Additional Python Tips and Techniques

Contents

Introduction

This is a Jupyter notebook rendered as a page of a Jupyter book. Please see this page for more information on the functionality available.

This section is for users who are already confident in basic Python operation and syntax, but are interested in some more advanced concepts and techniques available in the language. Many of these are not especially useful (by themselves) in numerical codes, but can significantly improve the experience for your users when applied appropriately, and are frequently used in libraries and packages.

None of this material is directly required on the course, but it can certainly be useful.

Python Function Decorators

Python treats functions in a very similar way to how it treats treats objects. You can assign a function a new name

```
def fun1(x):
    print(2*x)

# this prints 20

fun1(10)

fun2 = fun1

#this prints 84
fun2(42)
```

```
20
84
```

you can pass one function into another (in fact, thanks to Python's duck typing, you can also pass a callable class, see the section in the lecture notebooks on operator overloading and magic class methods).

```
import math

def cube_function(func, x):
    return func(x)**3

# These two expressions evaluate the same

print(math.sin(0.5)**3)
print(cube_function(math.sin, 0.5))
```

```
0.11019540730213864
0.11019540730213864
```

In fact, calling one function can even return another function. This isn't something you'd want to do frequently (it's a relatively expensive process) but it can be convenient sometimes. To code up a trivial example (note that we have to assign the docstring separately since we're creating it programatically) :

```
def power_factory(n):
    """This function creates new functions which return x^n for a fixed n."""
    def pow_n(x):
        return x**n
    pow_n.__doc__ = f"Raise x to the power {n}"

    return pow_n


square = power_factory(2)
cube = power_factory(3)

print(square(4))
print(cube(6))
```

```
16
216
```

If we combine the ability to pass functions to functions with the ability to create functions inside functions, we can create utiiity functions which “decorate” a function by “wrapping” it into a new function which both performs the original job and doing some “side effect” at the same time (e.g. logging, caching, profiling or other good stuff like that). Since the utility function is separate, we can update and store it in it’s own module or package if we choose.

Given a decorator, say `my_decorator(func, arg1, arg2)`, we might want to assign it to the same name as the original function with this sort of pattern

```
def my_func(x):
    x += 3
    # do cool stuff to x

my_func = my_decorator(my_func, arg1, arg2)
```

For a long function, this can be easy to miss. to help, Python allows a convenience syntax:

```
@my_decorator(arg1, arg2)
def my_func(x):
    x += 3
    # do cool stuff to x
```

which does exactly the same job.

There are some useful predefined decorators available in the standard Python libraries, such as

- `dataclass` in the `dataclasses` module. This decorator automatically adds `__init__`, `__repr__` and `__eq__` methods to classes based on their data attributes.
- `singledispatch` from `functools` allows C++ style overloading of functions based on the datatype of the arguments.

Magic Class Methods

Thanks to "duck typing" and its inbuilt fallback function evaluation patterns Python class syntax has a number of developer-friendly functionalities it makes available using special or "magic" (also known as "double underscore" or "dunder") class methods. These methods often take the form `__functioname__` (e.g. the class initialization method`__init__`, which you will probably have already seen). A full list of special names can be found in the documentation for the version of Python you are using, but we will highlight some partcularly useful examples.

Operator overloading

As discussed in the 4th presessional notebook, the Python operators include mathematical operations (e.g. `+`, `-`, `*`, `\`) comparison operators (eg. `>`, `==`, `>=`) and bitwise operators (eg. `&`, `|`, `>>`). Functionally, these are implemented on objects by attempting to call special methods on the objects involved, falling back if methods don't exist or fail, so for example `x+y` attempt to call `x.__add_(y)` (or, more strictly `type(x).__add__(x,y)`), then `y.__radd(x)__` if the `__add__` call has returned `NotImplemented`. this means that we can create classes which work with operators

Class creation

Classes are often introduced as being created through calling the class name as a function:

```
class MyClass:
    pass

my_instance = MyClass()
```

This is explained as calling the `__init__` method of the class with the arguments you provide. This is true, but it's not the whole story.

The `__new__` method is actually called first, with the default `object` method then calling the class `__init__` init method. It is this function which responsible for creating the class instance, and is possible to override this method to create classes in a different way, for example by returning an existing instance of the class rather than creating a new one. This is a relatively advanced technique, but can be useful for creating classes which are singletons (i.e. classes only one instance of the class can exist at a time).

Classes created as Python extensions (e.g. via the C API) can do even more exotic and exciting things, but that's beyond the scope of this course.

Context

A common pattern in Python (and coding more generally) is that you want to attempt something with an object, and then clean up your work afterwards. For example, you might want to open a file, read some data from it, and then close it again.

This is such a common pattern that Python provides a special syntax for it, the `with` statement block. The `with` statement is used to create a "context" in which a particular object is available, and then automatically clean up after the context is exited. The syntax is

```
with context_object as name:
    # do stuff with name
```

where `context_object` is an object which has a `__enter__` and `__exit__` method, and the `name` is the name you want to use to refer to the object inside the context. The `__enter__` method is called when the context is entered, and the `__exit__` method is called when the context is exited. The `__exit__` method is called with three arguments, the type, value and

traceback of any exception which caused the context to exit. If the context exited normally, these will all be `None`.

When using files, the `open` function returns a context object which opens the file when the context is entered, and closes it when the context is exited. This means that you can write code like this:

```
with open('myfile.txt') as f:
    f.write('Hello world!\n')
```

Generators

Generically, a Python generator is an object which can be iterated over, but which does not store all of its values in your memory at once, instead calculating them as needed. This is useful for large datasets, or for infinite sequences. The most common way to create a generator is to use a generator expression, which is similar to a list comprehension but with round brackets instead of square brackets:

```
my_generator = (x**2 for x in range(1000000))

for i in my_generator:
    if i%3 == 2:
        print(i)
```

If we used a list here (i.e. `[X**2 for x in range(1000000)]`) we would have to store all 1 million values in memory at once, which would be very inefficient in this contes. Instead, the generator expression creates a generator object which stores the current value of `x` and the expression `x**2`, and calculates the next value of `x` when it is needed.

Type Hints

Python is a dynamically typed language, which means that the type of a variable is not known until runtime. It also leverages “Duck” typing, where code doesn’t care about the precise data type a given object has, but only that it supports the requested attributes or methods.

This is very convenient for the programmer, since it means many things *just work* but can make it difficult for other people to understand your code, or for you to understand your own code if you come back to it after a long time.

Python 3.5 introduced a new syntax for adding type hints to your code, which can be used by external tools to check your code for errors, and to help you understand your code. For example, the following code

```
def integer_remainder(float: y, int: x) -> float:
    return mod(y, x)
```

Note that this is **not** checked when the code is run, and will not cause an error if the types are wrong. It is purely for the benefit of the programmer and any tools they use.

Further Reading

- Python Decorators
- Python Classes
- Python Magic Methods
- Python Generators
- Python Iterators
- Python Type Hints
- Python Context Managers