Python programming for beginners

Stefan Zhauryd
Instructor

Module

Object-Oriented Programming

In this module, you will learn about:

- Classes, instances, attributes, methods, as well as working with class and instance data;
- abstract classes, method overriding, static and class methods, special methods;
- inheritance, polymorphism, subclasses, and encapsulation;
- advanced exception handling techniques;
- metaclasses.

Introduction to Object-Oriented Programming Telegram api

class — an idea, blueprint, or recipe for an instance;

instance — an instantiation of the class; very often used interchangeably with the term 'object';

object — Python's representation of data and methods; objects could be aggregates of instances;

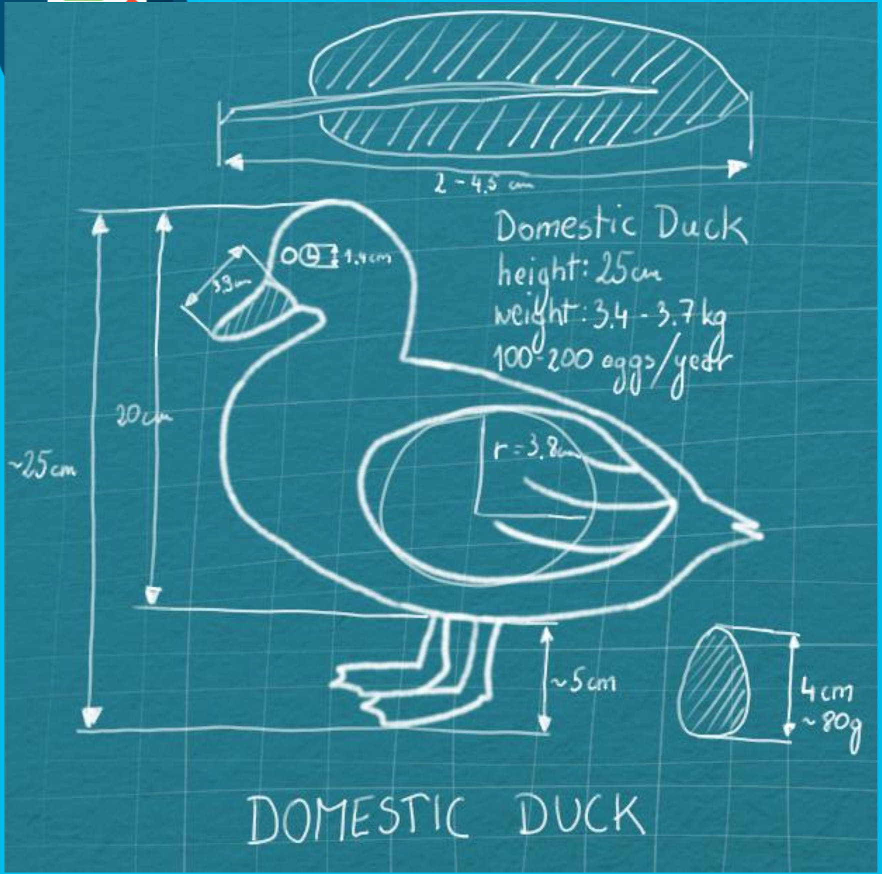
attribute — any object or class trait; could be a variable or method;

method — a function built into a class that is executed on behalf of the class or object; some say that it's a 'callable attribute';

type — refers to the class that was used to instantiate the object

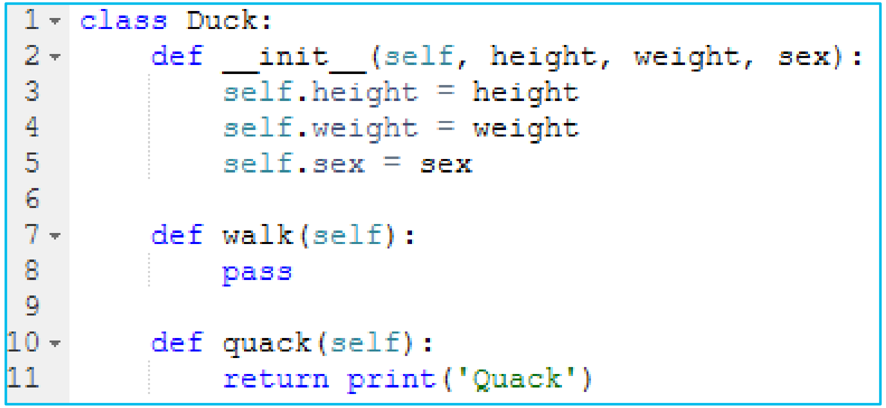
The following issues will be addressed during this and the next lecture:

- creation and use of decorators;
- implementation of **core syntax**;
- class and static methods;
- abstract methods;
- comparison of inheritance and composition;
- attribute encapsulation;
- metaprogramming.



we have defined a class named Duck, consisting of some functionalities and attributes.

A class is a place which binds data with the code.

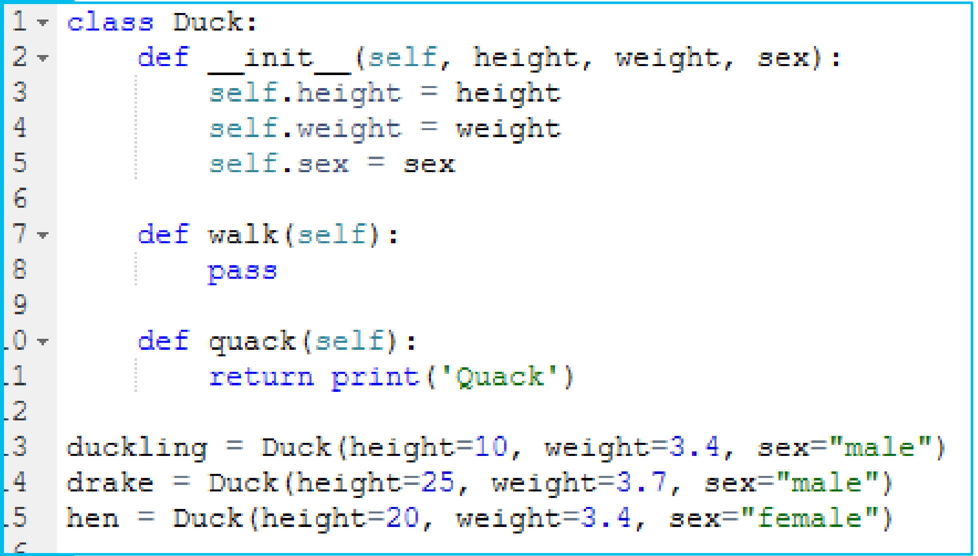

```python
class Duck:
    def __init__(self, height, weight, sex):
        self.height = height
        self.weight = weight
        self.sex = sex

    def walk(self):
        pass

    def quack(self):
        return print('Quack')
```

An instance is one particular physical instantiation of a class that occupies memory and has data elements. This is what 'self' refers to when we deal with class instances.

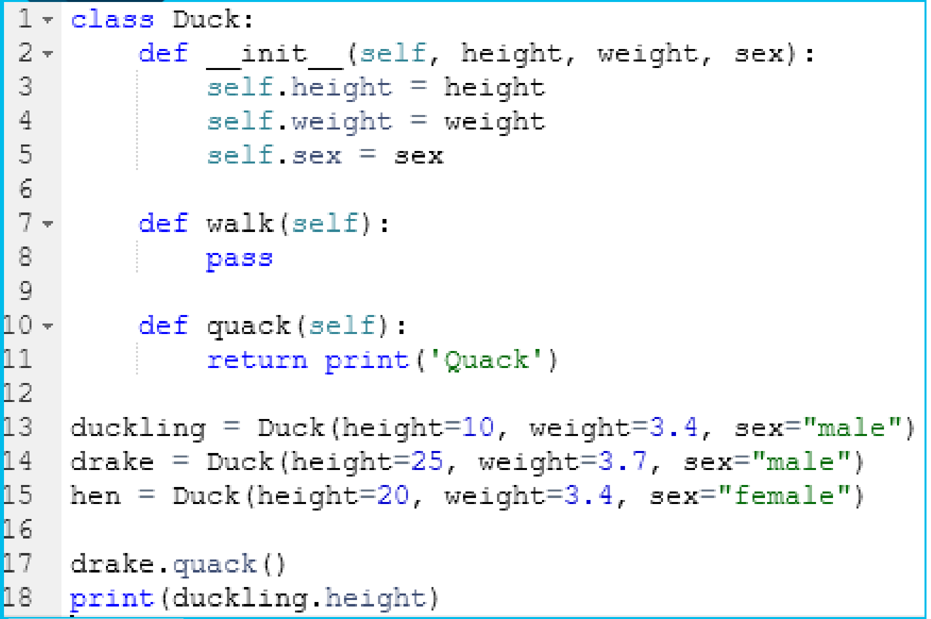
An object is everything in Python that you can operate on, like a class, instance, list, or dictionary.

Classes, Instances, Attributes, Methods

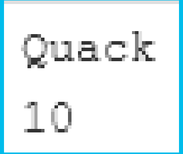
we have created three different instances based on the Duck class: duckling, drake and hen. We haven't called any object attributes.



```python
class Duck:
    def __init__(self, height, weight, sex):
        self.height = height
        self.weight = weight
        self.sex = sex

    def walk(self):
        pass

    def quack(self):
        return print('Quack')

duckling = Duck(height=10, weight=3.4, sex="male")
drake = Duck(height=25, weight=3.7, sex="male")
hen = Duck(height=20, weight=3.4, sex="female")
```



```python
class Duck:
    def __init__(self, height, weight, sex):
        self.height = height
        self.weight = weight
        self.sex = sex

    def walk(self):
        pass

    def quack(self):
        return print('Quack')

duckling = Duck(height=10, weight=3.4, sex="male")
drake = Duck(height=25, weight=3.7, sex="male")
hen = Duck(height=20, weight=3.4, sex="female")

drake.quack()
print(duckling.height)
```

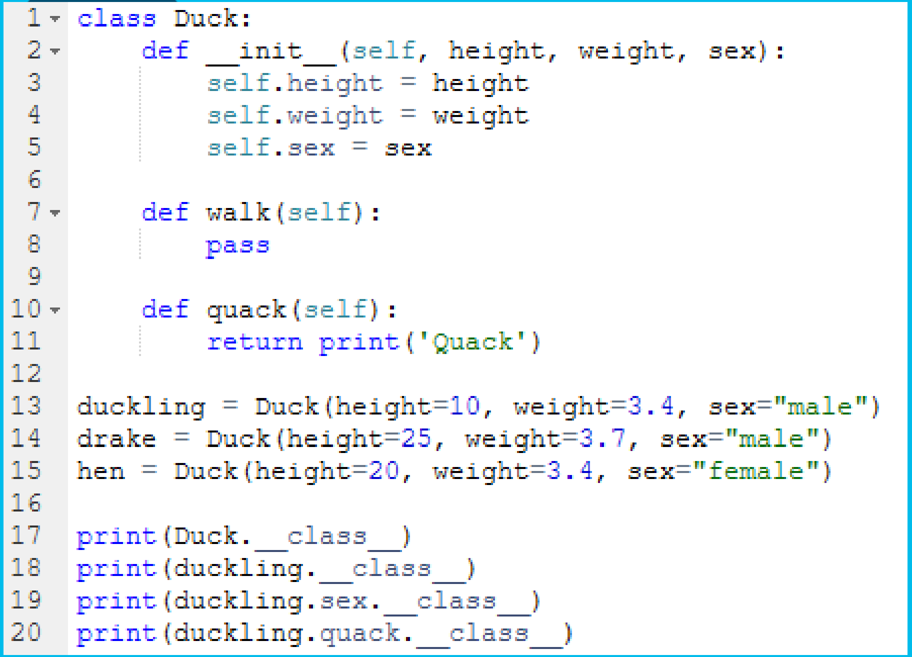


```
Quack
10
```

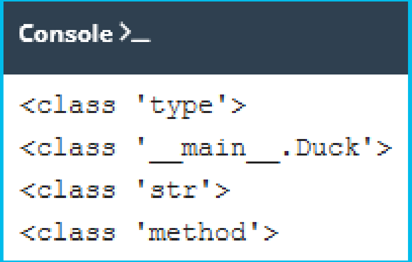
An **attribute** is a capacious term that can refer to two major kinds of class traits:

- **variables**, containing information about the class itself or a class instance; classes and class instances can own many variables;
- **methods**, formulated as Python functions; they represent a behavior that could be applied to the object.

Class attributes are most often addressed with 'dot' notation, i.e., <class>dot<attribute>. The other way to access attributes (variables) it to use the getattr() and setattr() functions.



```python
class Duck:
    def __init__(self, height, weight, sex):
        self.height = height
        self.weight = weight
        self.sex = sex

    def walk(self):
        pass

    def quack(self):
        return print('Quack')

duckling = Duck(height=10, weight=3.4, sex="male")
drake = Duck(height=25, weight=3.7, sex="male")
hen = Duck(height=20, weight=3.4, sex="female")

print(Duck.__class__)
print(duckling.__class__)
print(duckling.sex.__class__)
print(duckling.quack.__class__)
```



Console >_

```
<class 'type'>
<class '__main__.Duck'>
<class 'str'>
<class 'method'>
```

A **type** is one of the most fundamental and abstract terms of Python:

- it is the foremost type that any class can be inherited from;
- as a result, if you're looking for the type of class, then type is returned;
- in all other cases, it refers to the class that was used to instantiate the object; it's a general term describing the type/kind of any object;
- it's the name of a very handy Python function that returns the class information about the objects passed as arguments to that function;

Information about the types of an object's class is contained in __class__.

Home work 12_1
My Mobile

Create a class representing a mobile phone;

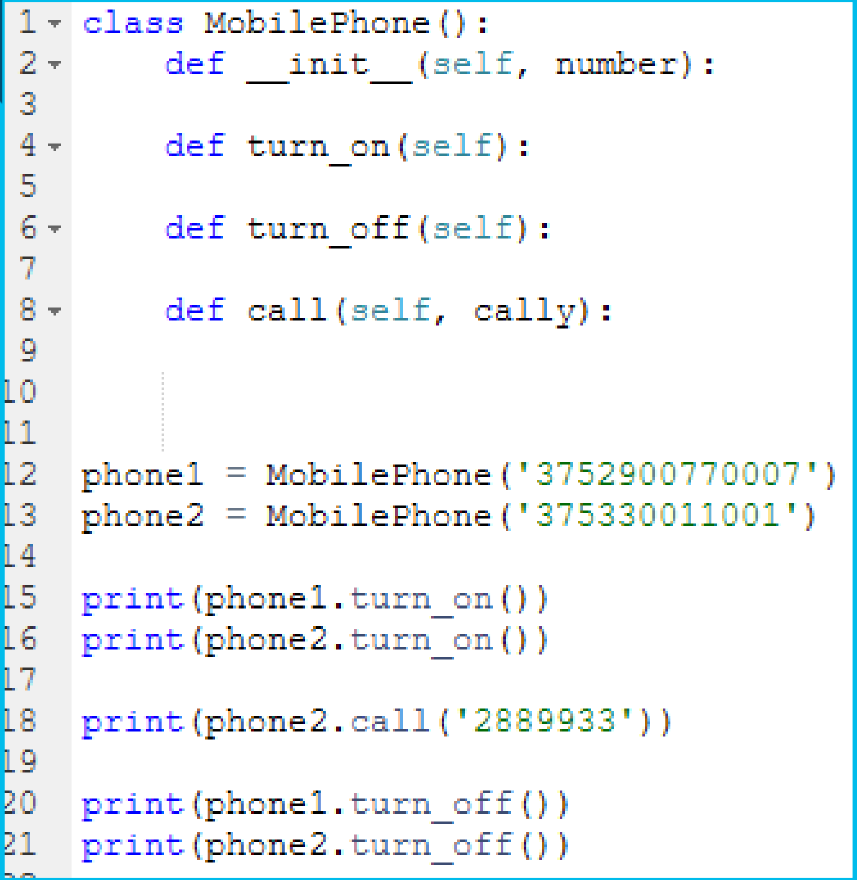
Your class should implement the following methods:

__init__ expects a number to be passed as an argument; this method stores the number in an instance variable **self.number**

turn_on() should **return** the message f'mobile phone {number} is turned on'. Curly brackets are used to mark the place to insert the object's number variable;

turn_off() should **return** the message f'mobile phone {number} is turned off';

call(cally) should **return** the message f'calling {number}'. Curly brackets are used to mark the place to insert the object's number variable;

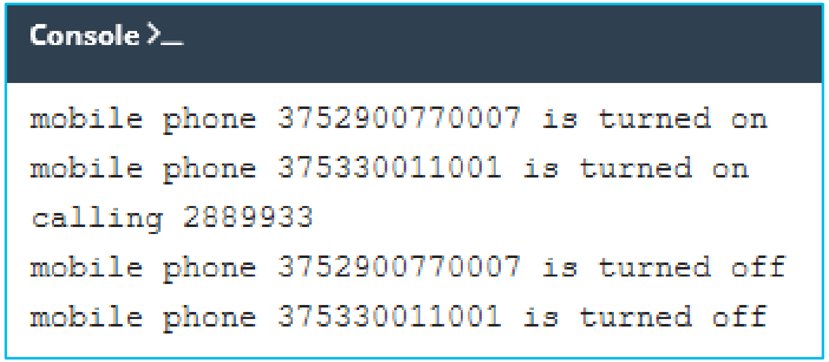


```python
class MobilePhone():
    def __init__(self, number):

    def turn_on(self):

    def turn_off(self):

    def call(self, cally):



phone1 = MobilePhone('3752900770007')
phone2 = MobilePhone('375330011001')

print(phone1.turn_on())
print(phone2.turn_on())

print(phone2.call('2889933'))

print(phone1.turn_off())
print(phone2.turn_off())
```

create two objects representing two different mobile phones; assign any random phone numbers to them;

implement a sequence of method calls on the objects to turn them on, call any number. Print the methods' outcomes;

turn off both mobiles.

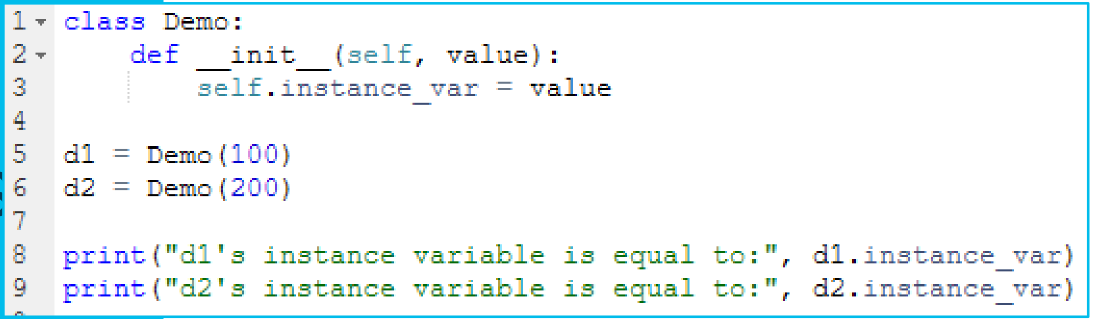


Console >_

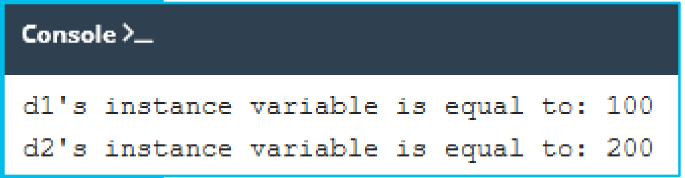
```
mobile phone 3752900770007 is turned on
mobile phone 375330011001 is turned on
calling 2889933
mobile phone 3752900770007 is turned off
mobile phone 375330011001 is turned off
```

Working with class and instance data

instance variable



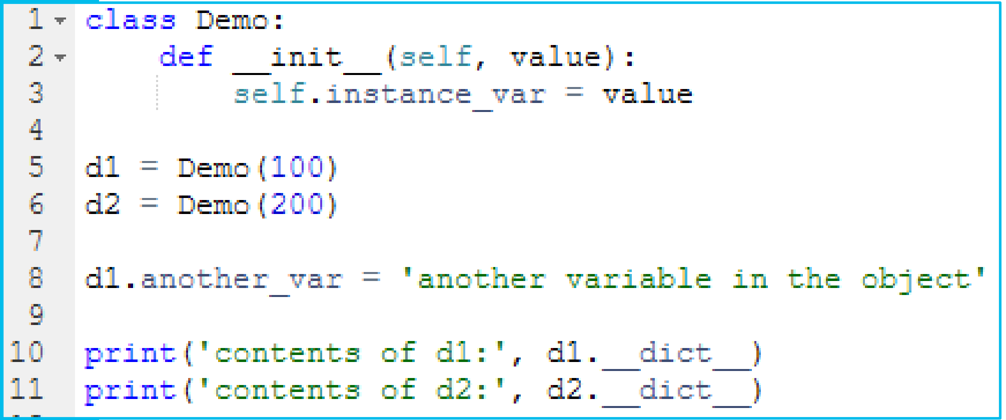
```python
class Demo:
    def __init__(self, value):
        self.instance_var = value

d1 = Demo(100)
d2 = Demo(200)

print("d1's instance variable is equal to:", d1.instance_var)
print("d2's instance variable is equal to:", d2.instance_var)
```



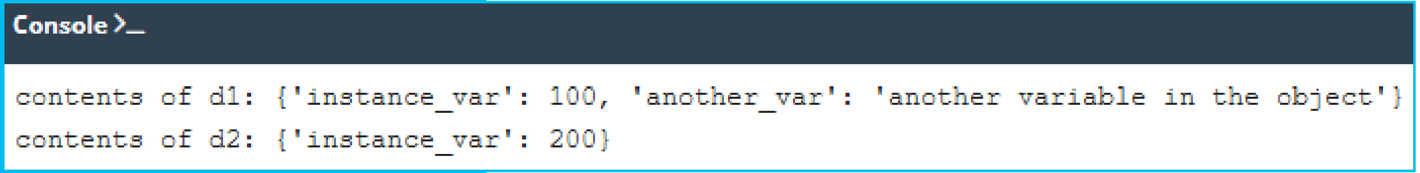
```
d1's instance variable is equal to: 100
d2's instance variable is equal to: 200
```

Working with class and instance data – **__dict__**

it lists the contents of each object, using the built-in __dict__ property that is present for every Python object.



```python
class Demo:
    def __init__(self, value):
        self.instance_var = value

d1 = Demo(100)
d2 = Demo(200)

d1.another_var = 'another variable in the object'

print('contents of d1:', d1.__dict__)
print('contents of d2:', d2.__dict__)
```

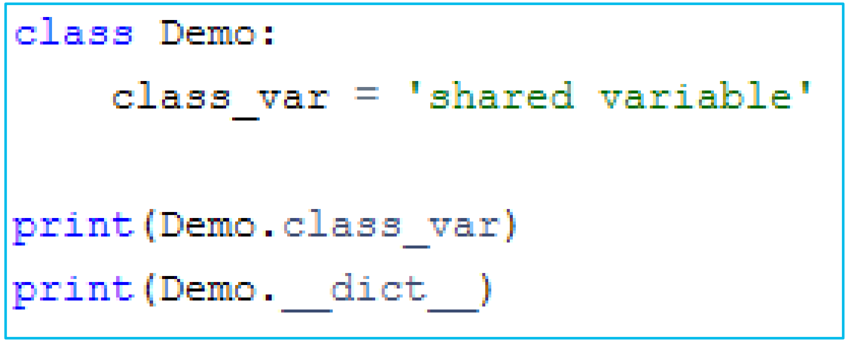


Console >_

```
contents of d1: {'instance_var': 100, 'another_var': 'another variable in the object'}
contents of d2: {'instance_var': 200}
```

Working with class and instance data – **Class variables**

Class variables are defined within the class construction, so these variables are available before any class instance is created.

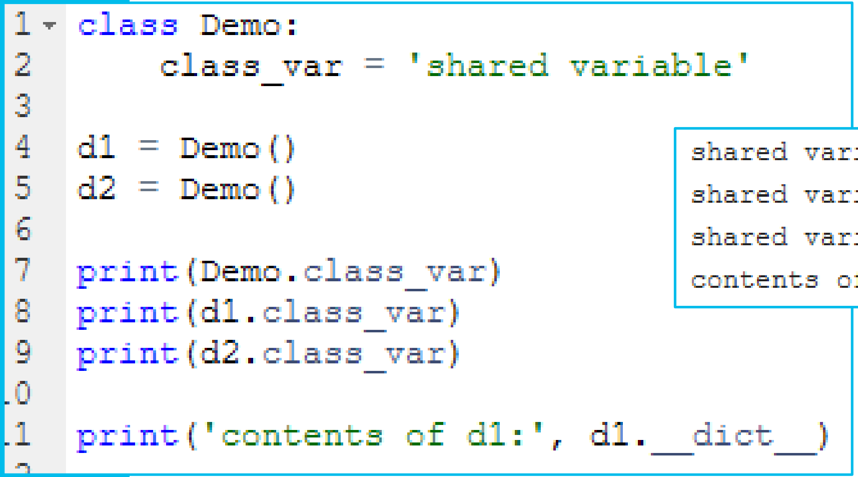


```
class Demo:
    class_var = 'shared variable'

print(Demo.class_var)
print(Demo.__dict__)
```

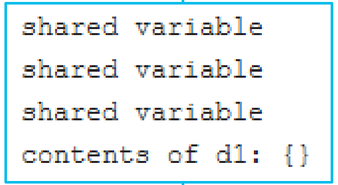
```
shared variable
{'__module__': '__main__', 'class_var': 'shared
variable', '__dict__': <attribute '__dict__' of 'Demo'
objects>, '__weakref__': <attribute '__weakref__'
of 'Demo' objects>, '__doc__': None}
```

Working with class and instance data – **Class variables**

Because it is owned by the class itself, all class variables are shared by all instances of the class. They will therefore generally have the same value for every instance; but the class variable is defined outside the object, it is not listed in the object's __dict__



```python
class Demo:
    class_var = 'shared variable'

d1 = Demo()
d2 = Demo()

print(Demo.class_var)
print(d1.class_var)
print(d2.class_var)

print('contents of d1:', d1.__dict__)
```

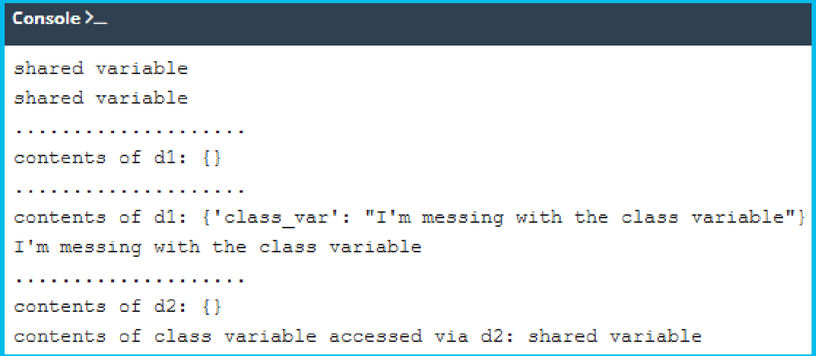


```
shared variable
shared variable
shared variable
contents of d1: {}
```

Conclusion: when you want to read the class variable value, you can use a class or class instance to access it.

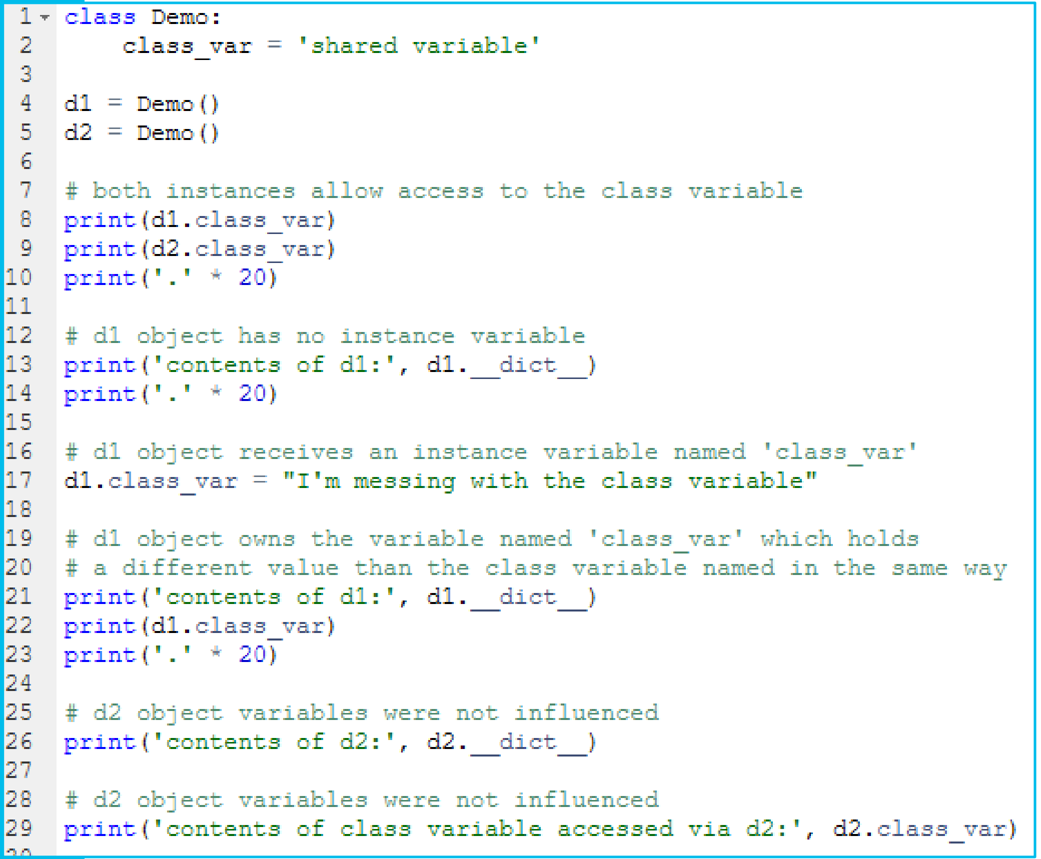
- class_var = 4
- Demo.class_var = 99
- d1.class_var = 99
- self.class_var = 99

Working with class and instance data – **Class variables**



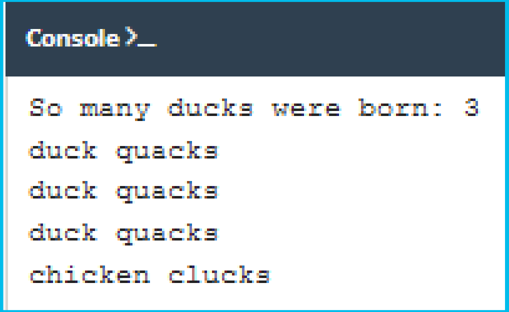
Console >_

```
shared variable
shared variable
....................
contents of d1: {}
....................
contents of d1: {'class_var': "I'm messing with the class variable"}
I'm messing with the class variable
....................
contents of d2: {}
contents of class variable accessed via d2: shared variable
```

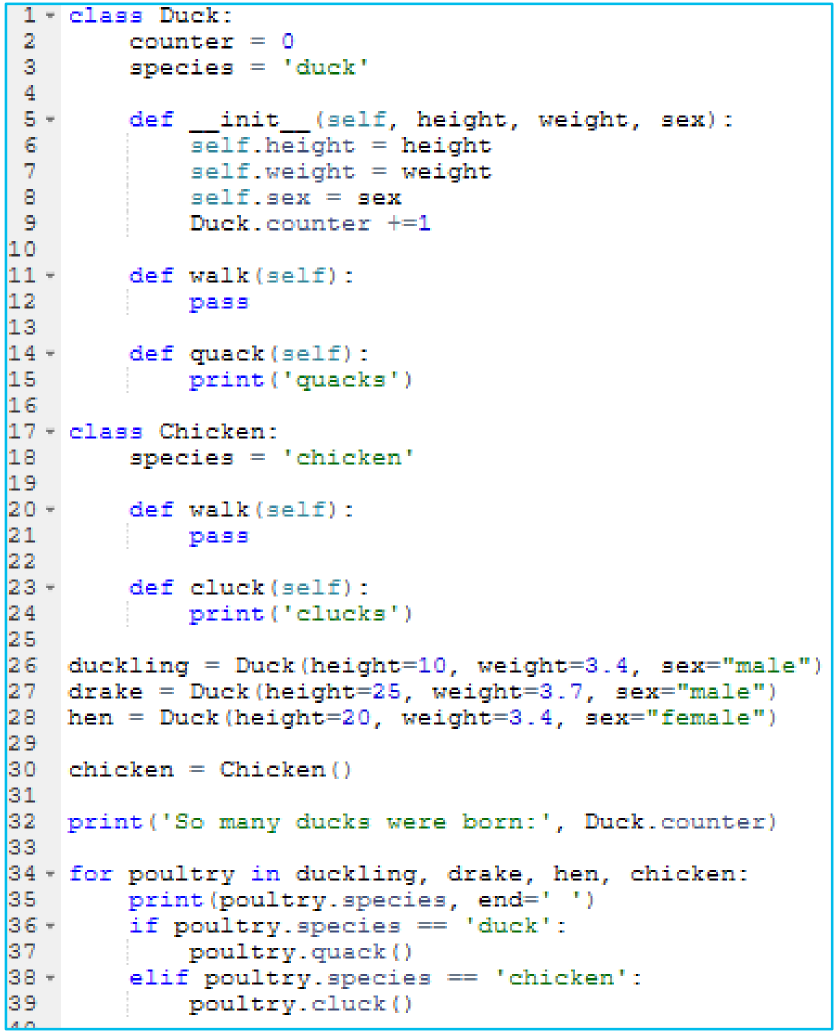


```python
class Demo:
    class_var = 'shared variable'

d1 = Demo()
d2 = Demo()

# both instances allow access to the class variable
print(d1.class_var)
print(d2.class_var)
print('.' * 20)

# d1 object has no instance variable
print('contents of d1:', d1.__dict__)
print('.' * 20)

# d1 object receives an instance variable named 'class_var'
d1.class_var = "I'm messing with the class variable"

# d1 object owns the variable named 'class_var' which holds
# a different value than the class variable named in the same way
print('contents of d1:', d1.__dict__)
print(d1.class_var)
print('.' * 20)

# d2 object variables were not influenced
print('contents of d2:', d2.__dict__)

# d2 object variables were not influenced
print('contents of class variable accessed via d2:', d2.class_var)
```

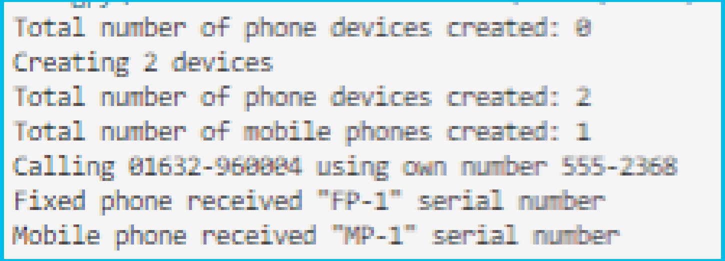
Working with class and instance data



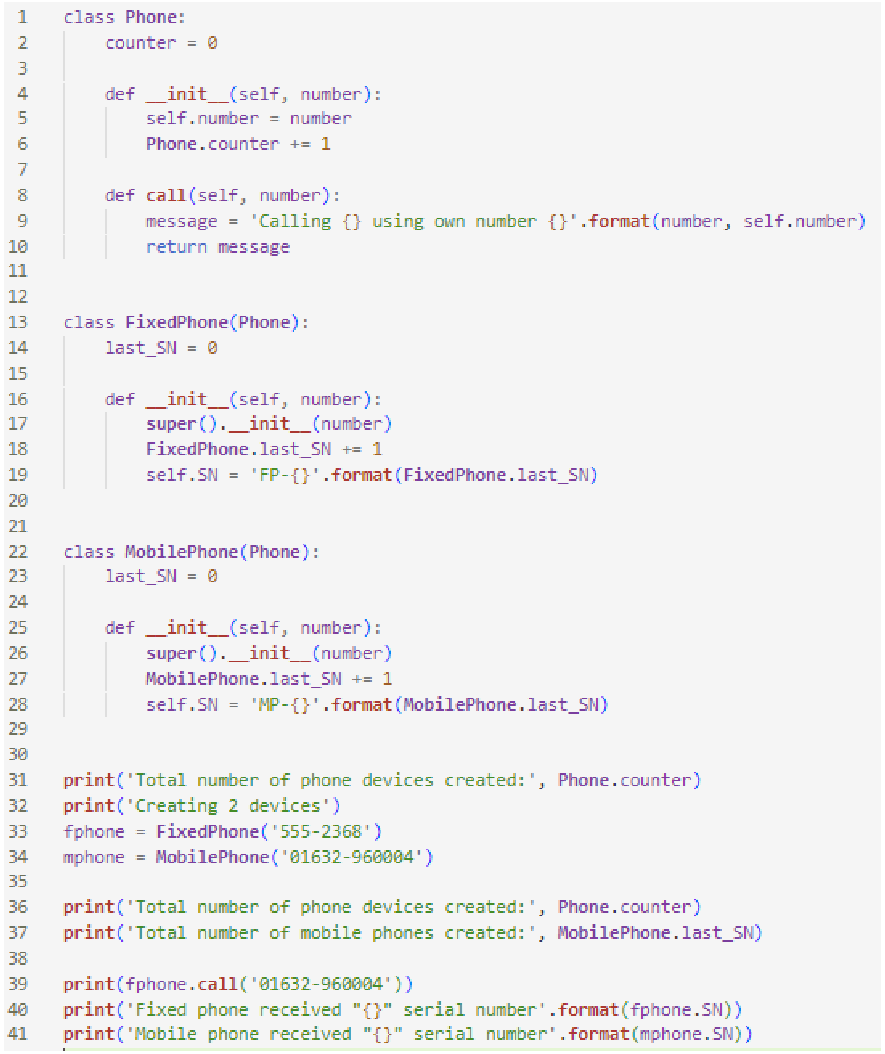
```
Console >_
So many ducks were born: 3
duck quacks
duck quacks
duck quacks
chicken clucks
```



```python
class Duck:
    counter = 0
    species = 'duck'

    def __init__(self, height, weight, sex):
        self.height = height
        self.weight = weight
        self.sex = sex
        Duck.counter +=1

    def walk(self):
        pass

    def quack(self):
        print('quacks')

class Chicken:
    species = 'chicken'

    def walk(self):
        pass

    def cluck(self):
        print('clucks')

duckling = Duck(height=10, weight=3.4, sex="male")
drake = Duck(height=25, weight=3.7, sex="male")
hen = Duck(height=20, weight=3.4, sex="female")

chicken = Chicken()

print('So many ducks were born:', Duck.counter)

for poultry in duckling, drake, hen, chicken:
    print(poultry.species, end=' ')
    if poultry.species == 'duck':
        poultry.quack()
    elif poultry.species == 'chicken':
        poultry.cluck()
```



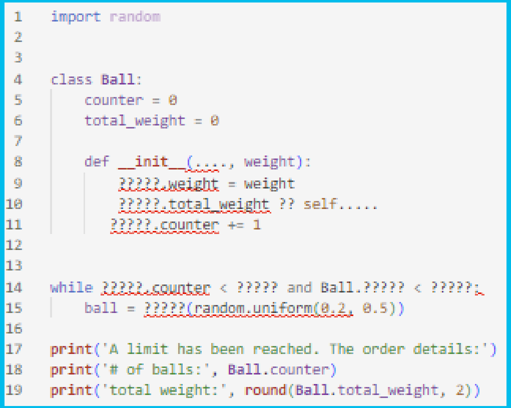
```
Total number of phone devices created: 0
Creating 2 devices
Total number of phone devices created: 2
Total number of mobile phones created: 1
Calling 01632-960004 using own number 555-2368
Fixed phone received "FP-1" serial number
Mobile phone received "MP-1" serial number
```



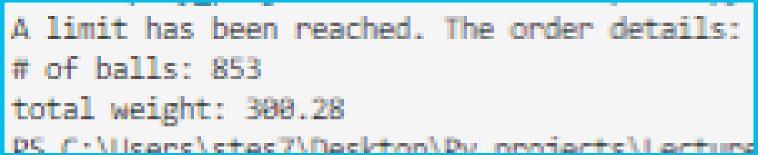
```python
class Phone:
    counter = 0

    def __init__(self, number):
        self.number = number
        Phone.counter += 1

    def call(self, number):
        message = 'Calling {} using own number {}'.format(number, self.number)
        return message


class FixedPhone(Phone):
    last_SN = 0

    def __init__(self, number):
        super().__init__(number)
        FixedPhone.last_SN += 1
        self.SN = 'FP-{}'.format(FixedPhone.last_SN)


class MobilePhone(Phone):
    last_SN = 0

    def __init__(self, number):
        super().__init__(number)
        MobilePhone.last_SN += 1
        self.SN = 'MP-{}'.format(MobilePhone.last_SN)


print('Total number of phone devices created:', Phone.counter)
print('Creating 2 devices')
fphone = FixedPhone('555-2368')
mphone = MobilePhone('01632-960004')

print('Total number of phone devices created:', Phone.counter)
print('Total number of mobile phones created:', MobilePhone.last_SN)

print(fphone.call('01632-960004'))
print('Fixed phone received "{}" serial number'.format(fphone.SN))
print('Mobile phone received "{}" serial number'.format(mphone.SN))
```

Home work 12_2 Balls weight

counter
total_weight
total_balls
self.weight
while loop



```
import random


class Ball:
    counter = 0
    total_weight = 0

    def __init__(...., weight):
        ?????.weight = weight
        ?????.total_weight ?? self.....
       ?????.counter += 1


while ?????.counter < ????? and Ball.????? < ?????:
    ball = ?????(random.uniform(0.2, 0.5))

print('A limit has been reached. The order details:')
print('# of balls:', Ball.counter)
print('total weight:', round(Ball.total_weight, 2))
```



```
A limit has been reached. The order details:
# of balls: 853
total weight: 300.28
PS C:\Users\stes7\Desktop\Py projects\Lecture
```

Imagine that you receive a task description of an application that monitors the process of ball packaging before the balls are sent to a shop.

A shop owner has asked for 1000 +1balls, but the total weight limitation cannot exceed 300 +1 units.

Write a code that creates objects representing balls as long as both limitations are met. When any limitation is exceeded, than the packaging process is stopped, and your application should print the number of ball class objects created, and the total weight.

Your application should keep track of two parameters:

- the number of balls processed, stored as a class variable;
- the total weight of the balls processed; stored as a class variable.

Assume that each ball's weight is random, and can vary between 0.2 and 0.5 of an imaginary weight unit;

Hint: Use a random.uniform(lower, upper) function to create a random number between the lower and upper float values.

Python core syntax

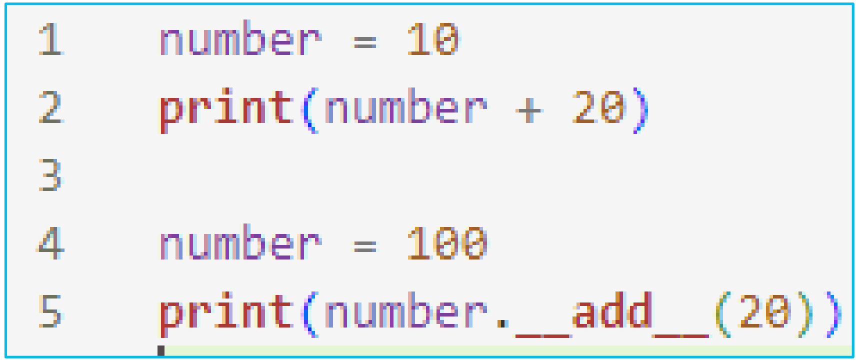
Python core syntax covers:

- operators like '+', '-', '*', '/', '%' and many others;
- operators like '==', '<', '>', '<=', 'in' and many others;
- indexing, slicing, subscripting;
- built-in functions like str(), len()
- reflexion – isinstance(), issubclass()
- and so on.

Python core syntax

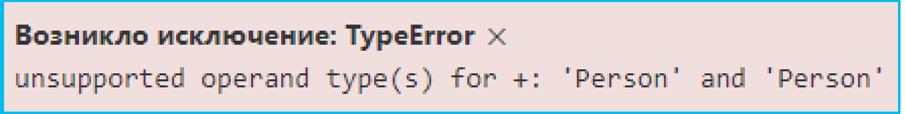
Python core syntax

The **'+' operator** is in fact converted to the **__add__() method** and the **len() function** is converted to the **__len__() method**.

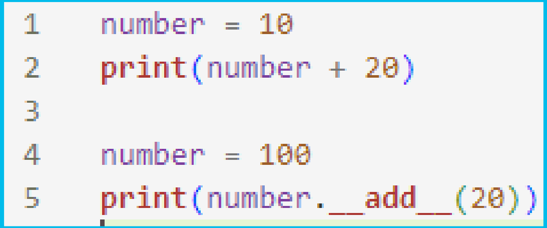


```
number = 10
print(number + 20)

number = 100
print(number.__add__(20))
```

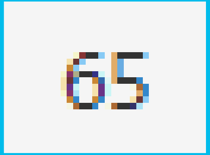
Python core syntax __add__


Возникло исключение: TypeError ×

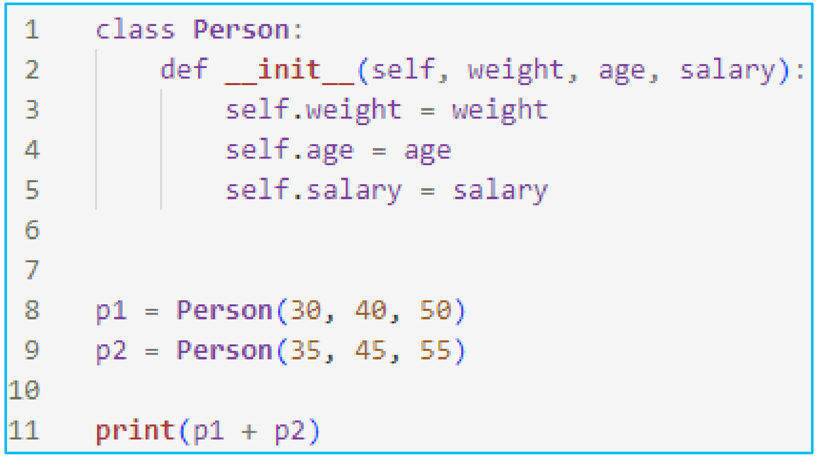
```
unsupported operand type(s) for +: 'Person' and 'Person'
```



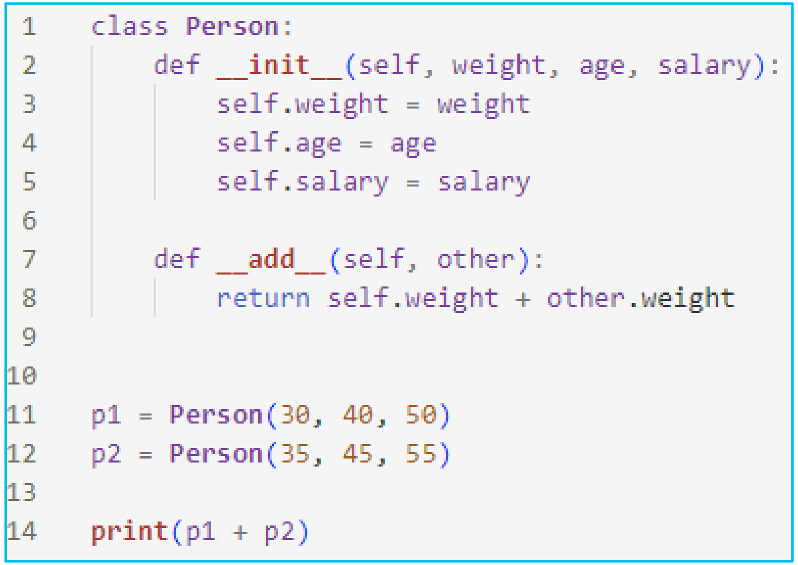
```python
number = 10
print(number + 20)

number = 100
print(number.__add__(20))
```



```
65
```



```python
class Person:
    def __init__(self, weight, age, salary):
        self.weight = weight
        self.age = age
        self.salary = salary


p1 = Person(30, 40, 50)
p2 = Person(35, 45, 55)

print(p1 + p2)
```



```python
class Person:
    def __init__(self, weight, age, salary):
        self.weight = weight
        self.age = age
        self.salary = salary

    def __add__(self, other):
        return self.weight + other.weight


p1 = Person(30, 40, 50)
p2 = Person(35, 45, 55)

print(p1 + p2)
```

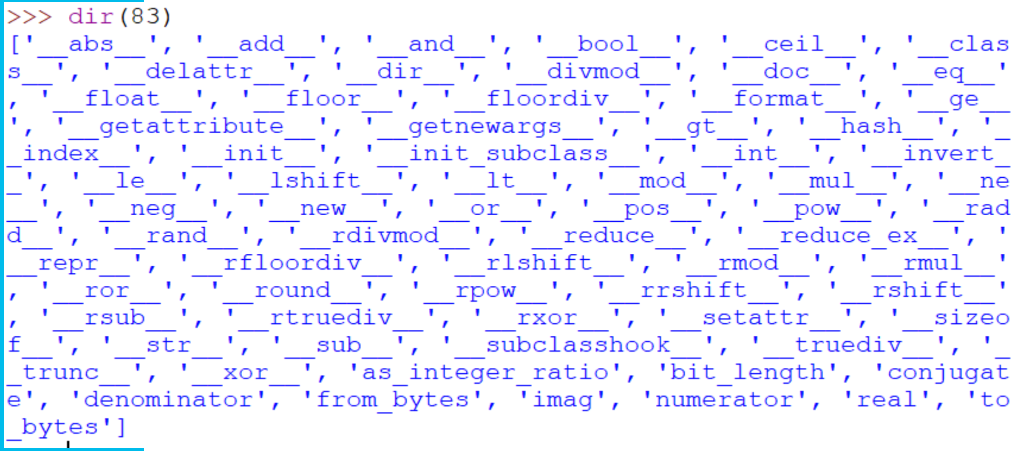
Python core syntax **__add__**

the __**add**__**()** method does not change any object attribute values – it just returns a value that is the result of adding the appropriate attribute values.

the string class has its own implementation for the '**+**' operator, which is inherent to strings, different than implementations inherent to integers or floats.

Python core syntax **dir()**

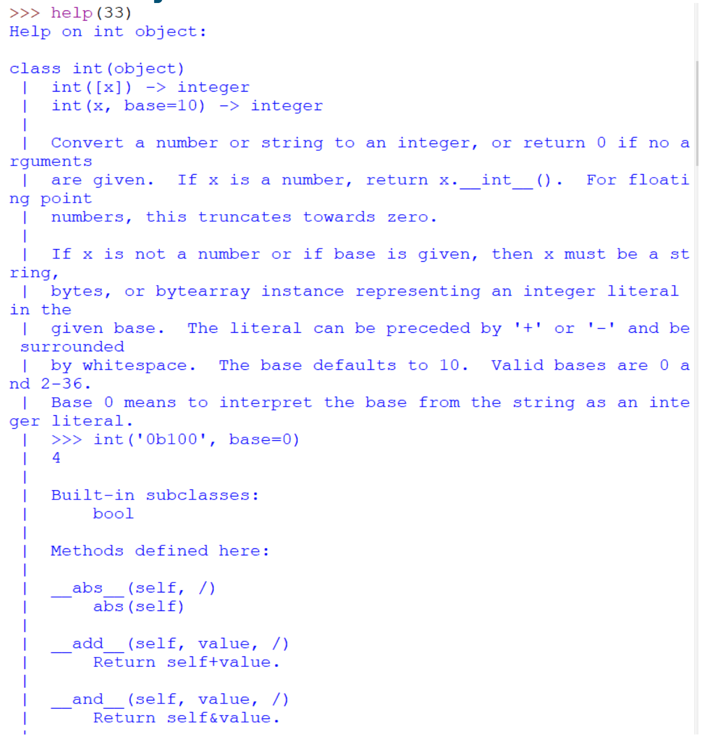
The **dir()** function gives you a quick glance at an object's capabilities and returns a list of the attributes and methods of the object. When you call dir() on integer 83, you'll get:



```
>>> dir(83)
['__abs__', '__add__', '__and__', '__bool__', '__ceil__', '__clas
s__', '__delattr__', '__dir__', '__divmod__', '__doc__', '__eq__'
, '__float__', '__floor__', '__floordiv__', '__format__', '__ge__
', '__getattribute__', '__getnewargs__', '__gt__', '__hash__', '_
_index__', '__init__', '__init_subclass__', '__int__', '__invert_
_', '__le__', '__lshift__', '__lt__', '__mod__', '__mul__', '__ne
__', '__neg__', '__new__', '__or__', '__pos__', '__pow__', '__rad
d__', '__rand__', '__rdivmod__', '__reduce__', '__reduce_ex__', '
__repr__', '__rfloordiv__', '__rlshift__', '__rmod__', '__rmul__'
, '__ror__', '__round__', '__rpow__', '__rrshift__', '__rshift__'
, '__rsub__', '__rtruediv__', '__rxor__', '__setattr__', '__sizeo
f__', '__str__', '__sub__', '__subclasshook__', '__truediv__', '_
_trunc__', '__xor__', 'as_integer_ratio', 'bit_length', 'conjugat
e', 'denominator', 'from_bytes', 'imag', 'numerator', 'real', 'to
_bytes']
```

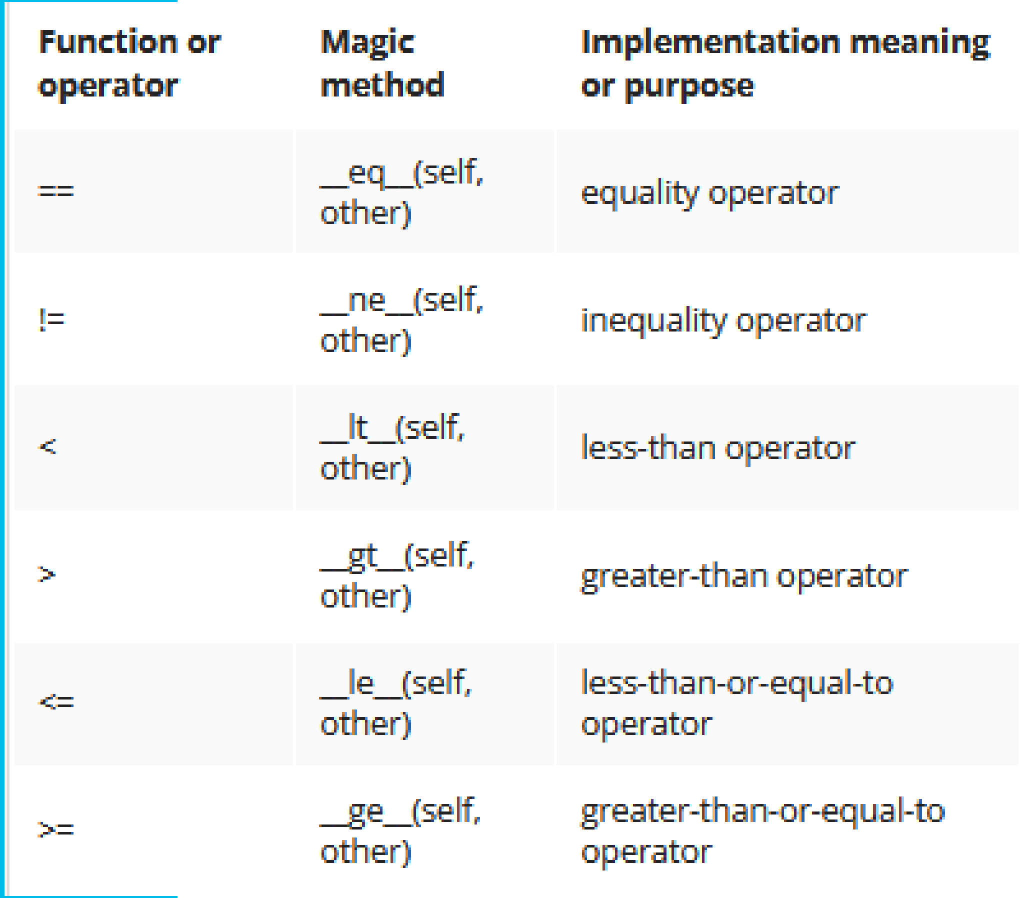
Python core syntax **help()**

To get more help on each attribute and method, issue the **help()** function on an object, as below:



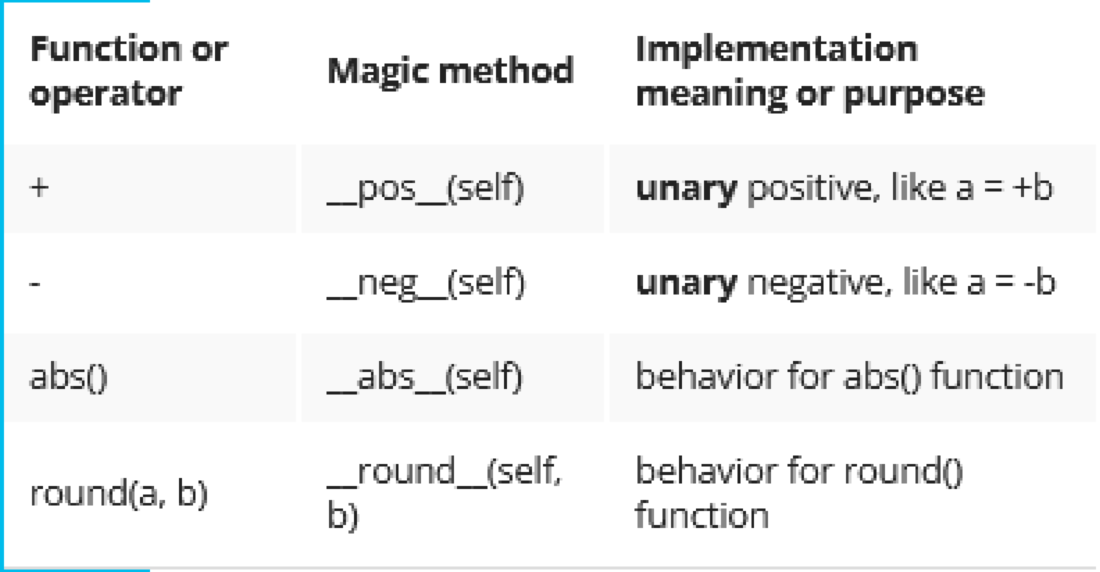
```
>>> help(33)
Help on int object:

class int(object)
 |  int([x]) -> integer
 |  int(x, base=10) -> integer
 |
 |  Convert a number or string to an integer, or return 0 if no a
rguments
 |  are given.  If x is a number, return x.__int__().  For floati
ng point
 |  numbers, this truncates towards zero.
 |
 |  If x is not a number or if base is given, then x must be a st
ring,
 |  bytes, or bytearray instance representing an integer literal
in the
 |  given base.  The literal can be preceded by '+' or '-' and be
 surrounded
 |  by whitespace.  The base defaults to 10.  Valid bases are 0 a
nd 2-36.
 |  Base 0 means to interpret the base from the string as an inte
ger literal.
 |  >>> int('0b100', base=0)
 |  4
 |
 |  Built-in subclasses:
 |      bool
 |
 |  Methods defined here:
 |
 |  __abs__(self, /)
 |      abs(self)
 |
 |  __add__(self, value, /)
 |      Return self+value.
 |
 |  __and__(self, value, /)
 |      Return self&value.
```

Comparison methods



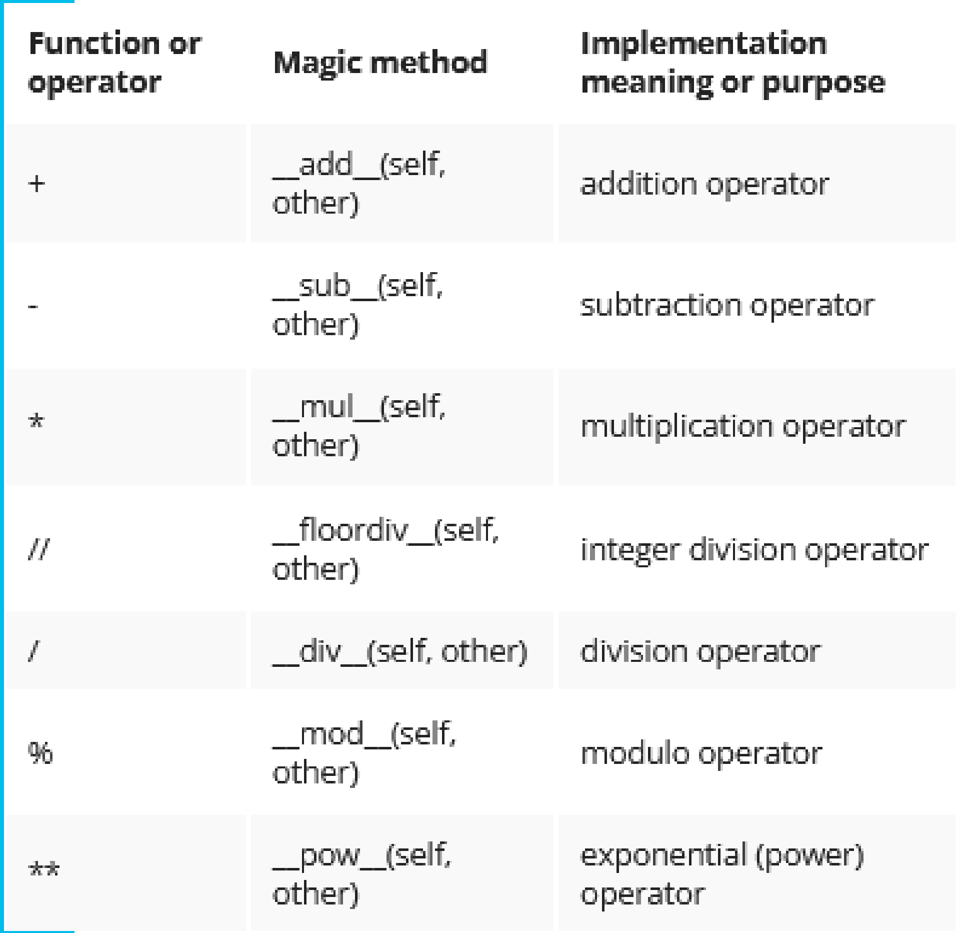
Function or operator	Magic method	Implementation meaning or purpose
==	__eq__(self, other)	equality operator
!=	__ne__(self, other)	inequality operator
<	__lt__(self, other)	less-than operator
>	__gt__(self, other)	greater-than operator
<=	__le__(self, other)	less-than-or-equal-to operator
>=	__ge__(self, other)	greater-than-or-equal-to operator

Numeric methods Unary operators and functions



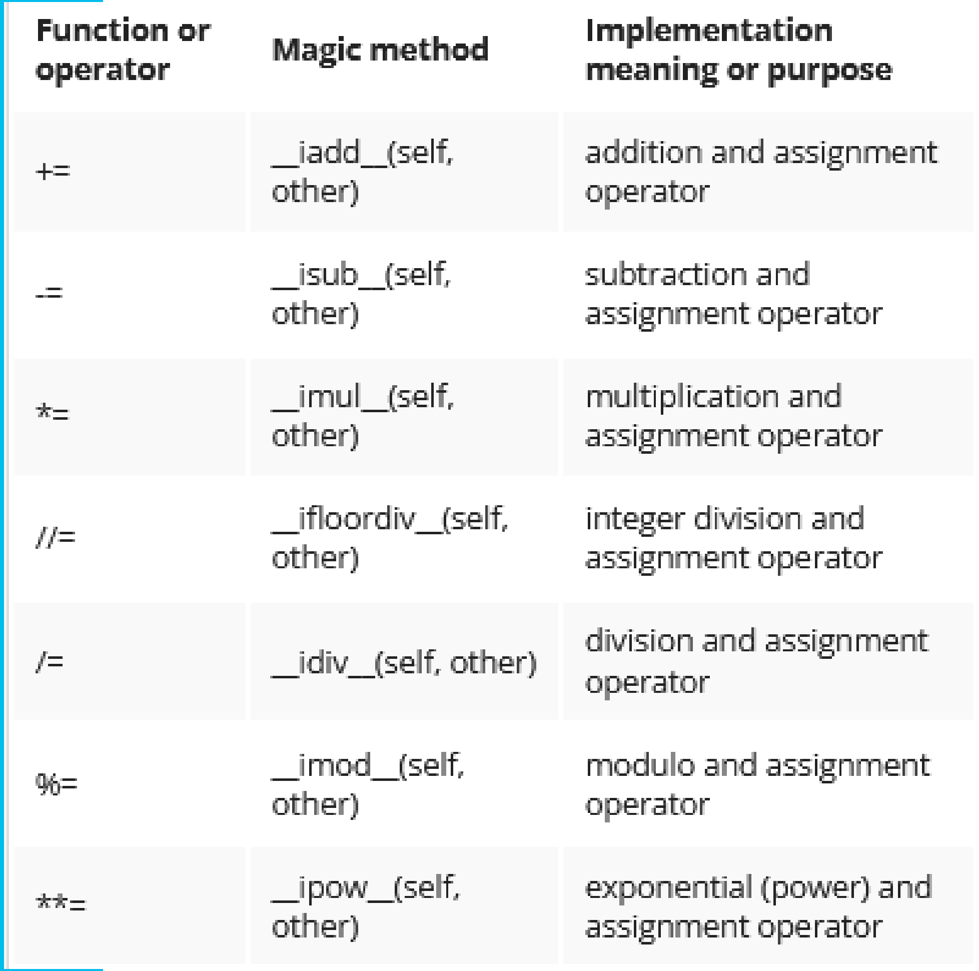
Function or operator	Magic method	Implementation meaning or purpose
+	__pos__(self)	**unary** positive, like a = +b
-	__neg__(self)	**unary** negative, like a = -b
abs()	__abs__(self)	behavior for abs() function
round(a, b)	__round__(self, b)	behavior for round() function

Common, binary operators and functions



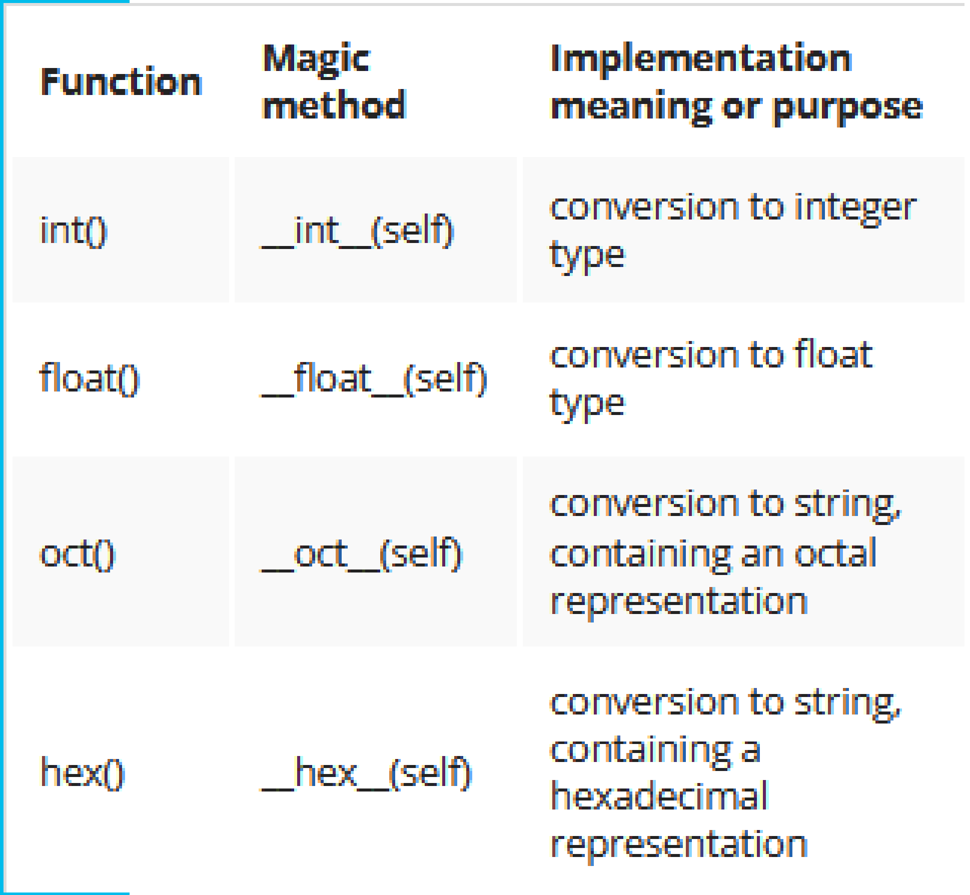
Function or operator	Magic method	Implementation meaning or purpose
+	__add__(self, other)	addition operator
-	__sub__(self, other)	subtraction operator
*	__mul__(self, other)	multiplication operator
//	__floordiv__(self, other)	integer division operator
/	__div__(self, other)	division operator
%	__mod__(self, other)	modulo operator
**	__pow__(self, other)	exponential (power) operator

Extended operators and functions



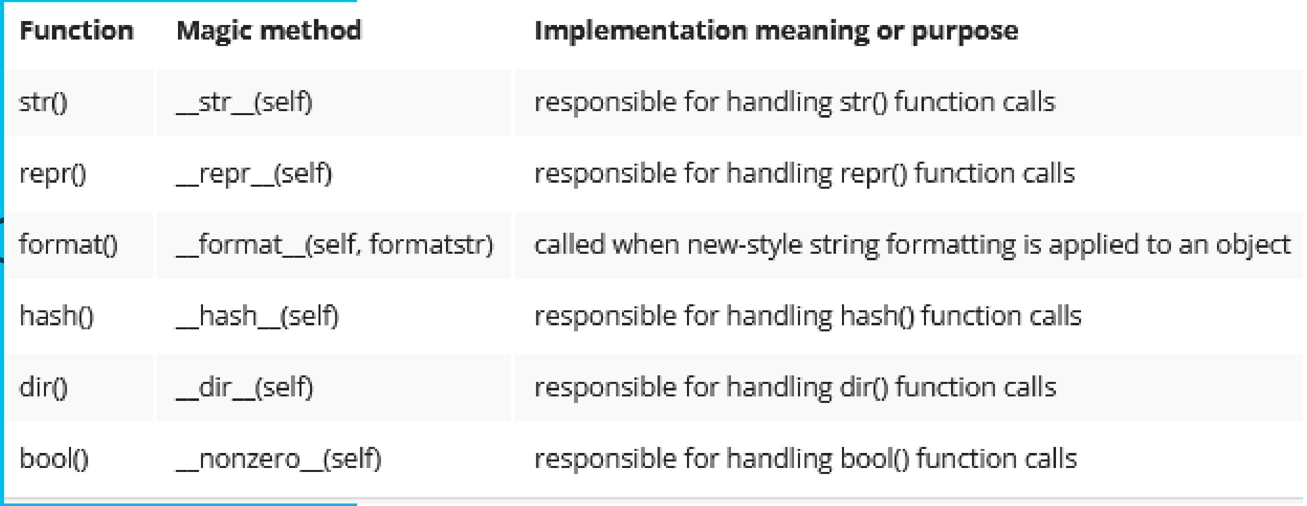
Function or operator	Magic method	Implementation meaning or purpose
+=	__iadd__(self, other)	addition and assignment operator
-=	__isub__(self, other)	subtraction and assignment operator
*=	__imul__(self, other)	multiplication and assignment operator
//=	__ifloordiv__(self, other)	integer division and assignment operator
/=	__idiv__(self, other)	division and assignment operator
%=	__imod__(self, other)	modulo and assignment operator
**=	__ipow__(self, other)	exponential (power) and assignment operator

Type conversion methods



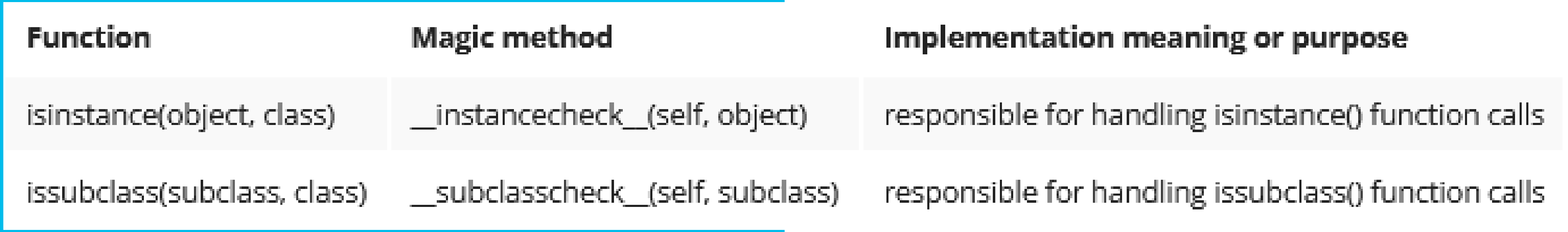
Function	Magic method	Implementation meaning or purpose
int()	__int__(self)	conversion to integer type
float()	__float__(self)	conversion to float type
oct()	__oct__(self)	conversion to string, containing an octal representation
hex()	__hex__(self)	conversion to string, containing a hexadecimal representation

Object introsp



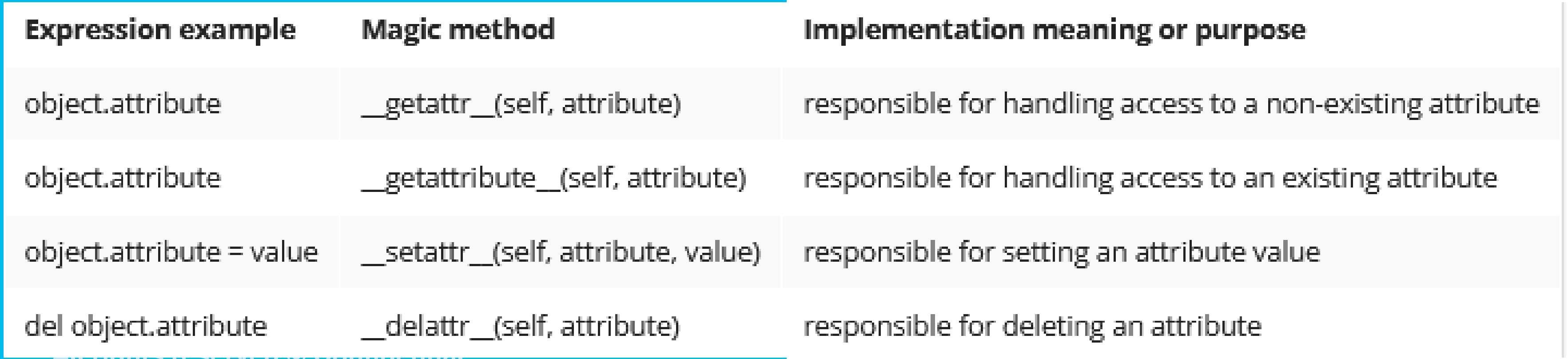
Function	Magic method	Implementation meaning or purpose
str()	__str__(self)	responsible for handling str() function calls
repr()	__repr__(self)	responsible for handling repr() function calls
format()	__format__(self, formatstr)	called when new-style string formatting is applied to an object
hash()	__hash__(self)	responsible for handling hash() function calls
dir()	__dir__(self)	responsible for handling dir() function calls
bool()	__nonzero__(self)	responsible for handling bool() function calls

Object retrospection



Function	Magic method	Implementation meaning or purpose
isinstance(object, class)	__instancecheck__(self, object)	responsible for handling isinstance() function calls
issubclass(subclass, class)	__subclasscheck__(self, subclass)	responsible for handling issubclass() function calls

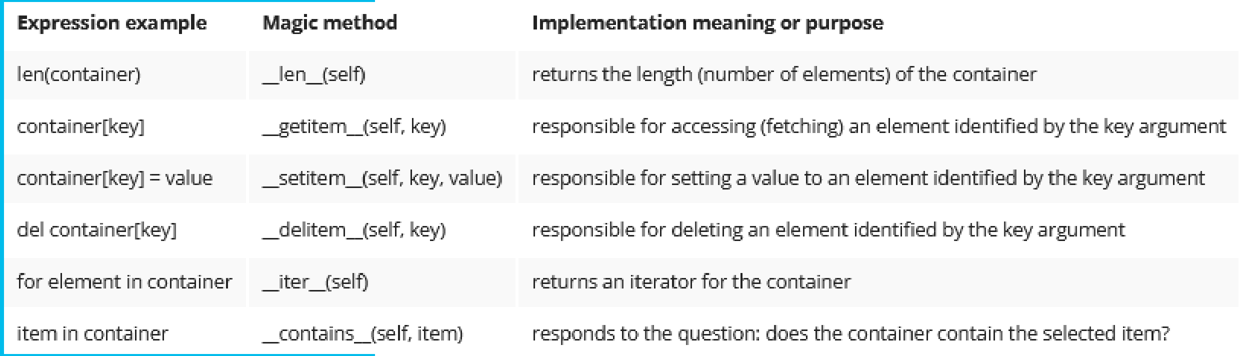
Object attribute access



Expression example	Magic method	Implementation meaning or purpose
object.attribute	__getattr__(self, attribute)	responsible for handling access to a non-existing attribute
object.attribute	__getattribute__(self, attribute)	responsible for handling access to an existing attribute
object.attribute = value	__setattr__(self, attribute, value)	responsible for setting an attribute value
del object.attribute	__delattr__(self, attribute)	responsible for deleting an attribute

Methods allowing access to containers

- Container examples: list, dictionary, tuple, and set.



Expression example	Magic method	Implementation meaning or purpose
len(container)	__len__(self)	returns the length (number of elements) of the container
container[key]	__getitem__(self, key)	responsible for accessing (fetching) an element identified by the key argument
container[key] = value	__setitem__(self, key, value)	responsible for setting a value to an element identified by the key argument
del container[key]	__delitem__(self, key)	responsible for deleting an element identified by the key argument
for element in container	__iter__(self)	returns an iterator for the container
item in container	__contains__(self, item)	responds to the question: does the container contain the selected item?

Python core syntax

The list of special methods built-in in Python contains more entities. For more information, refer to

https://docs.python.org/3/reference/datamodel.html#special-method-names

Python core syntax

Think of any complex problems that we solve every day like:

- adding time intervals, like: add 21 hours 58 minutes 50 seconds to 1hr 45 minutes 22 seconds;
- adding length measurements expressed in the imperial measurement system, like: add 2 feet 8 inches to 1 yard 1 foot 4 inches.

Home work 12_3
Time interval
isinstance()
__add__
__sub__
__str__

Create a class representing a time interval;

- the class should implement its own method for **addition, subtraction** on time interval class objects;
- the class should implement its own method for **multiplication** of time interval class objects by an integer-type value;
- the **__init__** method should be based on keywords to allow accurate and convenient object initialization, but limit it to hours, minutes, and seconds parameters;
- the **__str__** method should return an HH:MM:SS string, where HH represents hours, MM represents minutes and SS represents the seconds attributes of the time interval object;
- heck the argument type, and in case of a mismatch, raise a TypeError exception.

Home work 12_3 Time interval

t1 = Time(21, 58, 50)
t2 = Time(1, 45, 22)
print(t1 – t2)

HINT1:

- just before doing the math, convert each time interval to a corresponding number of seconds to simplify the algorithm;
- for addition and subtraction, you can use one internal method, as subtraction is just ... negative addition.

Test data:

the first time interval (fti) is hours=21, minutes=58, seconds=50

the second time interval (sti) is hours=1, minutes=45, seconds=22

the expected result of addition (fti + sti) is 23:44:12

the expected result of subtraction (fti - sti) is 20:13:28

the expected result of multiplication (fti * 2) is 43:57:40

Home work 12_3 Time interval assert result == '23:54:54'

you can use the assert statement to validate if the output of the __str__ method applied to a time interval object equals the expected value.

```
If isinstance(first_time_inter, int):
        dsjfsdjfs
Else:
        raise Typeerror(‘’)
```

Home work 12_3*
Time interval ext.
other
if isinstance

- Extend the class implementation prepared in the previous lab to support the addition and subtraction of integers to time interval objects;
- to add an integer to a time interval object means to add seconds;
- to subtract an integer from a time interval object means to remove seconds.

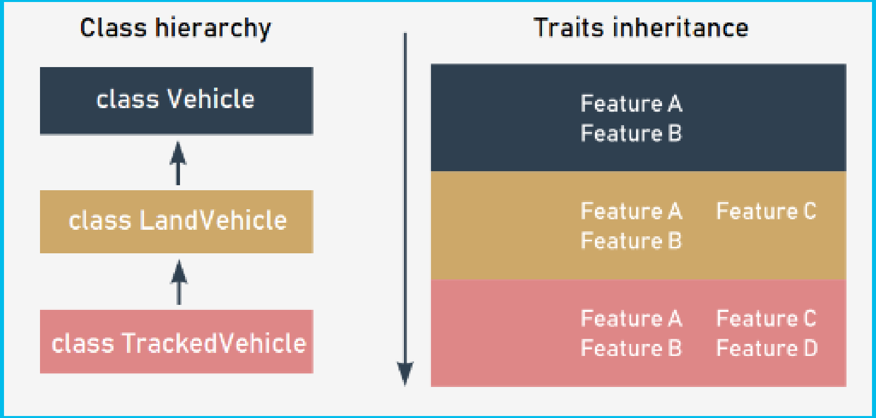
Home work 12_3* Time interval ext.

in the case when a special method receives an integer type argument, instead of a time interval object, create a new time interval object based on the integer value.

Test data:

- the time interval (tti) is hours=21, minutes=58, seconds=50
- the expected result of addition (tti + 62) is 21:59:52
- the expected result of subtraction (tti - 62) is 21:57:48

Inheritance and polymorphism — Inheritance is a pillar of OOP



A very simple example of two-level inheritance is presented here:

```
class Vehicle:
    pass

class LandVehicle(Vehicle):
    pass

class TrackedVehicle(LandVehicle):
    pass
```

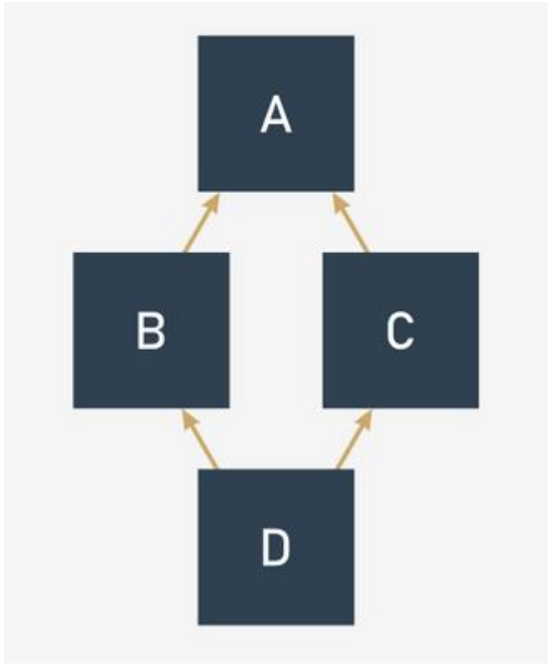
Inheritance and polymorphism — Single inheritance vs. multiple inheritance

- a single inheritance class is always simpler, safer, and easier to understand and maintain;
- multiple inheritance may make method overriding somewhat tricky; moreover, using the super() function can lead to ambiguity;
- it is highly probable that by implementing multiple inheritance you are violating the single responsibility principle;

If your solution tends to require multiple inheritance, it might be a good idea to think about implementing composition.

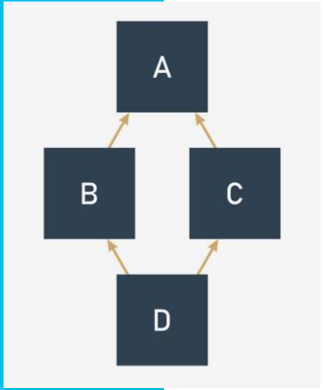
MRO — Method Resolution Order

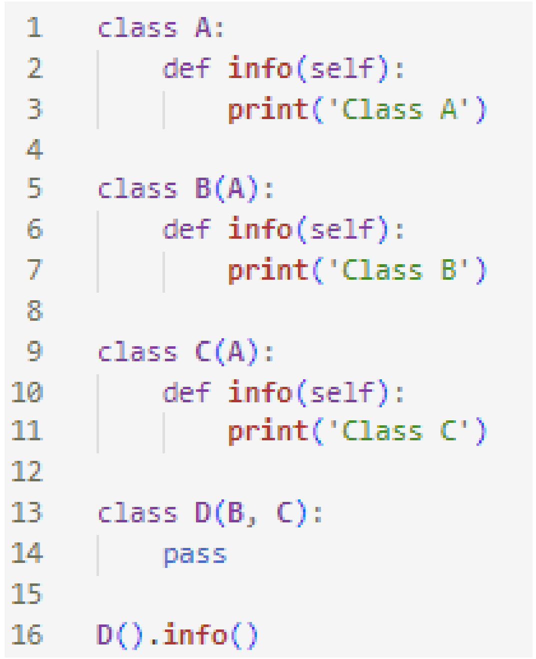
The spectrum of issues possibly coming from multiple inheritance is illustrated by a classical problem named the **diamond problem**, or even the **deadly diamond of death**.



Inheritance and polymorphism — Single inheritance vs. multiple inheritance

In the multiple inheritance scenario, **any specified attribute is searched for first in the current class. If it is not found, the search continues into the direct parent classes in depth-first level (the first level above), from the left to the right, according to the class definition**. This is the result of the **MRO algorithm**.

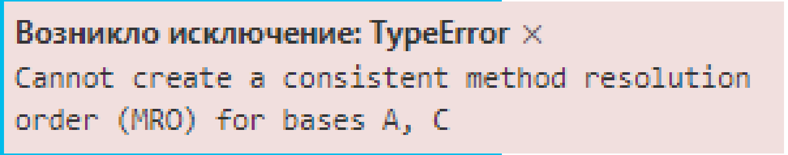




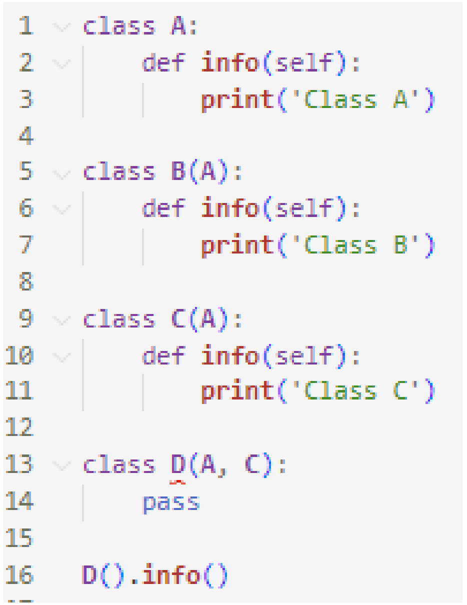
```
class A:
    def info(self):
        print('Class A')

class B(A):
    def info(self):
        print('Class B')

class C(A):
    def info(self):
        print('Class C')

class D(B, C):
    pass

D().info()
```

This message informs us that the MRO algorithm had problems determining which method (originating from the A or C classes) should be called.

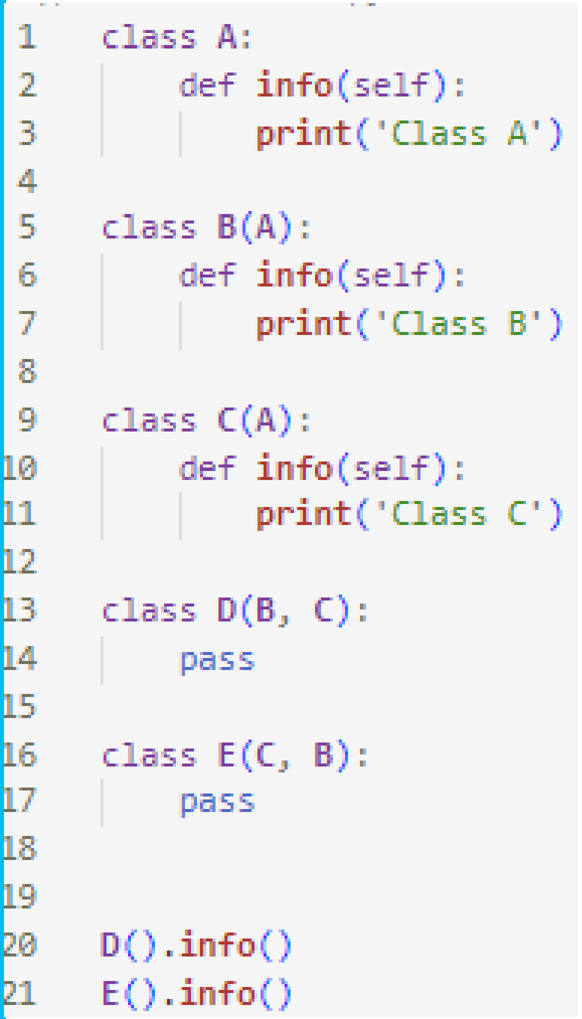
Possible pitfalls — **MRO** inconsistency


Возникло исключение: TypeError ×

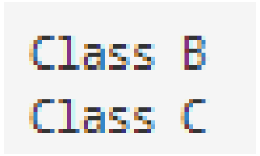
```
Cannot create a consistent method resolution
order (MRO) for bases A, C
```


```
class A:
    def info(self):
        print('Class A')

class B(A):
    def info(self):
        print('Class B')

class C(A):
    def info(self):
        print('Class C')

class D(A, C):
    pass

D().info()
```

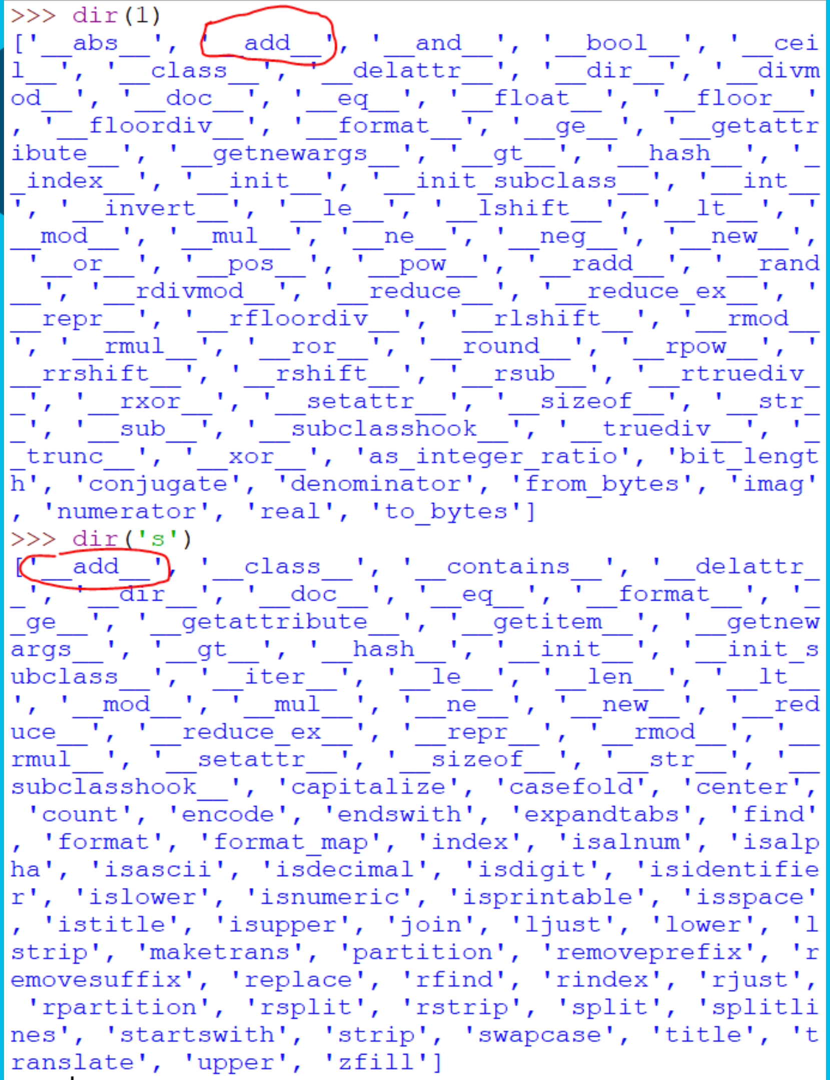
Possible pitfalls — **MRO** inconsistency



```python
class A:
    def info(self):
        print('Class A')

class B(A):
    def info(self):
        print('Class B')

class C(A):
    def info(self):
        print('Class C')

class D(B, C):
    pass

class E(C, B):
    pass


D().info()
E().info()
```

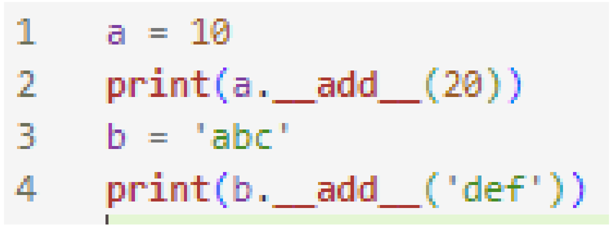


```
Class B
Class C
```

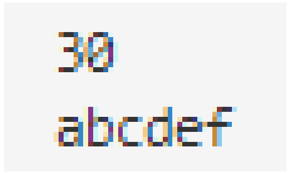


```
>>> dir(1)
['__abs__', '__add__', '__and__', '__bool__', '__cei
l__', '__class__', '__delattr__', '__dir__', '__divm
od__', '__doc__', '__eq__', '__float__', '__floor__'
, '__floordiv__', '__format__', '__ge__', '__getattr
ibute__', '__getnewargs__', '__gt__', '__hash__', '_
_index__', '__init__', '__init_subclass__', '__int__
', '__invert__', '__le__', '__lshift__', '__lt__', '
__mod__', '__mul__', '__ne__', '__neg__', '__new__',
 '__or__', '__pos__', '__pow__', '__radd__', '__rand
__', '__rdivmod__', '__reduce__', '__reduce_ex__', '
__repr__', '__rfloordiv__', '__rlshift__', '__rmod__
', '__rmul__', '__ror__', '__round__', '__rpow__', '
__rrshift__', '__rshift__', '__rsub__', '__rtruediv_
_', '__rxor__', '__setattr__', '__sizeof__', '__str_
_', '__sub__', '__subclasshook__', '__truediv__', '_
_trunc__', '__xor__', 'as_integer_ratio', 'bit_lengt
h', 'conjugate', 'denominator', 'from_bytes', 'imag'
, 'numerator', 'real', 'to_bytes']
>>> dir('s')
['__add__', '__class__', '__contains__', '__delattr_
_', '__dir__', '__doc__', '__eq__', '__format__', '_
_ge__', '__getattribute__', '__getitem__', '__getnew
args__', '__gt__', '__hash__', '__init__', '__init_s
ubclass__', '__iter__', '__le__', '__len__', '__lt__
', '__mod__', '__mul__', '__ne__', '__new__', '__red
uce__', '__reduce_ex__', '__repr__', '__rmod__', '__
rmul__', '__setattr__', '__sizeof__', '__str__', '__
subclasshook__', 'capitalize', 'casefold', 'center',
 'count', 'encode', 'endswith', 'expandtabs', 'find'
, 'format', 'format_map', 'index', 'isalnum', 'isalp
ha', 'isascii', 'isdecimal', 'isdigit', 'isidentifie
r', 'islower', 'isnumeric', 'isprintable', 'isspace'
, 'istitle', 'isupper', 'join', 'ljust', 'lower', 'l
strip', 'maketrans', 'partition', 'removeprefix', 'r
emovesuffix', 'replace', 'rfind', 'rindex', 'rjust',
 'rpartition', 'rsplit', 'rstrip', 'split', 'splitli
nes', 'startswith', 'strip', 'swapcase', 'title', 't
ranslate', 'upper', 'zfill']
```

polymorphism is the provision of a single interface to objects of different types. In other words, it is the ability to create abstract methods from specific types in order to treat those types in a uniform way.

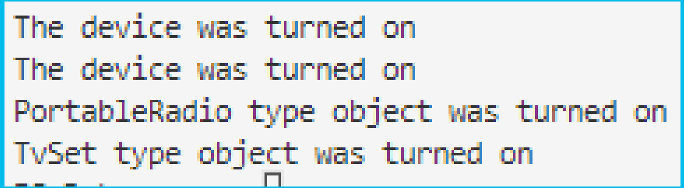


```python
a = 10
print(a.__add__(20))
b = 'abc'
print(b.__add__('def'))
```

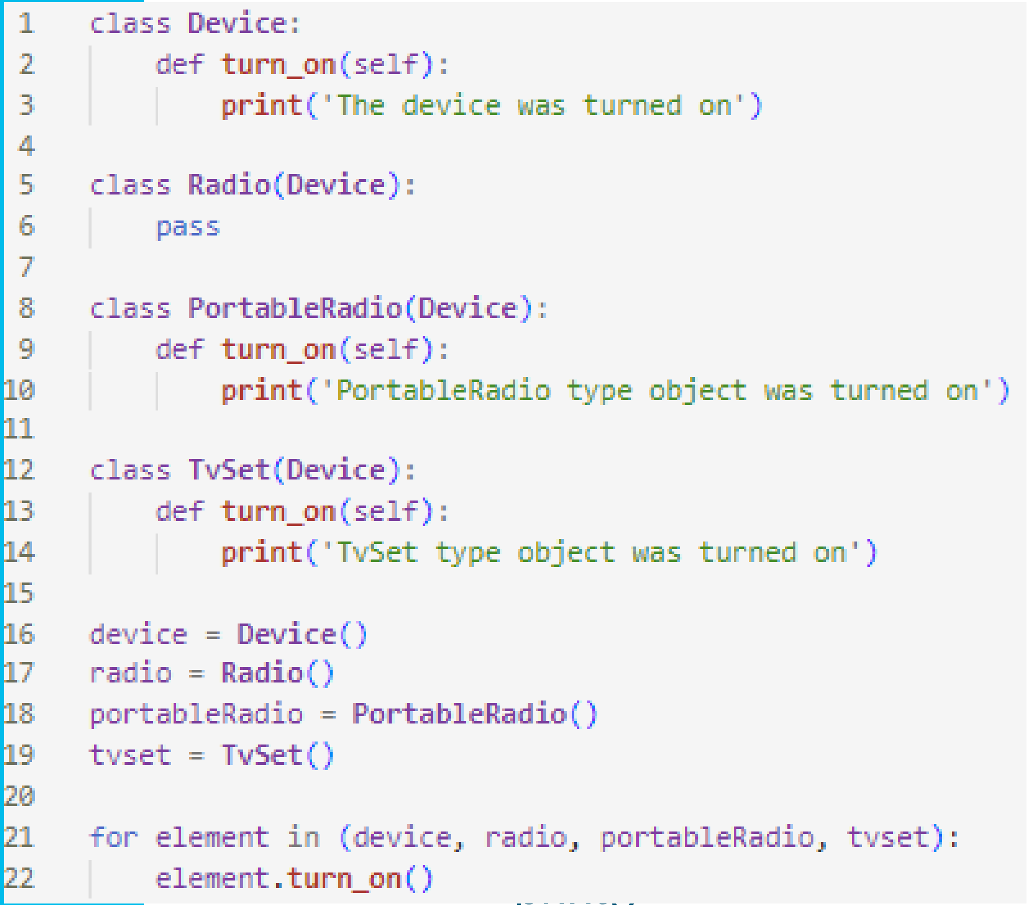


```
30
abcdef
```

Inheritance and polymorphism — two pillars of OOP combined

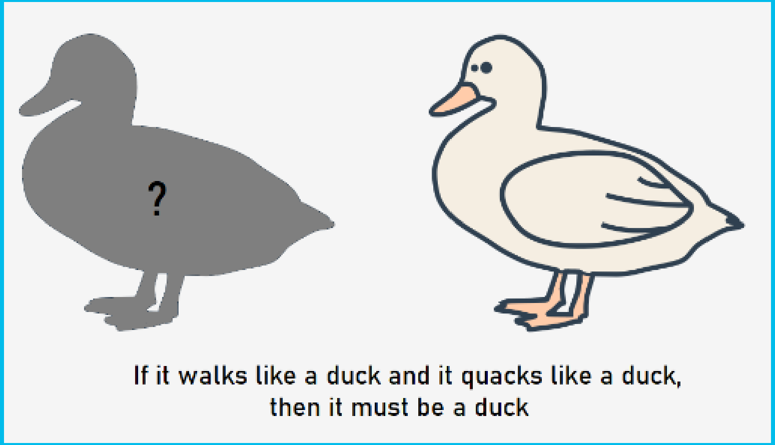


```
The device was turned on
The device was turned on
PortableRadio type object was turned on
TvSet type object was turned on
```



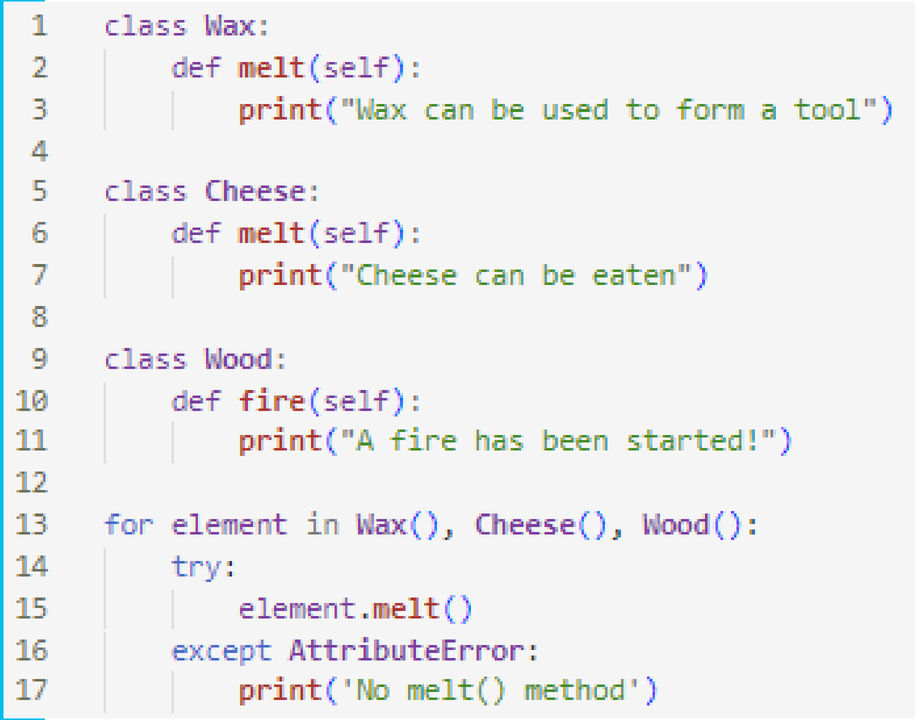
```python
class Device:
    def turn_on(self):
        print('The device was turned on')

class Radio(Device):
    pass

class PortableRadio(Device):
    def turn_on(self):
        print('PortableRadio type object was turned on')

class TvSet(Device):
    def turn_on(self):
        print('TvSet type object was turned on')

device = Device()
radio = Radio()
portableRadio = PortableRadio()
tvset = TvSet()

for element in (device, radio, portableRadio, tvset):
    element.turn_on()
```

Inheritance and polymorphism — duck typing

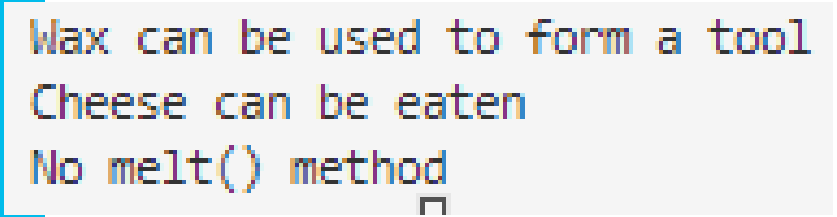


Duck typing is a fancy name for the term describing an application of the duck test: "If it walks like a duck and it quacks like a duck, then it must be a duck", which determines whether an object can be used for a particular purpose. An object's suitability is determined by the presence of certain attributes, rather than by the type of the object itself.

Inheritance and polymorphism — duck typing



```python
class Wax:
    def melt(self):
        print("Wax can be used to form a tool")

class Cheese:
    def melt(self):
        print("Cheese can be eaten")

class Wood:
    def fire(self):
        print("A fire has been started!")

for element in Wax(), Cheese(), Wood():
    try:
        element.melt()
    except AttributeError:
        print('No melt() method')
```



```
Wax can be used to form a tool
Cheese can be eaten
No melt() method
```

Summary

ЗАДАНИЯ

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

Почитать:

1) Byte of Python - стр. 109-120, 71-72, 136-137, 139

Крайний срок сдачи 23/06 в 21:00 (можно раньше, но не позже)

ЗАДАНИЯ

Название файлов, которые вы отправляете мне в telegram:

Vasia_Pupkin_class_work_L12.py
Vasia_Pupkin_L12_1.py, Vasia_Pupkin_L12_2.py,
Vasia_Pupkin_L12_3.py, Vasia_Pupkin_12_3.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 12 про ООП.
И присылаете файлы

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

Q&A

Create your
possibilities.
Bye bye.