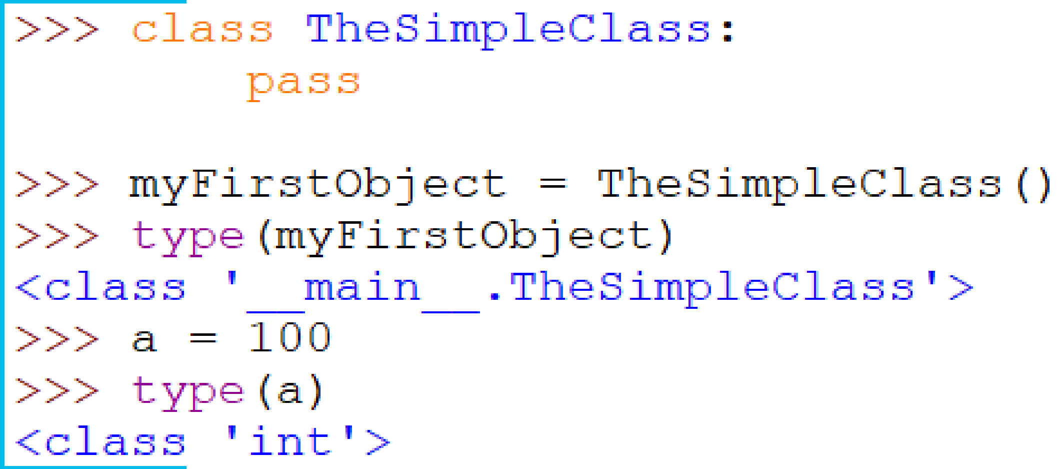
st = list(‘dsdsfsd’)
a = int(‘123’)



```
class TheSimpleClass:
    pass
```

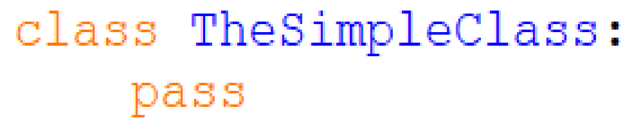


```
myFirstObject = TheSimpleClass()
```

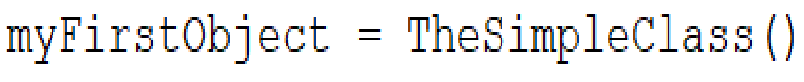


```
>>> class TheSimpleClass:
        pass

>>> myFirstObject = TheSimpleClass()
>>> type(myFirstObject)
<class '__main__.TheSimpleClass'>
>>> a = 100
>>> type(a)
<class 'int'>
```

Key takeaways



```python
class TheSimpleClass:
    pass
```



```python
myFirstObject = TheSimpleClass()
```

1. A class is an idea (more or less abstract) which can be used to create a number of incarnations – such an incarnation is called an object.

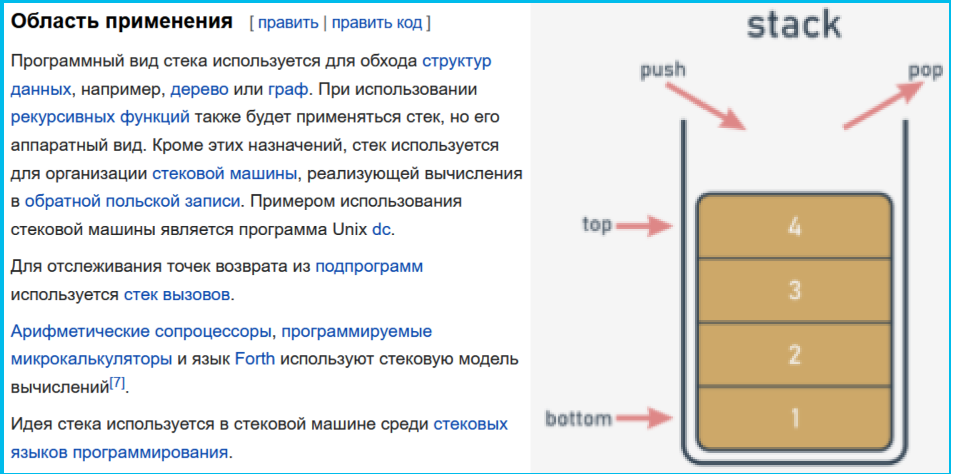
2. When a class is derived from another class, their relation is named inheritance. The class which derives from the other class is named a subclass. The second side of this relation is named superclass. A way to present such a relation is an inheritance diagram, where:

- superclasses are always presented above their subclasses;
- relations between classes are shown as arrows directed from the subclass toward its superclass.

3. **Objects** are equipped with:

- a **name which identifies them** and allows us to distinguish between them;
- **a set of properties** (the set can be empty)
- **a set of methods** (can be empty, too)

What is a stack?



Область применения [править | править код]

Программный вид стека используется для обхода структур данных, например, дерево или граф. При использовании рекурсивных функций также будет применяться стек, но его аппаратный вид. Кроме этих назначений, стек используется для организации стековой машины, реализующей вычисления в обратной польской записи. Примером использования стековой машины является программа Unix dc.

Для отслеживания точек возврата из подпрограмм используется стек вызовов.

Арифметические сопроцессоры, программируемые микрокалькуляторы и язык Forth используют стековую модель вычислений[7].

Идея стека используется в стековой машине среди стековых языков программирования.

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

The alternative name for a stack (but only in IT terminology) is **LIFO**.

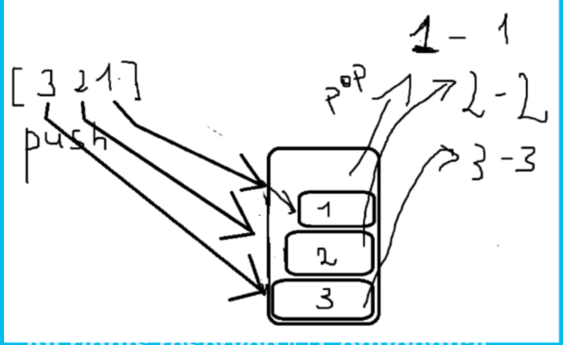
It's an abbreviation for a very clear description of the stack's behavior: Last In - First Out. Who came last onto the stack will leave first.

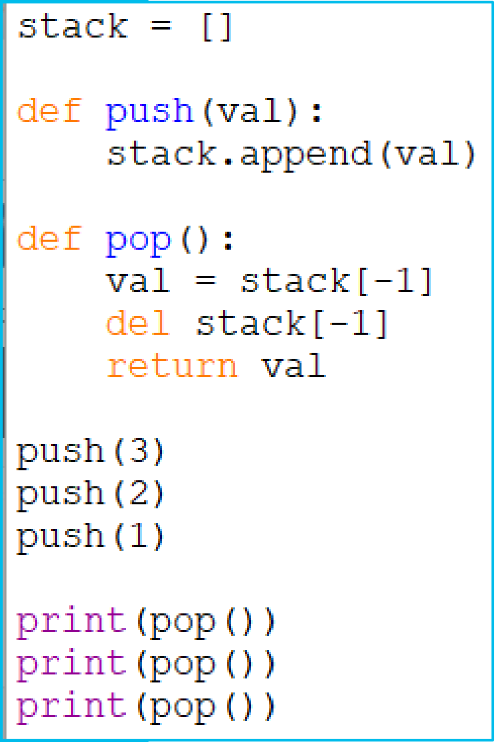
A **stack** is an object with two elementary operations, conventionally named **push** (when a new element is put on the top) and **pop** (when an existing element is taken away from the top).

https://ru.wikipedia.org/wiki/Стек

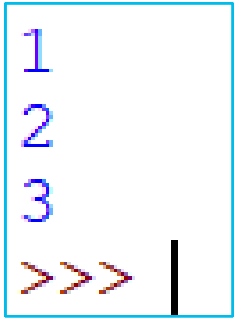
https://habr.com/ru/post/341586/

The stack - the procedural approach





```
stack = []

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

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

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

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

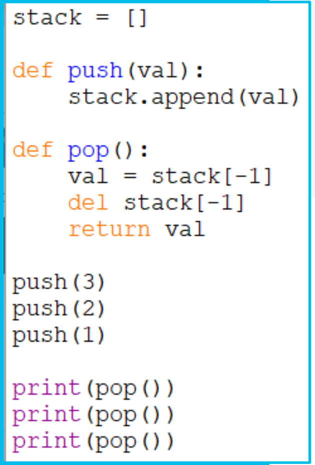


```
1
2
3
>>> |
```

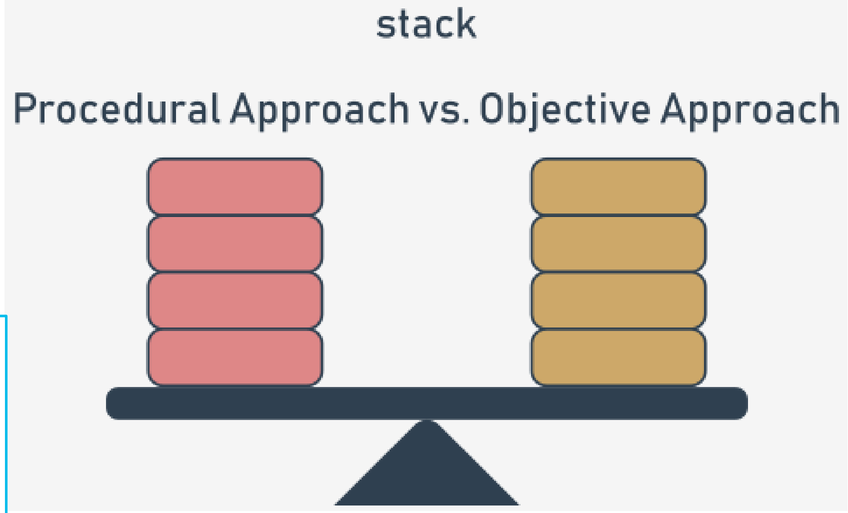
[] - 3 2 1

1 2 3

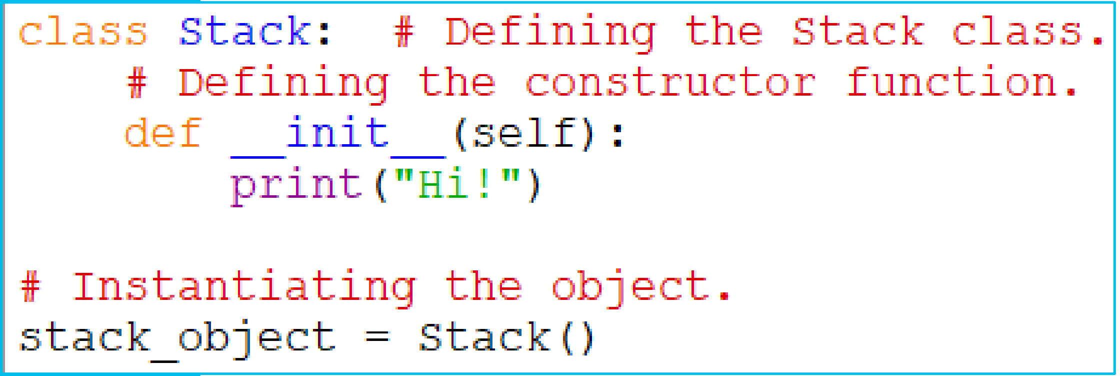
The stack - the procedural approach vs. the object-oriented approach



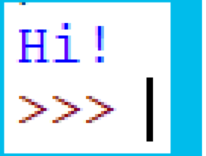
```
stack = []

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

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

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

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



The stack - the object approach

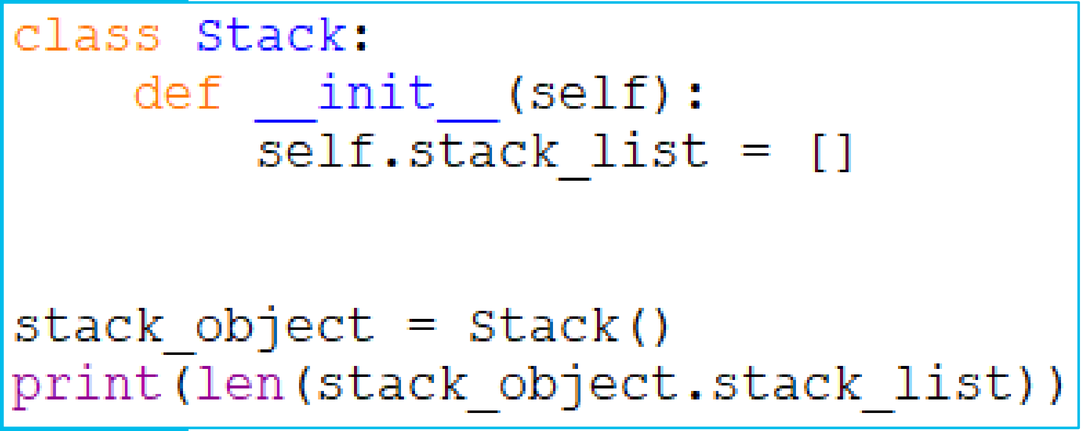


```
class Stack:  # Defining the Stack class.
    # Defining the constructor function.
    def __init__(self):
        print("Hi!")

# Instantiating the object.
stack_object = Stack()
```

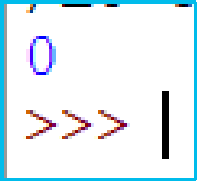


```
Hi!
>>> 
```

The stack - the object approach: continued



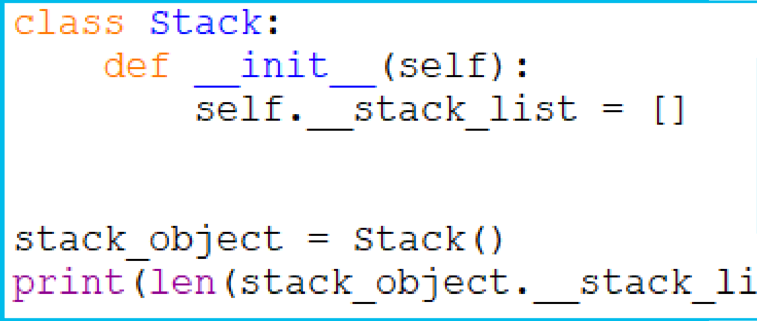
```python
class Stack:
    def __init__(self):
        self.stack_list = []


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



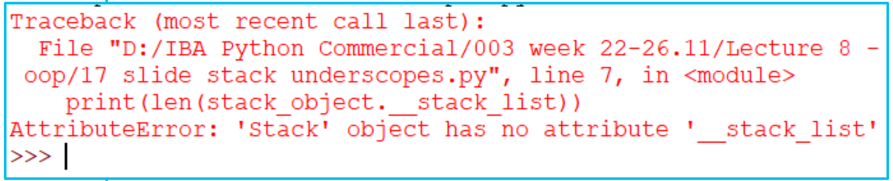
```
0
>>> 
```

https://tirinox.ru/encapsulation-python/

The stack - the object approach: continued



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


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



```
Traceback (most recent call last):
  File "D:/IBA Python Commercial/003 week 22-26.11/Lecture 8 -
 oop/17 slide stack underscopes.py", line 7, in <module>
    print(len(stack_object.__stack_list))
AttributeError: 'Stack' object has no attribute '__stack_list'
>>> |
```

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

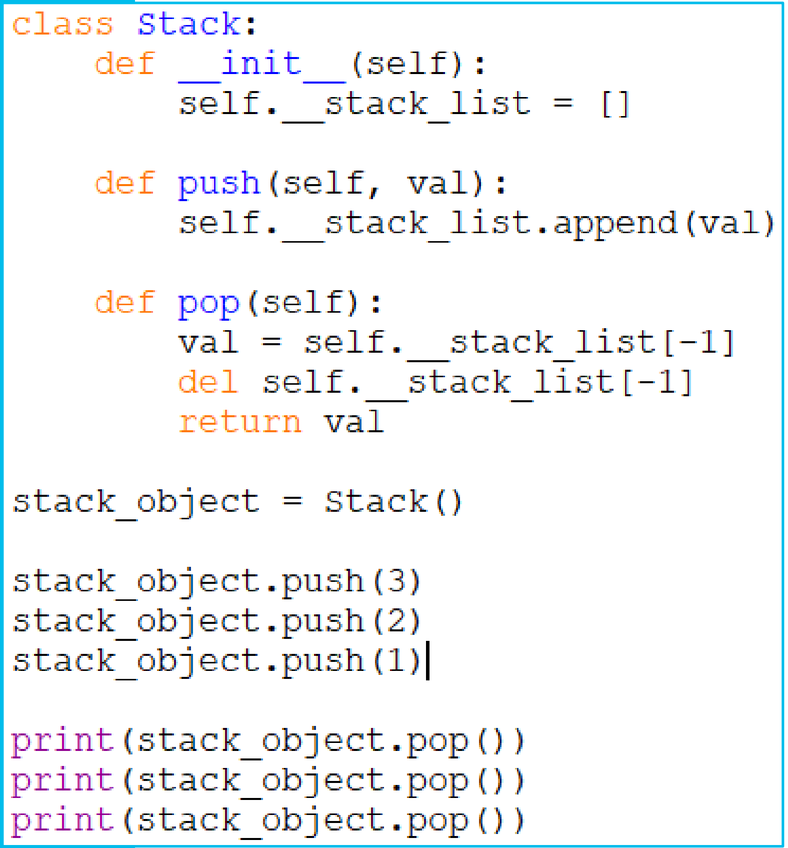
You cannot see it from the outside world. This is how Python implements the **encapsulation concept**.

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

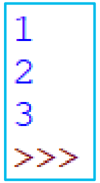
https://pythonist.ru/zachem-nuzhno-nizhnee-podcherkivanie-v-python/

The object approach: a stack from scratch

l = []
l.append()
g = []
g.append()



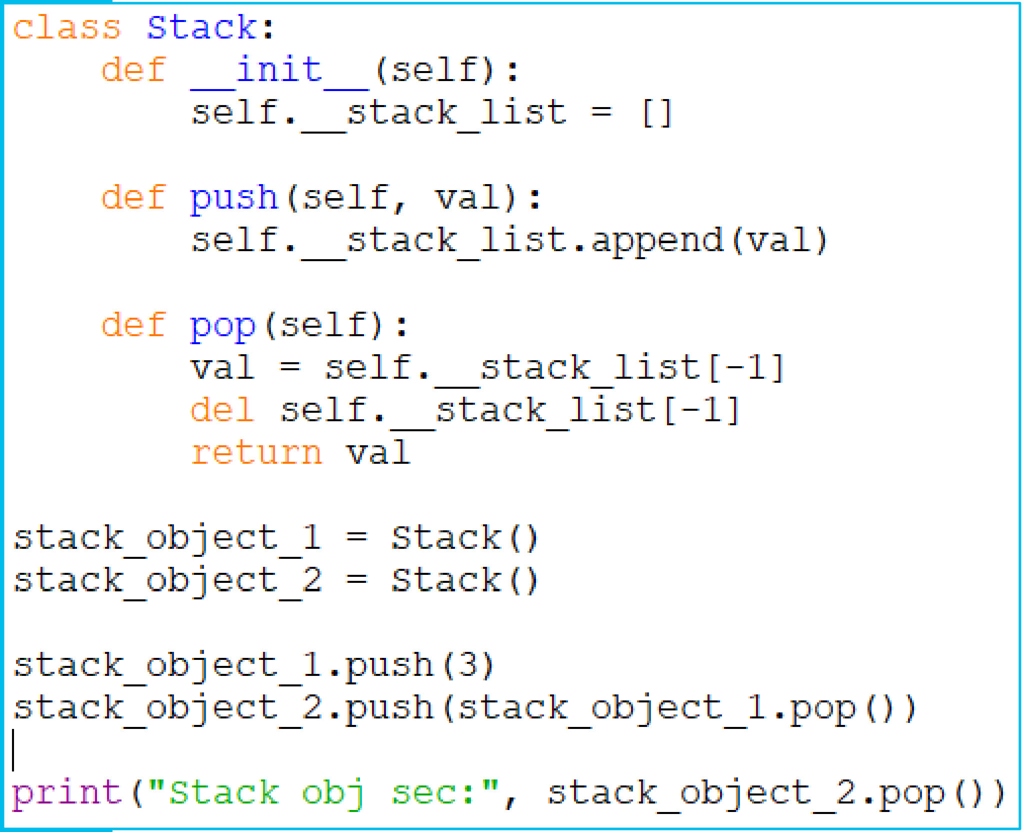
```python
class Stack:
    def __init__(self):
        self.__stack_list = []

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

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

stack_object = Stack()

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

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



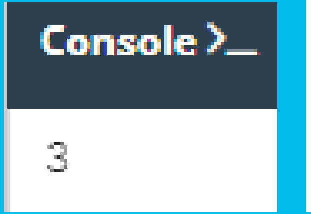
```
1
2
3
>>>
```

The class declaration is complete, and all its components have been listed. The class is ready for use.

The object approach: a stack from scratch



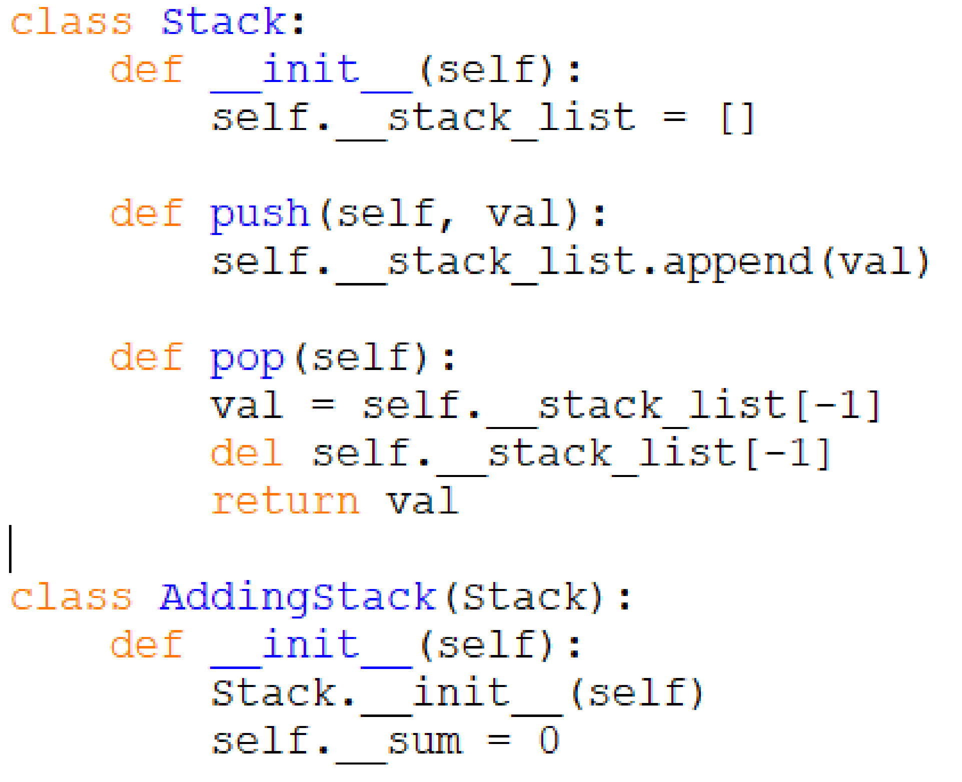
```
class Stack:
    def __init__(self):
        self.__stack_list = []

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

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

stack_object_1 = Stack()
stack_object_2 = Stack()

stack_object_1.push(3)
stack_object_2.push(stack_object_1.pop())

print("Stack obj sec:", stack_object_2.pop())
```



Console >_

```
3
```

The object

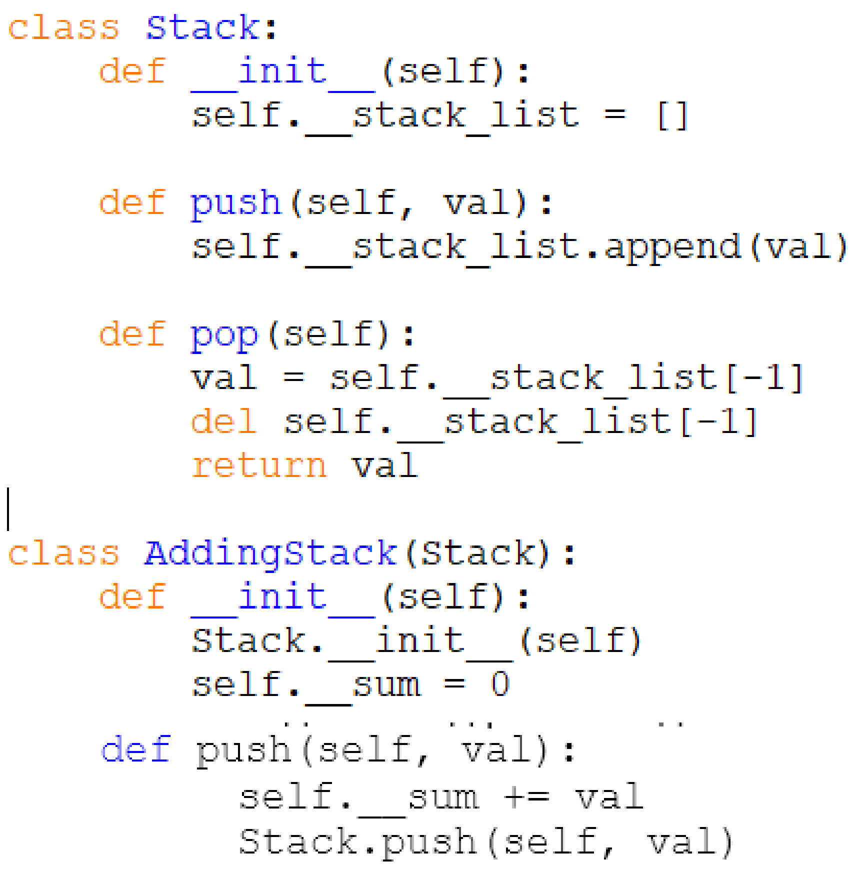


```
class Stack:
    def __init__(self):
        self.__stack_list = []

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

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

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

Note the syntax:

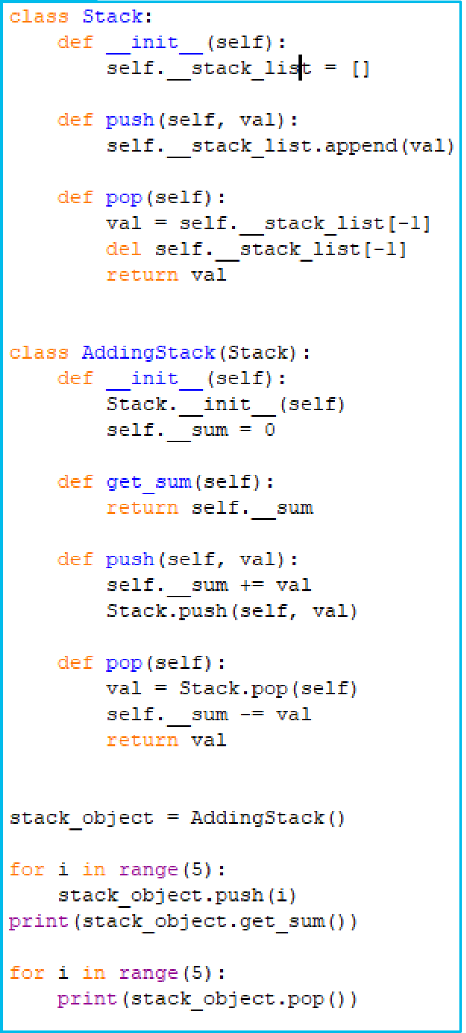
- **you specify the superclass's name** (this is the class whose constructor you want to run)
- **you put a dot (.)after it**;
- **you specify the name of the constructor**;
- **you have to point to the object** (the class's instance) which has to be initialized by the constructor - this is why you have to specify the argument and use the self variable here; note: invoking any method (including constructors) from outside the class never requires you to put the self argument at the argument's list - invoking a method from within the class demands explicit usage of the self argument, and it has to be put first on the list.

The object approach: a stack from scratch (continued)

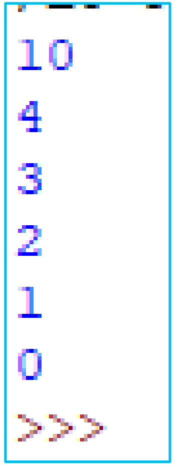


```
class Stack:
    def __init__(self):
        self.__stack_list = []

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

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

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

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

The object approach: a stack from scratch (continued)



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

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

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


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

    def get_sum(self):
        return self.__sum

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

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


stack_object = AddingStack()

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

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



```
10
4
3
2
1
0
>>>
```

Key takeaways

1. A **stack** is an object designed to store data using the **LIFO model**. The stack usually accomplishes at least two operations, named **push()** and **pop()**.

2. **Implementing the stack in a procedural model** raises several problems which can be solved by the techniques offered by **OOP** (Object Oriented Programming):

3. A **class method** is actually a function declared inside the class and able to access all the class's components.

4. The part of the Python class responsible for creating new objects is called the **constructor**, and it's implemented as **a method of the name __init__**.

5. Each class method declaration must contain at least one parameter (always the first one) usually referred to as **self**, and is used by the objects to identify themselves.

6. If we want to hide any of a class's components from the outside world, we should start its name with __. Such components are called **private**.

Home work 9_1
My Stack

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

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

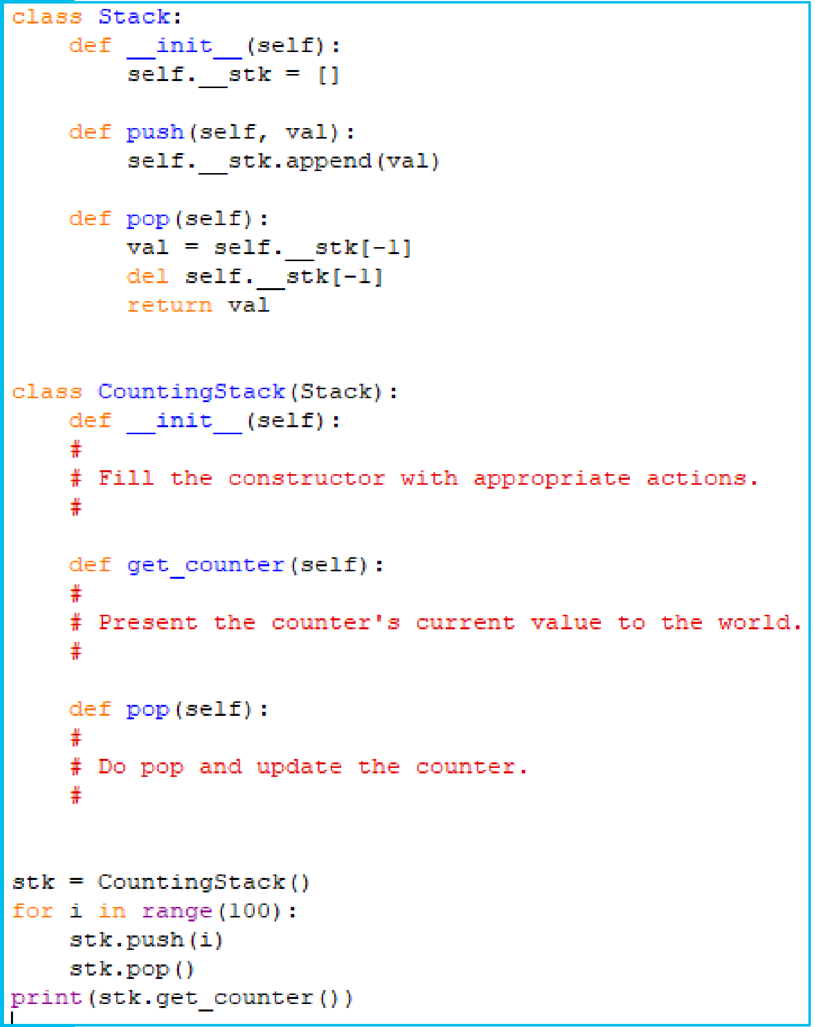
Use the Stack class I've provided in the editor. (next slide)

Follow the hints:

- introduce a property designed to count pop operations and name it in a way which guarantees hiding it;
- initialize it to zero inside the constructor;
- provide a method which returns the value currently assigned to the counter (name it get_counter()).

Complete the code in the next slide. Run it to check whether your code outputs 100.

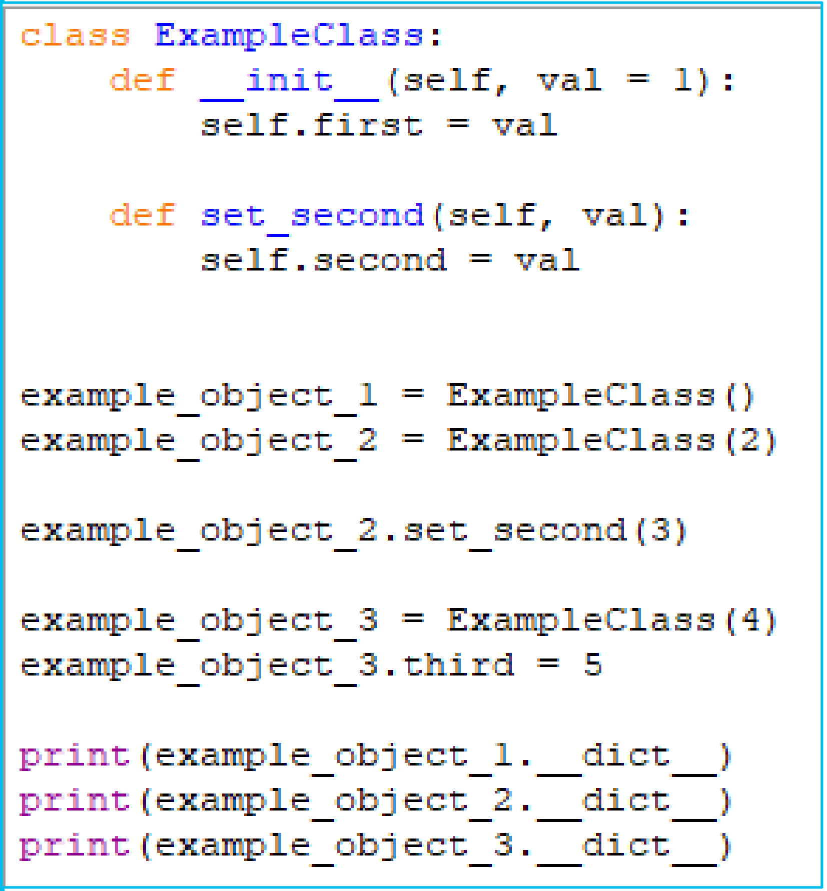
Home work 9_1
__counter = 0



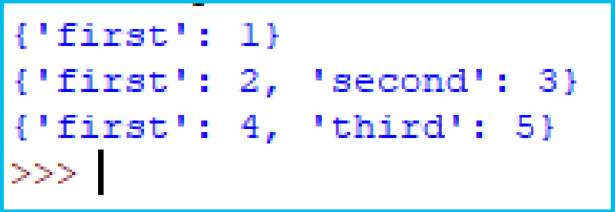
```
class Stack:
    def __init__(self):
        self.__stk = []

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

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


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

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

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


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

Instance variables

__dict__ - names and vals each vars, that an object have at the time

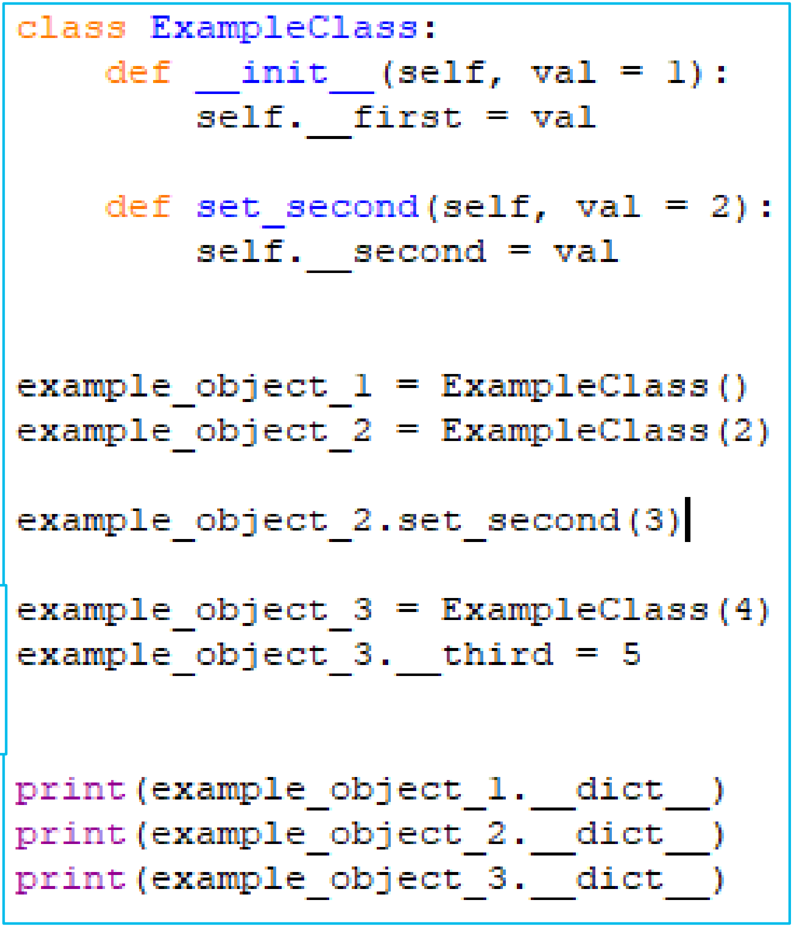


```python
class ExampleClass:
    def __init__(self, val = 1):
        self.first = val

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



example_object_1 = ExampleClass()
example_object_2 = ExampleClass(2)

example_object_2.set_second(3)

example_object_3 = ExampleClass(4)
example_object_3.third = 5

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

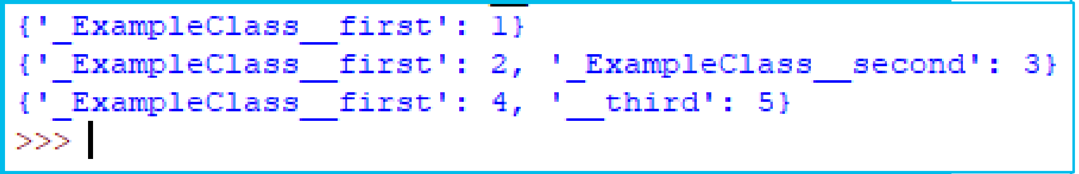


```
{'first': 1}
{'first': 2, 'second': 3}
{'first': 4, 'third': 5}
>>>
```

Instance variables: continued



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

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


example_object_1 = ExampleClass()
example_object_2 = ExampleClass(2)

example_object_2.set_second(3)

example_object_3 = ExampleClass(4)
example_object_3.__third = 5


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



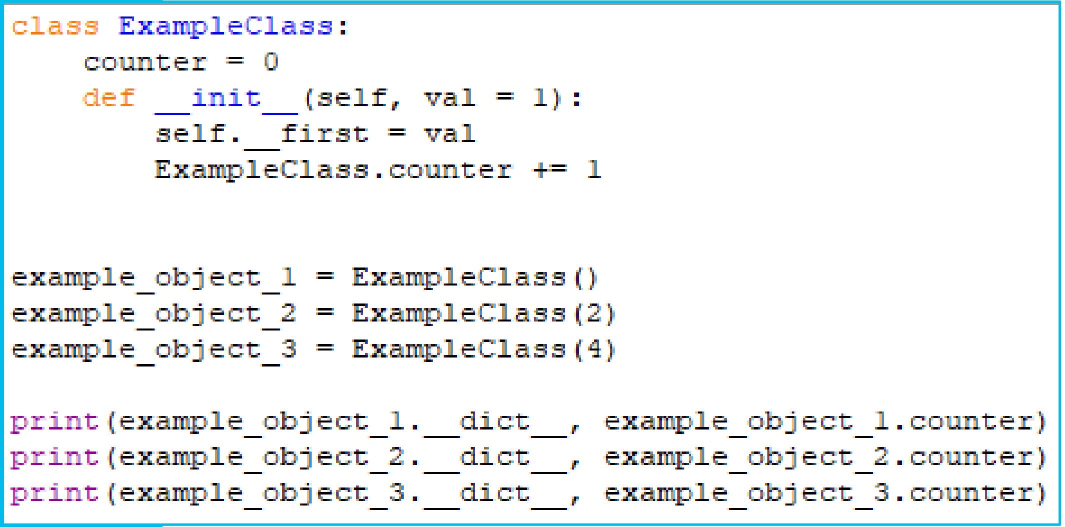
```
{'_ExampleClass__first': 1}
{'_ExampleClass__first': 2, '_ExampleClass__second': 3}
{'_ExampleClass__first': 4, '__third': 5}
>>> 
```

Class variables

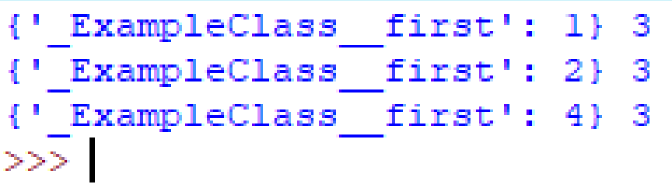
A class variable is a property which exists in just one copy and is stored outside any object.

Two important conclusions come from the example:

- class variables aren't shown in an object's __**dict**__ (this is natural as class variables aren't parts of an object) but you can always try to look into the variable of the same name, but at the class level - we'll show you this very soon;
- a class variable always presents the same value in all class instances (objects)

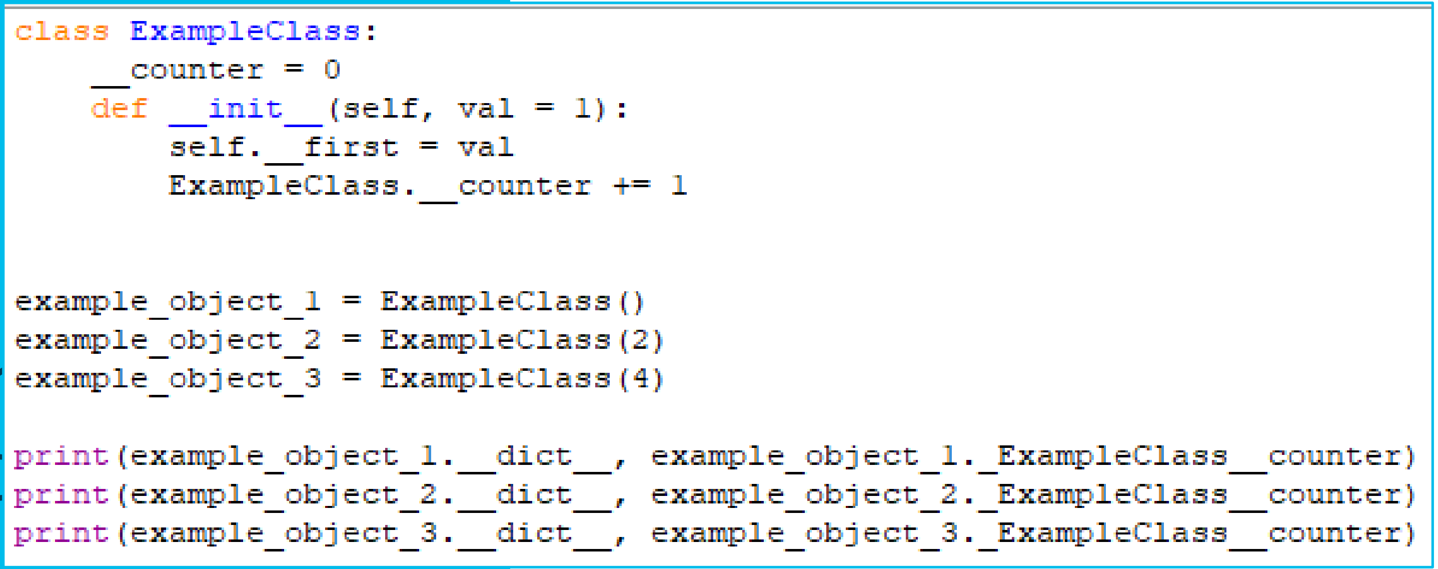


```
class ExampleClass:
    counter = 0
    def __init__(self, val = 1):
        self.__first = val
        ExampleClass.counter += 1


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

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

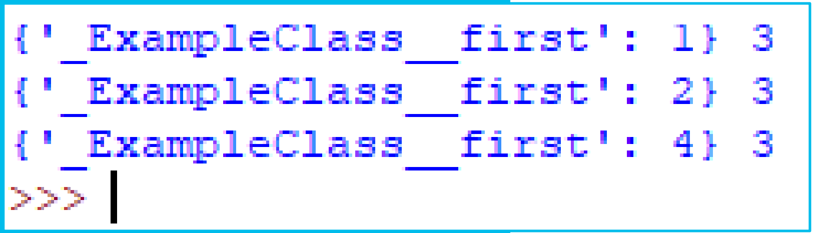


```
{'_ExampleClass__first': 1} 3
{'_ExampleClass__first': 2} 3
{'_ExampleClass__first': 4} 3
>>> 
```

Class var
continuec

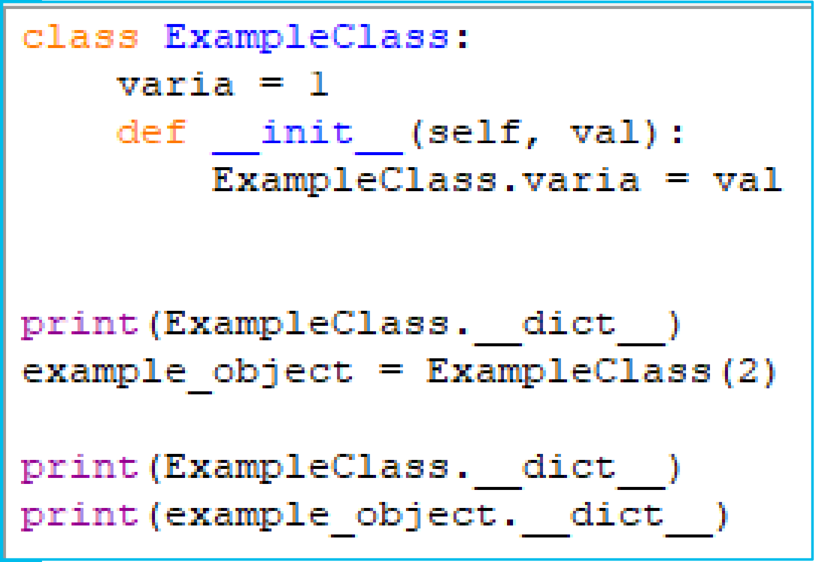


```python
class ExampleClass:
    __counter = 0
    def __init__(self, val = 1):
        self.__first = val
        ExampleClass.__counter += 1


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

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

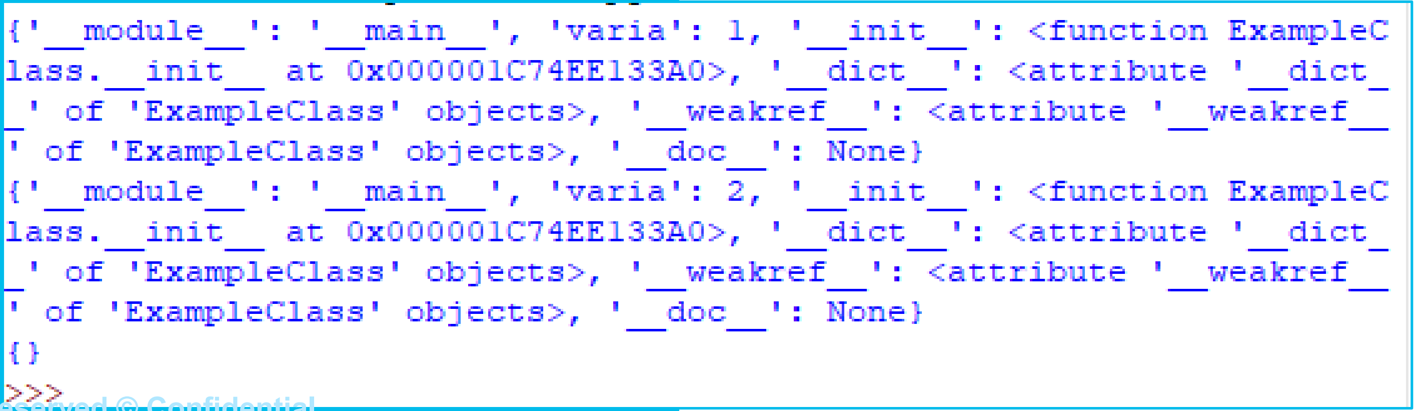


```
{'_ExampleClass__first': 1} 3
{'_ExampleClass__first': 2} 3
{'_ExampleClass__first': 4} 3
>>> 
```

Class variables: continued



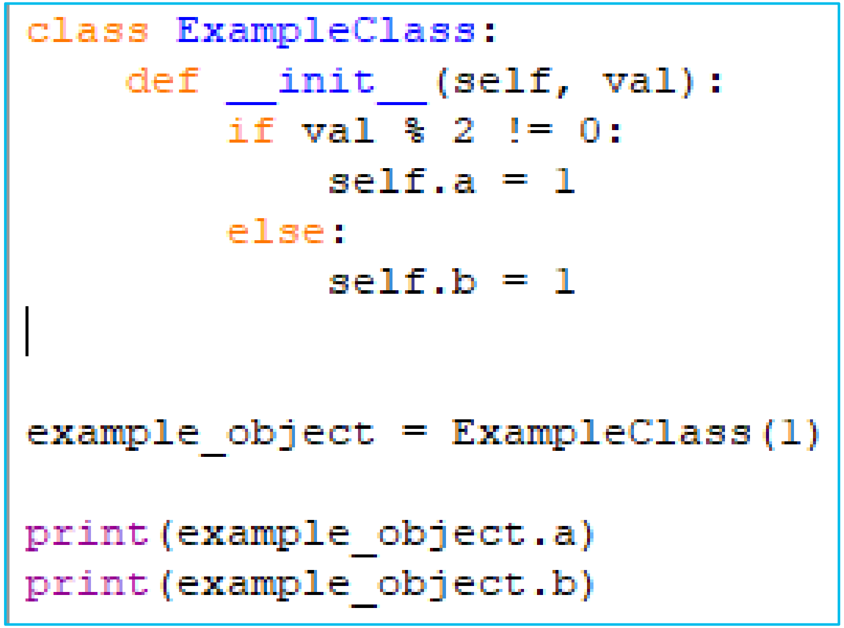
```
class ExampleClass:
    varia = 1
    def __init__(self, val):
        ExampleClass.varia = val


print(ExampleClass.__dict__)
example_object = ExampleClass(2)

print(ExampleClass.__dict__)
print(example_object.__dict__)
```



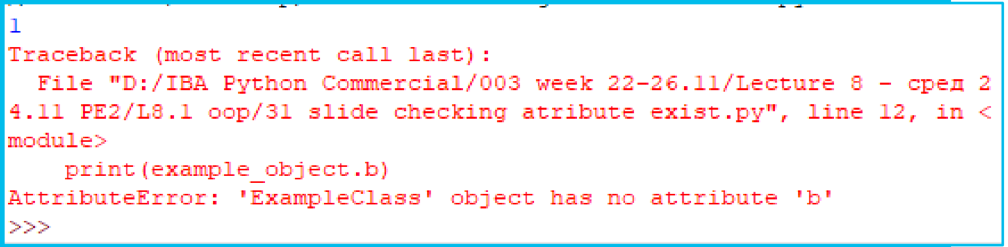
```
{'__module__': '__main__', 'varia': 1, '__init__': <function ExampleC
lass.__init__ at 0x000001C74EE133A0>, '__dict__': <attribute '__dict_
_' of 'ExampleClass' objects>, '__weakref__': <attribute '__weakref__
' of 'ExampleClass' objects>, '__doc__': None}
{'__module__': '__main__', 'varia': 2, '__init__': <function ExampleC
lass.__init__ at 0x000001C74EE133A0>, '__dict__': <attribute '__dict_
_' of 'ExampleClass' objects>, '__weakref__': <attribute '__weakref__
' of 'ExampleClass' objects>, '__doc__': None}
{}
>>>
```

Checking an attribute's existence

Python's attitude to object instantiation raises one important issue - in contrast to other programming languages, you may not expect that all objects of the same class have the same sets of properties.

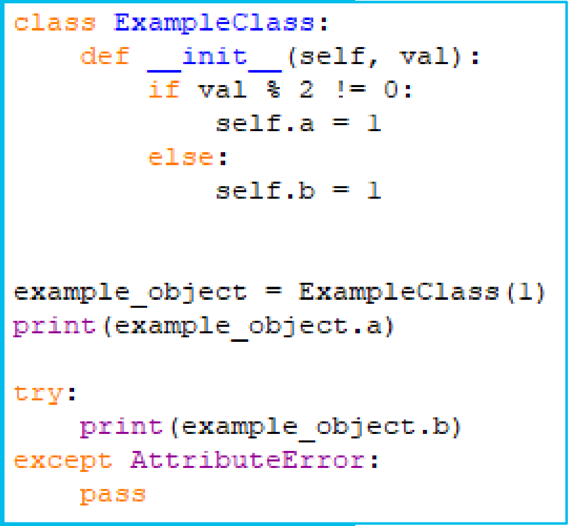


```python
class ExampleClass:
    def __init__(self, val):
        if val % 2 != 0:
            self.a = 1
        else:
            self.b = 1


example_object = ExampleClass(1)

print(example_object.a)
print(example_object.b)
```

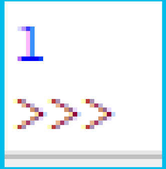


```
1
Traceback (most recent call last):
  File "D:/IBA Python Commercial/003 week 22-26.11/Lecture 8 - сред 2
4.11 PE2/L8.1 oop/31 slide checking atribute exist.py", line 12, in <
module>
    print(example_object.b)
AttributeError: 'ExampleClass' object has no attribute 'b'
>>>
```

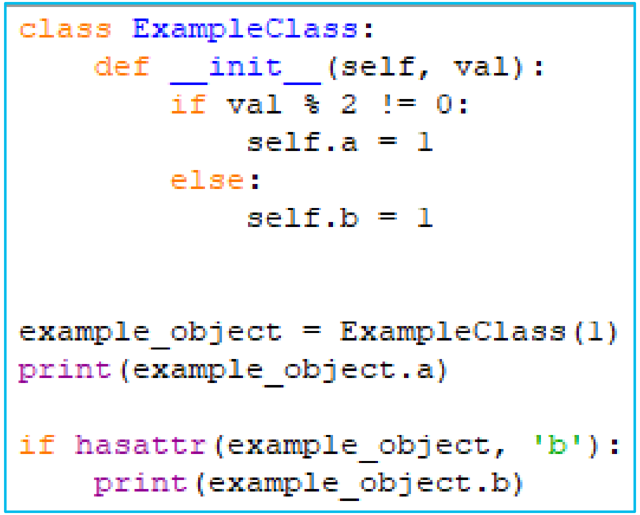
Checking an attribute's existence: continued



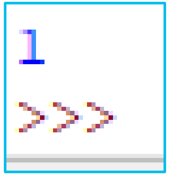
```
class ExampleClass:
    def __init__(self, val):
        if val % 2 != 0:
            self.a = 1
        else:
            self.b = 1


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

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



```
1
>>>
```



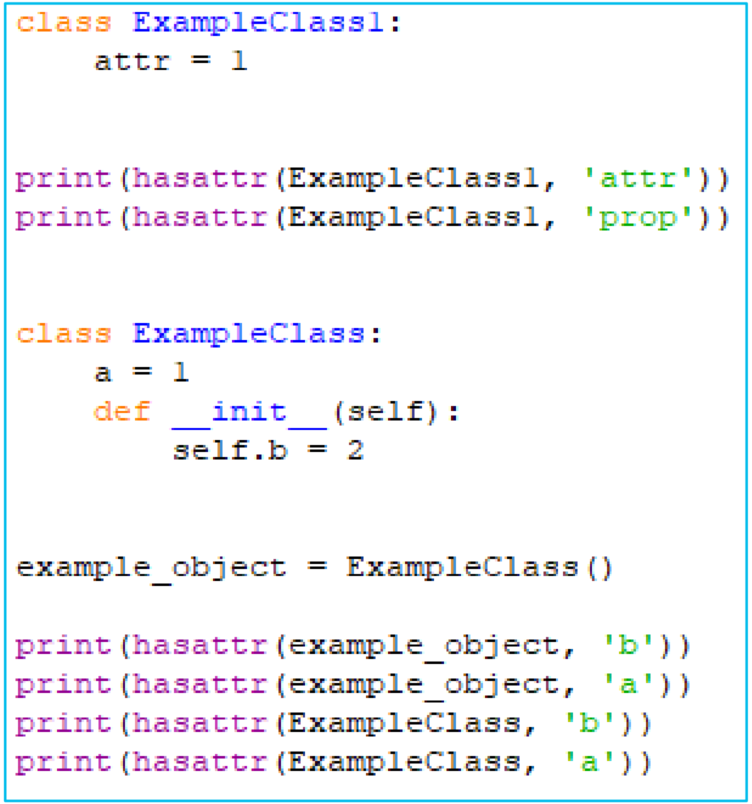
```
class ExampleClass:
    def __init__(self, val):
        if val % 2 != 0:
            self.a = 1
        else:
            self.b = 1


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

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



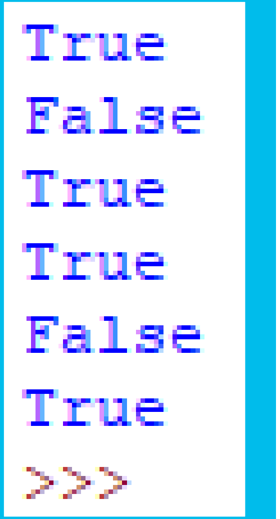
```
1
>>>
```

Checking an attribute's existence: continued

Don't forget that the hasattr() function can operate on classes, too. You can use it to find out if a class variable is available.



```python
class ExampleClass1:
    attr = 1


print(hasattr(ExampleClass1, 'attr'))
print(hasattr(ExampleClass1, 'prop'))


class ExampleClass:
    a = 1
    def __init__(self):
        self.b = 2


example_object = ExampleClass()

print(hasattr(example_object, 'b'))
print(hasattr(example_object, 'a'))
print(hasattr(ExampleClass, 'b'))
print(hasattr(ExampleClass, 'a'))
```



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

Key takeaways

1. An **instance variable** is a property whose existence depends on the creation of an object. **Every object can have a different set of instance variables.**

Moreover, they can be freely added to and removed from objects during their lifetime. All object instance variables are stored inside a dedicated dictionary named __**dict**__, contained in every object separately.

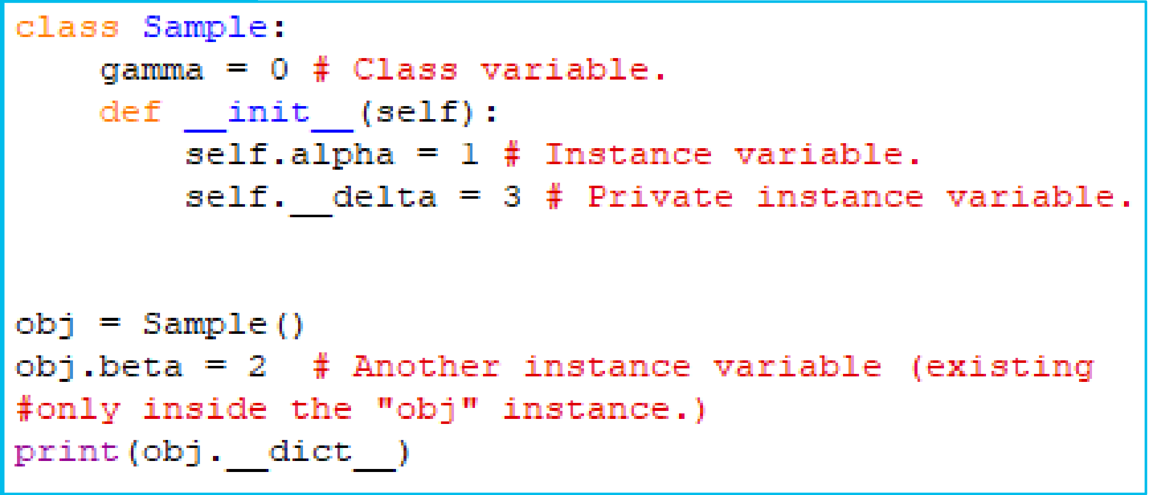
2. An instance **variable can be private** when its name starts with __, but don't forget that such a property is still accessible from outside the class using a mangled name constructed as **_ClassName__PrivatePropertyName.**

3. A class variable is a property which exists in exactly one copy, and doesn't need any created object to be accessible. **Such variables are not shown as __dict__ content.**

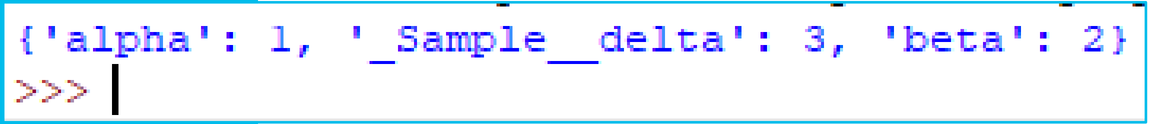
All a class's class **variables are stored inside a dedicated dictionary named __dict__, contained in every class separately.**

Key takeaways

4. A function named **hasattr()** can be used to determine if any object/class contains a specified property.



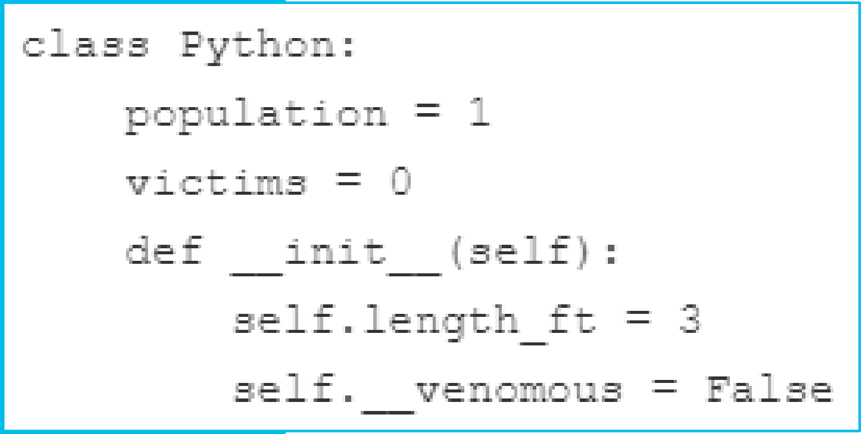
```
class Sample:
    gamma = 0 # Class variable.
    def __init__(self):
        self.alpha = 1 # Instance variable.
        self.__delta = 3 # Private instance variable.


obj = Sample()
obj.beta = 2  # Another instance variable (existing
#only inside the "obj" instance.)
print(obj.__dict__)
```



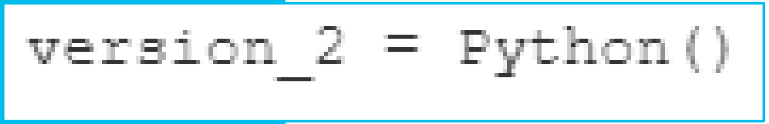
```
{'alpha': 1, '_Sample__delta': 3, 'beta': 2}
>>> 
```

All rights reserved © Confidential

Examples



```
class Python:
    population = 1
    victims = 0
    def __init__(self):
        self.length_ft = 3
        self.__venomous = False
```



```
version_2 = Python()
```

ЗАДАНИЯ

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

Почитать:

1) Byte of Python - стр.108-120

Крайний срок сдачи 14/10 в 21:00 (можно раньше, но не позже)

ЗАДАНИЯ

Название файлов, которые вы отправляете мне в telegram:

Vasia_Pupkin_class_work_L9_P1.py

Без классной работы домашнее не принимается на проверку

+все задания ОДНИМ ФАЙЛОМ - Vasia_Pupkin_L9_P1.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 9 часть 1 про ООП.

И отправляете файлы

SOLOLEARN – 16/10

Крайний срок сдачи 14/10 в 21:00 (можно раньше, но не позже)

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

Tap to links
if you want to know more

Work with files:
https://www.youtube.com/watch?v=oRr_bEXJbV0
https://www.w3schools.com/python/python_ref_file.asp

Books for great peoples:
992 pages of "real" python

993 pages of "real" python

Watch this channel, useful things:
https://www.youtube.com/c/egoroffchannel/playlists

https://www.w3schools.com/python/default.asp

https://www.youtube.com/channel/UCr-KbmZWfDyTbqT_clZmhfw/videos

Q&A

Create your possibilities. Bye bye.