Practical Programming
in Python

Inspired by 'Practical Programming' by Paul Gries, Jennifer Campbell, Jason Montojo

Lecture 7: Using Methods

Types, Classes, Methods, Object Orientation

Kurt Rinnert, Kate Shaw

Physics Without Frontiers

The Abdus Salam
ICTP International Centre
for Theoretical Physics

Abstract

"Any sufficiently advanced technology is indistinguishable from magic."
– Arthur C. Clarke

We have seen functions in several different contexts: functions inside modules and functions we have defined ourselves.
A *method* is a different kind of function that is attached to a particular type.
There are methods attached to `str`, `int` or `float`, for example.
In this lecture we'll learn how to use methods and how they differ from the functions we have seen so far.

Overview

- Modules, classes, methods
- Calling methods
- Object orientation
- Exploring string methods
- Methods special to Python

Methods are efficient and readable. Know the types you use.

Classes

- We learned that modules are objects that can contain functions and other variables
- *Classes* are a similar kind of object
- You have been using classes all along: they are how Python represents types

```
>>> help(str)
class str(object)
|  str(object='') -> str
|  str(bytes_or_buffer[, encoding[, errors]]) -> str
|
|  Create a new string object from the given object. If encoding or
|  errors is specified, then the object must expose a data buffer
|  that will be decoded using the given encoding and error handler.
|  Otherwise, returns the result of object.__str__() (if defined)
|  or repr(object).
|  encoding defaults to sys.getdefaultencoding().
|  errors defaults to 'strict'.
[...]
```

We will explore some methods of the `str` class now.

String Methods

- At the top of the string documentation we find this:

```
|  str(bytes_or_buffer[, encoding[, errors]]) -> str
|
|  Create a new string object from the given object.
```

- This shows how to use `str` as a function
- We can call it to create a string object, for example:

```
>>> str(17)
'17'
>>> str(5.34)
'5.34'
```

We can say that the `str` functions converts other types to strings.

String Methods

- Further down we find this:

```
|  capitalize(...)
|      S.capitalize( -> str
|
|      Return a capitalized version of S, i.e. make the first character
|      have upper case and the rest lower case.
```

- We call a method of class `str` much like a function in module `math`:

```
>>> str.capitalize('turing')
'Turing'
```

Every method in class `str` *requires* a string object as a first argument.

String Methods

- This requirement is not documented because *all* `str` methods require a string as a first argument
- More generally, *all class methods* require an object of the class type as the first argument
- Python programmers are supposed to know this
- Let's look at two more examples from the `str` type:

```
>>> str.center('Sonnet 43', 26)
'        Sonnet 43         '
>>> str.count('How do I love thee?  Let me count the ways.', 'the')
2
```

Check the `str` documentation for these two methods.

The Object Oriented Way

- We learned that *all class methods* require an object of the class type as the first argument
- Python provide a shorthand for this common task where the object appears first and then the method call
- Here we meet the dot operator `.` again – just like when calling functions from a module:

```
>>> 'turing'.capitalize()
'Turing'
>>> 'Sonnet 43'.center(26)
'        Sonnet 43         '
>>> 'How do I love thee?  Let me count the ways.'.count('the')
2
```

Python does the translation for us. We will use this notation from now on.

The Object Oriented Way

- The `help` documentation for methods also uses this form
- Let's have a look at the documentation of the `lower` method of class `str`:

```
>>> help(str.lower)
Help on method_descriptor:

lower(...)
    S.lower() -> str

    Return a copy of the string S converted to lowercase)
```

- Contrast this with the documentation of `math.sqrt`:

```
>>> help(math.sqrt)
Help on built-in function sqrt in module math:

sqrt(...)
    sqrt(x)

    Return the square root of x.
```

Note that the `math.sqrt` documentation shows no prefix object.

What Object Orientation Is

- The term *object oriented* describes a style of programming
- In this style, the objects are the main focus when thinking about solutions to problems
- We tell objects to do things by calling their methods to manipulate their values and communicate with other objects
- This is different from *imperative* or *procedural* programming that focusses on free functions that take objects as arguments
- There are more styles of programming, for example *functional* programming which focusses on functions and their combinations (this one is very interesting and requires quite a different mindset)
- Python supports all of these to some extent
- It is the programmer's – your – job to choose what best fits the problem you want to solve

The Object Oriented Way: Expressions

- The general form of a method call is a follows:

```
expression.method_name(arguments)
```

- So far, we have seen examples where `expression` was a literal object
- But any expression can be used, in particular a variable
- It just has to evaluate to the type that has the method you call:

```
>>> ('TTA' + 'G' * 3).count('T')
2
>>> dna = 'TTA' + 'G' * 3
>>> dna.count('G')
3
```

Methods can be called on the result of an expression, if the result type implements them.

The Object Oriented Way: Evaluation Order

1. Evaluate `expression`, for example:

   ```
   (TTA' + 'G' * 3)
   ```

   This produces a single object object.
2. Now that we have an object, evaluate the method arguments left to right.
   In our DNA example, the argument is `'T'`.
3. Pass the resulting objects into the method.
   In our DNA example, our code is equivalent to:

   ```
   str.count('TTAGGG', 'T')
   ```
4. Execute the method

When the method call finishes, it produces a value.

Spend some time reading about string methods using

```
>>> help(str)
```

or at

https://docs.python.org/3.6/library/stdtypes.html#text-sequence-type-str

Then we will explore some of the methods together

Exploring String Methods

- The method `startswith` takes a string argument and returns **`True`** if the objects it is called for (to the left of the `.`) starts with the argument string:

```
>>> 'species'.startswith('a')
False
>>> 'species'.startswith('spe')
True
```

- There is also an `endswith` method:

```
>>> 'species'.endswith('a')
False
>>> 'species'.endswith('es')
True
```

Methods of built-ins are very efficient. Use them.

- The methods `lstrip`, `rstrip` and `strip` remove whitespace from the front, the back and from both, respectively:

```
>>> compound = '  \n Methyl \n butanol  \n'
>>> compound.lstrip()
'Methyl \n butanol  \n'
>>> compound.rstrip()
'  \n Methyl \n butanol'
>>> compound.strip()
'Methyl \n butanol'
```

- The whitespace *inside* the string is not affected

Exploring String Methods

- The documentation of the `format` method is quite daunting
- A few examples should clear things up

```
>>> 'there are {0} {1}!'.format(4, 'lights')
'there are 4 lights!'
```

- The position numbers can be omitted:

```
>>> 'there are {} {}!'.format(4, 'lights')
'there are 4 lights!'
```

- We use the position numbers to control order:

```
>>> '"{0}" is derived from the {2} "{1}"'.format('December', 'decem', 'Latin')
'"December" is derived from the Latin "decem"'
```

It is good practice to use position numbers.

Exploring String Methods

- We can specify the number of decimal places for a `float` in a replacement field
- This will round the `float` number accordingly:

```
>>> import math
>>> 'Pi rounded to {0} decimal places is {1:.2f}.'.format(2, math.pi)
'Pi rounded to 2 decimal places is 3.14.'
>>> 'Pi rounded to {0} decimal places is {1:.3f}.'.format(3, math.pi)
'Pi rounded to 3 decimal places is 3.142.'
```

- Formatting is also useful for integers:

```
>>> integer_table = '{0:4}\n{1:4}\n{2:4}\n{3:4}'.format(1000, 42, 234, 7)
>>> print(integer_table)
1000
  42
 234
   7
```

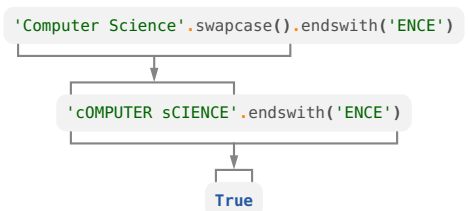
Formatting numbers with the `format` method is very common.

Chaining Method Calls

- Python programmers often *chain* method calls:

```
>>> 'Computer Science'.swapcase().endswith('ENCE')
True
```

- Chained calls are evaluated left to right:



Chaining method calls is useful. Keep your code readable.

Types & Classes

- Classes are the way Python implements types
- This means `int` and `float` are classes, too
- We say objects are *instances* of classes. For example, the integer `42` is an instance of the class `int`
- It is possible to call `help` on an object, rather than the class:

```
>>> help(42)
Help on int object:

class int(object)
 |  int(x=0) -> integer
 |  int(x, base=10) -> integer
[...]
```

Until we learn how to create our own classes we'll use objects of built-in types.

Methods Special to Python

- Let's look further down in the documentation of `int`:

```
>>> help(int)
[...]
 |  __abs__(self, /)
 |      abs(self)
 |
 |  __add__(self, value, /)
 |      Return self+value.
[...]
```

- The underscores `__` signify methods that are special to Python
- These methods are called when a built-in function is called with an instance of the class
- Operators are also functions in Python
- For example, when Python encounters operator **+**,
  it calls the appropriate `__add__` method

This gives some insight into how Python works under the hood.

Methods Special to Python

- To nobody's surprise `str` also provides special methods:

```
>>> help(str)
[...]
 |  __add__(self, value, /)
 |      Return self+value.
[...]
```

- The string method `__add__` is called when we concatenate strings with operator **+**:

```
>>> 'TTA' + 'GGG"
'TTAGGG'
>>> 'TTA'.__add__('GGG')
'TTAGGG'
```

Python programmers almost *never* call these methods directly.

Methods Special to Python

- Let's look at a few more examples for class `int`:

```
>>> abs(-3)
3
>>> (-3).__abs__()
3
```

- Of course, we can also add integers this way:

```
>>> 2 + 7
9
>>> 2 .__add__(7)
9
```

- Note the space added after 2. Why is that necessary?

Keep in mind that Python uses methods to handle all these operators.

Variables Special to Python

- Just like special methods there are special variables
- They are also marked with double underscores `__`
- The variable `__doc__` is automatically created from a docstring
- Let's see how this works when we define a function:

```
>>> def square(x):
...     """
...     Return the square of the number x.
...
...     Examples:
...
...         >>> square(3.0)
...         9.0
...     """
...     return x * x
```

```
>>> print(square.__doc__)
    Return the square of the number x.

    Examples:

        >>> square(3.0)
        9.0
```

- Try `help(square)` and compare

Every object keeps track of its docstring in the variable `__doc__`.

Exercises Lecture 7