2.1 Python core syntax

So far we have been using Python core operations that allow us to operate on strings, lists, integers, and floats. It’s natural for us to formulate expressions using algebraic symbols representing operators, or to get a number of elements in a sequence or dictionary.

We are able to add two or more strings together, which results in the strings’ concatenation; we are able to add integers and we know what the result should be, all done by using the “+” operator, but on respective data types.

Now, we'll use the same function `len()` to get a number of elements of a tuple, list, dictionary, or characters in a string.

This is Python core syntax – an ability to perform specific operations **on different** data types, when operations are formulated using **the same** operators or instructions, or even functions.

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 a few more elements.

Python allows us to employ operators when applied to our objects, so we can use core operators on our objects.

Let's imagine the following situation:

you've created a class that represents a person; each instance of the class owns a set of attributes describing a person: age, salary, weight, etc. What does it mean to add two objects of the Person class? Well, it depends on the domain of your application.

If you’re developing an application for an elevator, then you should remember that every elevator has safety limits regarding the total weight it can lift.

In this case, the '+' operator, when applied to two objects, should mean: add the weight attribute values and return the corresponding result.

If the total sum of the weights of the Person class objects does not exceed the safety limit, your elevator should allow them to be lifted.

When called, functions and operators that state the core syntax are translated into magic methods delivered by specific classes.

So Python knows what to do when the interpreter spots the '+' operator between two strings or two integers, or even two Person class objects – it looks for the magic method responsible for the called function or operator in order to apply it to the operands (objects).

Of course, the '+' operator is just a handy example, and later on we'll discuss other operators.

In Python, these magic methods are sometimes called special purpose methods, because these methods are designated by design to handle specific operations.

The name of each magic method is surrounded by double underscores (Pythonistas would say “dunder” for double underscores, as it’s a shorter and more convenient phrase). Dunders indicate that such methods are not

called directly, but called in a process of expression evaluation, according to Python core syntax rules.

The '+' operator is in fact converted to the add() method and the len() function is converted to the len() method. These methods must be delivered by a class (now it's clear why we treat classes as blueprints) to perform the appropriate action.

Look at the following simple code:

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

It is in fact translated to:

```
number = 10
print(number.__add__(20))
```

Try running these snippets on your own to compare the results for floats and strings.

What Python does is it executes the magic method __add__() on the left operand ("number" object in the example) of the expression, and the right operand (number 20) is passed as a method argument.

Let's get back to our elevator example and try out a way to become compliant with Python core syntax.

First, create a Person class without the magic method responsible for addition:

```
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)
```

output

```
Traceback (most recent call last):
  File "core#010.py", line 11, in
    print(p1 + p2)
TypeError: unsupported operand type(s) for +: 'Person' and 'Person'
```

However, there's a handy special purpose method, add(), which will fix the problem.

Run the code to check our prediction.

```
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) # 65
```

Pay attention to the fact that 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.

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

When you design and implement your own classes and you want to make use of Python core syntax to operate on those class objects, you can easily achieve this by implementing the appropriate magic methods.

You could ask: how can I know what magic method is responsible for a specific operation?

The answer could be: start with the dir() and help() functions.

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 10, you'll get:

output

```
>>> dir(10)
['__abs__', '__add__', '__and__', '__bool__', '__ceil__', '__class__',
'__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__', '__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__', 'bit_length', 'conjugate', 'denominator', 'from_bytes', 'imag',
'numerator', 'real', 'to_bytes']
```

output

```
>>> help(10)
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 arguments
 |  are given.  If x is a number, return x.__int__().  For floating point
 |  numbers, this truncates towards zero.
 |
 |  If x is not a number or if base is given, then x must be a string,
 |  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 and 2-36.
```

```
 |  Base 0 means to interpret the base from the string as an integer literal.
 |  >>> int('0b100', base=0)
 |  4
 |
 |  Methods defined here:
 |
 |  __abs__(self, /)
 |      abs(self)
 |
 |  __add__(self, value, /)
 |      Return self+value.
(...)
```

The names listed above look somehow familiar, so let's see the details delivered by Python itself.

The Python help() function is used to display the documentation (if delivered!) of modules, functions, classes, and keywords. In our example, we have displayed the documentation for the built-in integer type. Look at the last lines of the output – it contains information about the abs and add special methods.

Try to run your own experiments with built-in methods.

Now it’s time to get familiar with Python core syntax expressions and their corresponding magic methods. The following tables will help you in situations where you'd like to implement a custom method for a Python core operation, as the tables enumerate the most popular operators and functions that own magic method counterparts. Treat it as a list from a universal map, unrelated to any special data type.

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

Augumented operators and functions

By augumented assignment we should understand a sequence of unary operators and assignments like `a += 20`

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

Python offers a set of methods responsible for the conversion of built-in data types.

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 introspection

Python offers a set of methods responsible for representing object details using ordinary strings.

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

Following the topic of object introspection, there are methods responsible for object reflection.

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

Access to object attributes can be controlled via the following magic methods

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

Containers are any object that holds an arbitrary number of other objects; containers provide a way to access the contained objects and to iterate over them. 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?

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.

What are some real life cases which could be implemented using special methods?

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.

With classes equipped with custom special methods, the tasks may start to look simpler.

Of course, don't forget to check the type of the objects passed as arguments to special methods. If someone uses your classes, they might use them incorrectly, so a good add method should check whether it's possible to perform the addition before doing it, or else raise an exception.

LAB 1

- 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;

- check the argument type, and in case of a mismatch, raise a TypeError exception.
- Hint 1:
    - 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
- Hint 2:
    - 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.

respuesta

```
class TimeInterval:
    def __init__(self,hours,minutes,seconds):
        # falta check minuts i segons
        self.hours = hours
        self.minutes = minutes
        self.seconds = seconds

    def __add__(self,other):
        total_self = self.hours*60*60+self.minutes*60+self.seconds
        total_other = other.hours*60*60+other.minutes*60+other.seconds
        total = total_self + total_other
        (hours,minutes,seconds) = self.total2values(total)
        print( "hores=%s, minuts=%s, segonds=%s" % (hours,minutes,seconds) )

    def __sub__(self,other):
        total_self = self.hours*60*60+self.minutes*60+self.seconds
        total_other = other.hours*60*60+other.minutes*60+other.seconds
        total = total_self - total_other
        (hours,minutes,seconds) = self.total2values(total)
        print( "hores=%s, minuts=%s, segonds=%s" % (hours,minutes,seconds) )

    def __mul__(self,value):
        total_self = self.hours*60*60+self.minutes*60+self.seconds
        total = total_self * value
        (hours,minutes,seconds) = self.total2values(total)
        print( "hores=%s, minuts=%s, segonds=%s" % (hours,minutes,seconds) )

    def __str__(self):
        return "%s:%02d:%02d" % (self.hours,self.minutes,self.seconds)

    def total2values(self,total):
        seconds = total % 60
        total_minutes = (total - seconds) / 60
        minutes = int (total_minutes % 60)
        hours = int (total_minutes / 60)

        return( (hours,minutes,seconds) )
```

```
fti = TimeInterval(hours=21,minutes=58,seconds=50)
sti = TimeInterval(hours=1,minutes=45,seconds=22)

print(fti)
print(sti)
fti+sti
fti-sti
fti*2
```

output

```
21:58:50
1:45:22
hores=23, minuts=44, segonds=12
hores=20, minuts=13, segonds=28
hores=43, minuts=57, segonds=40
```

LAB 2

- 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.
- Hint 1:
    - 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

resposta

```
class TimeInterval:
    def __init__(self,hours,minutes,seconds):
        # falta check minuts i segons
        self.hours = hours
        self.minutes = minutes
        self.seconds = seconds
        self.total = hours*60*60+minutes*60+seconds

    def __add__(self,value):
        total = self.total + value
        (hours,minutes,seconds) = self.total2values(total)
        print( "hores=%s, minuts=%s, segonds=%s" % (hours,minutes,seconds) )

    def __sub__(self,value):
        total = self.total - value
        (hours,minutes,seconds) = self.total2values(total)
        print( "hores=%s, minuts=%s, segonds=%s" % (hours,minutes,seconds) )
```

```
    def __str__(self):
        return "%s:%02d:%02d" % (self.hours,self.minutes,self.seconds)

    def total2values(self,total):
        seconds = total % 60
        total_minutes = (total - seconds) / 60
        minutes = int (total_minutes % 60)
        hours = int (total_minutes / 60)

        return( (hours,minutes,seconds) )

fti = TimeInterval(hours=21,minutes=58,seconds=50)

print(fti)
fti+62
fti-62
```

output

```
21:58:50
hores=21, minuts=59, segonds=52
hores=21, minuts=57, segonds=48
```