In/Out: Modules, Files, Objects

Introduction to Python

Data Sciences Institute, University of Toronto
Instructor: A Mahfouz | TA: Kaylie Lau
July 2022

Contents:

Modules

What is a module?

A *module* is a file that contains Python function definitions and executable statements. Just as functions let us reuse code in a prigram, modules let us reuse code across multiple programs. Modules to do related tasks can be collected into a *package*. Later on, we will use packages to work with datasets. You may hear terms module, package, and library used interchangeably -- they all refer to reusable collections of code that make it easier to do certain tasks.

Working with modules

Python comes with several modules built in. To use one, we `import` it. Normally, `import` statements go at the very top of the file -- we're only putting them in the middle here for teaching purposes.

```
In [1]: import math
```

When using code in a module, we first reference the module, follwed by a period, then the value or function being used.

In [2]:
```
math.pi
```

Out[2]:

```
3.141592653589793
```

In [3]:
```
math.gcd(13984, 232, 18356)
```

Out[3]:

```
4
```

Not referencing the module produces an error.

```
In [4]: pi
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Input In [4], in <cell line: 1>()
----> 1 pi

NameError: name 'pi' is not defined
```

```
In [5]: gcd(13984, 232, 18356)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Input In [5], in <cell line: 1>()
----> 1 gcd(13984, 232, 18356)

NameError: name 'gcd' is not defined
```

If we only want a part of a module, we can import parts with a `from...import` statement. This is useful for large modules like `datetime`, where we may only want to use a few features.

In [6]:
```
from datetime import date
```

In [7]:
```
# we don't have to write datetime.date.today()
print(date.today())
```

```
2022-07-13
```

Working with dates

Earlier, we encountered the `datetime` module. `datetime` provides date, time, datetime, and timedelta types, plus ways to manipulate them.

In [8]:
```
# import more submodules from datetime
from datetime import datetime, timedelta
```

Creating dates

The easiest day to create a date is with `date` and `datetime`'s `today()` method. Of course, this is only good so long as we want the current date. We can use the `date()` and `datetime()` functions to create dates in general, passing in integers for year, month, and day, plus an optional timezone argument. For `datetime()`, we can also pass arguments for hour, minute, second, and microsecond.

In [9]:
```
print(datetime.today())
datetime.today()
```

```
2022-07-13 08:01:34.318522
```

Out[9]:

```
datetime.datetime(2022, 7, 13, 8, 1, 34, 318522)
```

In [10]:
```
py_launch = date(1991, 2, 20)
py_launch
```

Out[10]:

```
datetime.date(1991, 2, 20)
```

Accessing date parts

We can access parts of a datetime, like the month or the minute.

In [11]:
```
py_launch.year
```

Out[11]:

```
1991
```

In [12]:
```
datetime.today().minute
```

Out[12]:

```
1
```

We can even do arithmetic with datetime parts, but it's not datetime-specific.

In [13]:
```
# we can add to datetime parts
py_launch.year + 5
```

Out[13]:

```
1996
```

In [14]:
```
datetime.today().minute + 60
```

Out[14]:

```
61
```

Date operations and `timedelta`

We can add and subtract dates from one another with the `+` and `-` operators. The value we get back isn't a date or a time, however, but a `timedelta` -- a different type of data that stores the difference in days, seconds, and microseconds.

In [15]:
```
datetime(2000, 1, 1) - datetime.today()
```

Out[15]:

```
datetime.timedelta(days=-8230, seconds=57505, microseconds=583480)
```

To format the age more nicely, we can access `timedelta`'s `days` part and divide.

In [16]:
```
# calculate how old Python is
py_age = date.today() - date(1991, 2, 20)
print(f'Python is {py_age}')
```

```
Python is 11466 days, 0:00:00
```

In [17]:
```
print(f'Python is {py_age.days//365} years old.')
```

```
Python is 31 years old.
```

If we want to add or subtract a specific amount of time, we can use the `timedelta()` function, passing in as keyword arguments the desired interval.

In [18]:
```
# find the time an hour from now
datetime.today() + timedelta(hours=1)
```

Out[18]:

```
datetime.datetime(2022, 7, 13, 9, 1, 34, 458519)
```

`strptime()` and `strftime()`

Often, we get dates and times as text in a variety of formats. Just as often, we need to format a date or a time for text. `strptime()` and `strftime()` let us convert strings into datetimes and datetimes into formatted strings, respectively. To do so, we specify the datetime format using format codes.

Datetime format codes

https://strftime.org/ has a complete cheat sheet for format codes. The most common codes are below.

Code	Description
%Y	Four-digit year
%m	Zero-padded month
%B	Month name
%A	Day name
%d	Zero-padded day
%H	Zero-padded hour (24-hour clock)
%I	Zero-padded hour (12-hour clock)
%M	Zero-padded minute
%S	Zero-padded second
%p	AM/PM

In [19]:
```
datetime.today().strftime('%H:%M %A %d %B %Y')
```

Out[19]:

```
'08:01 Wednesday 13 July 2022'
```

In [20]:
```
# parse a string to datetime
datetime.strptime("Feb 20 '91", "%b %d '%y")
```

Out[20]:

```
datetime.datetime(1991, 2, 20, 0, 0)
```

Regular expressions

Python also has a module `re`, for *regular expressions*. Regular expressions, or *regex*, can be thought of a mini-language for pattern matching text. Regular expressions give us more powerful tools for finding substrings.

In [21]:
```
import re
```

In [22]:
```
text = '''My favourite thing to do on nice days is program.
Programming is fun as long as there aren't too many bugs!'''
```

regex: a brief guide

There are several special characters in regex. These are some of the most common. Python's documentation provides a **guide to regular expression syntax**.

Character	Meaning
*	Match zero or more of the RE before
+	Match one or more of the RE before
?	Match zero or one of the RE before
\	Escape character
\d	Match any decimal digit
\s	Match any whitespace
\w	Match any word character (letters, digits, underscore)
[aeiou]	Match any of the characters in brackets

Searching for text

`re`'s `search()` function lets us pattern match text. It returns a special match type that tells us what text was matched and where in the string it was found. Note that only the first match is indicated.

In [23]:
```
re.search('favou?rite', text)
```

Out[23]:

```
<re.Match object; span=(3, 12), match='favourite'>
```

In [24]:
```
if re.search('favou?rite', text):
            print('Text includes favourites')
else:
    print('Text is about something else')
```

```
Text includes favourites
```

`findall()` returns all non-overlapping matches of a pattern in a string. The result can be either a list of strings or a list of tuples, depending on the regular expression. Note that we can add in flags -- essentially settings -- for pattern matching.

In [25]:
```
# match words ending in -ing
re.findall('\w*ing', text)
```

Out[25]:

```
['thing', 'Programming']
```

In [26]:
```
# see what happens if we just search for 'ing'
re.findall('ing', text)
```

Out[26]:

```
['ing', 'ing']
```

In [27]:
```
# use re.I for case insensitive matching
re.findall('program', text, flags=re.I)
```

Out[27]:

```
['program', 'Program']
```

Replacing Text

The `sub()` function replaces occurrences of a pattern. `sub()` takes as arguments the pattern to look for, the replacement string, and the string to make replacements in. It also takes an optional `count` argument for how many occurrences to replace and a `flags` argument.

In [28]:
```
# replace 'program' with 'garden'
re.sub('program+', 'garden', text, flags=re.I)
```

Out[28]:

```
"My favourite thing to do on nice days is garden.\ngardening is fun as long as there aren't too m
any bugs!"
```

Reusing regular expressions

If we want to use a regular expression in multiple places in our program, we can `compile()` the pattern and any flags. Then, we can use the pattern's methods to find and replace text.

```
In [29]: program_re = re.compile('program', flags=re.I)
```

```
In [30]: about_py = 'Python is a programming language first released in 1991.'
```

```
In [31]: print(program_re.findall(text))
         print(program_re.findall(about_py))
```

```
['program', 'Program']
['program']
```

Working with Files

So far we have worked with data bundled with our notebooks. Most data, of course, is stored elsewhere: in databases and files stored locally and online. We need to be able to read data into a program to do analyses. We should also be able to write data out to files of our own.

Reading and writing files

Luckily, Python has a built-in function, `open()` for opening files. `open()` takes a string indicating the *file path*, or location, of the file to open, plus a one-character string indicating whether we should open the file in read-only, overwrite, or append mode.

`open()` mode	Description
`'r'`	Read-only. Produces an error if the file does not already exist.
`'w'`	Write. Creates a new file if one does not exist. If the file already exists, the current contents are deleted and overwritten.
`'a'`	Append. Adds to an existing file. If the file does not exist, it will be created.

Paths can be *absolute*, like `C:/Users/Owner/Documents/data/demo_data.csv`, or *relative*, like `data/demo_data.csv`. Relative paths are relative to the folder the Python file is in -- if our code is not in `Documents`, `data/demo_data.csv` won't work.

Opening files

To open a file, we use a `with` statement, which follows the pattern `with open('file_path') as file_variable_name:`, then an indented block of code to process the file. The `with` statement ensures that Python closes the file when we're done working with it.

```
In [32]: with open('sample_data/california_housing_test.csv', 'r') as f:
             print(f)
```

```
<_io.TextIOWrapper name='sample_data/california_housing_test.csv' mode='r' encoding='cp1252'>
```

Reading files

Opening a file doesn't immediately get us the file's contents. To do that, we must use a read method.

- `read()` returns the full file contents, which can be overwhelming for larger files.
- `readline()` returns only the next line in the file. Python keeps track of where it is in the file.
- `readlines()` returns the full file as a list. Each item is one line in the file.

In [33]:
```
with open('sample_data/california_housing_test.csv', 'r') as f:
    for i in range(5):
        print(f.readline())
```

```
longitude,latitude,housing_median_age,total_rooms,total_bedrooms,population,households,median_inc
ome,median_house_value

-122.05,37.37,27,3885,661,1537,606,6.6085,344700

-118.3,34.26,43,1510,310,809,277,3.599,176500

-117.81,33.78,27,3589,507,1484,495,5.7934,270500

-118.36,33.82,28,67,15,49,11,6.1359,330000
```

Writing files

There are corresponding `write()` methods for files.

- `write()` writes a string to file.
- `writelines()` writes each item in an iterable to file, with no separating text in between.

In [34]:
```
provinces = ['BC', 'AB', 'SK', 'MB', 'ON', 'QC', 'NL', 'NB', 'NS', 'PE']
with open('provinces.txt', 'w') as province_file:
    province_file.writelines(provinces)
```

In [35]:
```
with open('provinces.txt', 'r') as province_file:
    print(province_file.read())
```

```
BCABSKMBONQCNLNBNSPE
```

Working with specific file formats

Python has built-in modules for working with specific file formats, like `csv` and `json`. We won't spend much time here, as we will soon be working with libraries that let us open, analyze, and write data in both formats.

In [36]:
```
import csv
```

In [ ]:
```
with open('sample_data/california_housing_test.csv', 'r') as f:
        contents = csv.reader(f)
    for row in contents:
        print(row)
```

Navigating folders

Being able to navigate the computer's file system enables us to work with entire folders' worth of files stored locally on a computer (or "locally" in an environment like Colab). Python's built-in `os` module lets us do just that.

In [38]:
```
import os
```

In [39]:
```
# get the path to the folder we're currently in
os.getcwd()
```

Out[39]:

```
'C:\\Users\\unive\\Documents\\GitHub\\dsi-python-workshop\\01-slides'
```

In [40]:
```
# see the contents of the current folder
os.listdir()
```

Out[40]:

```
['.ipynb_checkpoints',
 '00_hello_python.ipynb',
 '01_getting_started_fundamentals.ipynb',
 '02_reality_control_flow_iterables.ipynb',
 '03_in_out_modules_files_oop.ipynb',
 '04a_data_numpy.ipynb',
 '04b_data_pandas.ipynb',
 'provinces.txt',
 'sample_data']
```

In [41]:
```
# loop over the files in the sample data folder
for i in os.listdir('sample_data'):
    print(i)
```

```
anscombe.json
california_housing_test.csv
```

Manipulating paths

The `os.path` submodule provides safe ways to manipulate paths. `os.path.join()` lets us create properly formatted paths from separate folder and file names, without worrying about getting slashes right. `os.path.exists()` lets us check for a file before trying to open or accidentally overwriting it.

In [42]:
```
# create full paths for sample data files

cwd = os.getcwd()
full_paths = []
for i in os.listdir('sample_data'):
    full_paths.append(os.path.join(cwd,
                                   'sample_data',
                                   i))
full_paths
```

Out[42]:

```
['C:\\Users\\unive\\Documents\\GitHub\\dsi-python-workshop\\01-slides\\sample_data\\anscombe.json
',
 'C:\\Users\\unive\\Documents\\GitHub\\dsi-python-workshop\\01-slides\\sample_data\\california_ho
using_test.csv']
```

In [43]:
```
# check if a file exists to avoid overwriting it
        text = 'Lavender is a small purple flower.'

if os.path.exists('plants.txt'):
    print('plants.txt already exists')
else:
    with open('plants.txt', 'w') as f:
        f.write(text)
```

Object-Oriented Programming

(a very brief introduction)

What is object-oriented programming (OOP)?

Object-oriented programming is an approach to writing programs that organizes code into *objects* with data, or *attributes*, and *methods* (which we've seen before!). The Python types we have seen are all objects. The data science packages we'll encounter adopt this approach as well.

In Python, objects are modeled with *classes*. We can think of a class as a template that defines an object's attributes and methods.

For example, we can model basketball players as a class. To keep it simple, let's say each player has a position, team, and jersey number. Players can dribble, shoot, or pass the ball to another player.
In this case, the class *BasketballPlayer* would have the attributes *position*, *team*, and _jersey*number*, and the methods *dribble()*, *shoot()*, and _pass*ball(player)*. To create a new BasketballPlayer, we would have to somehow specify their position, team, and jersey number.

We can model abstract concepts as classes, too. A *Dataset* class might have attributes like values and size. Methods could include calculating summary statistics, like finding the mean or counting unique values. To create a new Dataset, we would have to specify the values, while the size could be calculated for us.
A *LinearModel* class might have attributes like coefficients and variables, and methods to fit the model to data or make predictions.

Creating a class

Let's turn *Dataset* into an actual class. Similar to how we use `def` to start a function definition, we use `class` to define a new class. Class names, by convention, start with a capital letter.

```
class Dataset:
    '''Data values and methods to summarize them''''
```

Then, we define the class's `__init__()` method, which is a special method that is called whenever we make a new instance of our class. It *initializes* the object in memory. The first parameter of every method is `self`, a reference to the object. `__init()__` should also take data to use to make an object -- in this case, values to store.

```
class Dataset:
    '''Data values and methods to summarize them'''

    def __init__(self, values):
        '''Create a new dataset with values as a list'''
```

We use `__init__()` to attach data, or attributes, to the individual object. This looks like variable assignments we've seen so far, except that every variable name starts with `self.`. Notice that attributes can be derived.

```
class Dataset:
    '''Data values and methods to summarize them'''

    def __init__(self, values):
        '''Create a new dataset with values as a list'''
        self.values = values
        self.size = len(values)
```

We can then define additional methods. Each method takes `self` as its first parameter. Attributes should always be referenced with the `self.` prefix.

```
def unique(self):
        '''Return a list of unique values'''
        return list(set(self.values))
```

A bare-bones Dataset class might look like this.

In [44]:
```
class Dataset:
    '''Data values and methods to summarize them'''

    def __init__(self, values):
        '''Create a new dataset with values as a list'''
        self.values = values
        self.size = len(values)

    def unique(self):
        '''Return a list of unique values'''
        return list(set(self.values))

    def mean(self):
        '''Return the mean of all numeric values'''
        numbers = []

        for i in self.values:
            if type(i) is int or type(i) is float:
                numbers.append(i)
        return sum(numbers)/len(numbers)
```

To create a Dataset instance, we do not call `__init__()` directly. Instead, we call `Dataset()`, passing in `values`. Once we have a Dataset instance, we can call Dataset methods just as we did built-in type methods.

In [45]:
```
grades = Dataset([90, 87, 70, 70, 87, 'A'])

print(grades.unique())
print(grades.size)
print(grades.mean())
```

```
['A', 90, 70, 87]
6
80.8
```

Why the double underscores?

`__init__()` is what Python calls a magic or dunder (for double underscore) method. These methods customize what Python does when other functions are called, like how `__init__()` defined `Dataset()`.
Other common magic methods we encounter in classes are `__repr__()`, which tells Python what should appear when we `print()` an object, and `__str__()`, which Python uses to convert an object to a string.

Why learn this?

It's possible to do a lot of analysis without ever designing our own classes. We do, however, encounter objects all the time. The hypothetical *Dataset* and *LinearModel* classes are simplified versions of real classes in packages like `numpy` and `scikit-learn`. Having an intuition for objects makes it easier to understand why code looks the way it does and helps demystify documentation, which often assumes the reader is familiar with OOP concepts.

References

- Bostroem, Bekolay, and Staneva (eds): "Software Carpentry: Programming with Python" Version 2016.06, June 2016, **https://github.com/swcarpentry/python-novice-inflammation**, 10.5281/zenodo.57492.
- Chapter 6, 10 and 14, Gries, Campbell, and Montojo, 2017, *Practical Programming: An Introduction to Computer Science Using Python 3.6*
- "Modules", Python Software Foundation, *Python Language Reference, version 3*. **https://docs.python.org/3/tutorial/modules.html**.
- "datetime", Python Software Foundation, *Python language Reference, version 3*. **https://docs.python.org/3/library/datetime.html**
- "os", Python Software Foundation, *Python language Reference, version 3*. **https://docs.python.org/3/library/os.html**
- "re", Python Software Foundation, *Python Language Reference, version 3*. **https://docs.python.org/3/library/re.html**