# OReilly - 2024-03March-07

March 7, 2024

## 1 Agenda: Day 5 – modules and packages

1. Q&A
2. What are modules? What do they contain? How do we use them?
3. The `import` statement – what it does, and how to use it (different forms)
4. Developing a simple module
5. Python's standard library
6. Modules vs. packages
7. PyPI and `pip`
8. What's next?

## 2 What are modules?

Remember DRY (don't repeat yourself): We want to have every piece of code in a single location.

1. If we have several lines in a row that are roughly the same, we can combine them into a loop.
2. If we have the same code repeated across several parts of our program, we can write that code once as a function, and then invoke the function multiple times.
3. If we have the same code repeated across several different programs, we can use a "library" to store that functionality, and then load it into any programs that might want to make use of it.

Libraries are everywhere in software: - Your OS is a bunch of libraries - Libraries for handling health, finance, statistics, or any other topic you can imagine – why re-invent the wheel?

Python's version of libraries are known as "modules" and "packages." You can think of a module as a single file containing code we want to reuse, and a package as a folder/directory containing modules.

Modules in Python actually play two different roles:

1. They let us put functionality into a module, and then load that functionality (variables, data structures, functions, or data types) wherever we might need them.
2. Modules in Python give us "namespaces," ensuring that if two programmers use the same variable or function name, they won't conflict with one another in a "namespace collision."

You can think of namespaces as last names (surnames), ensuring that we don't confuse variables of the same name with one another.

It's a rare Python program that doesn't use at least one module to do its work.

## 3 How do we use a module?

Using a module in Python is done with the `import` statement:

1. It's a keyword and a statement. It's not a function, so don't use parentheses.
2. In many programming languages, we give a filename (i.e., a string) to the equivalent of `import`, and that file is then loaded. As we'll see, that's not how it works in Python; we give a module name, i.e., a variable name that we want defined, and then Python has to figure out how to translate that into a filename.

When we use `import`: - Python looks for the module on disk, and loads it, creating what we can call a "module object," basically a storage facility for variables and functions. - Python then assigns the variable that we named after the `import` statement to that module object. - If the module was loaded already, then the variable is still assigned, but it isn't loaded a second time.

[2]:
```
# let's load the "random" module, which comes with Python

import random
```

[3]:
```
type(random)    # what kind of value is random referring to?
```

[3]:
```
module
```

[4]:
```
# what names does it contain?
# one easy way to find out is (in Jupyter) to run the "dir" function on it
# this returns a list of strings, the names that the module contains
# (inventory of our warehouse)

dir(random)
```

[4]:
```
['BPF',
 'LOG4',
 'NV_MAGICCONST',
 'RECIP_BPF',
 'Random',
 'SG_MAGICCONST',
 'SystemRandom',
 'TWOPI',
 '_ONE',
 '_Sequence',
 '__all__',
 '__builtins__',
 '__cached__',
 '__doc__',
 '__file__',
 '__loader__',
 '__name__',
 '__package__',
 '__spec__',
```

```
'_accumulate',
'_acos',
'_bisect',
'_ceil',
'_cos',
'_e',
'_exp',
'_fabs',
'_floor',
'_index',
'_inst',
'_isfinite',
'_lgamma',
'_log',
'_log2',
'_os',
'_pi',
'_random',
'_repeat',
'_sha512',
'_sin',
'_sqrt',
'_test',
'_test_generator',
'_urandom',
'_warn',
'betavariate',
'binomialvariate',
'choice',
'choices',
'expovariate',
'gammavariate',
'gauss',
'getrandbits',
'getstate',
'lognormvariate',
'normalvariate',
'paretovariate',
'randbytes',
'randint',
'random',
'randrange',
'sample',
'seed',
'setstate',
'shuffle',
'triangular',
```

```
 'uniform',
 'vonmisesvariate',
 'weibullvariate']
```

Names without a `_` at the start and/or end are fair game for us to use in a module. These are all "attributes" of the module, meaning that they're names which come after a `.` and the module's name.

[5]:
```
# randint is a function defined in the random module
# we can invoke it by calling random.randint, passing two arguments (min and␣
 ↪max values)

random.randint(0, 100)
```

[5]:
```
85
```

## 4 Underscores in Python names

In variables, functions, and also in modules, you'll often see names that start and/or end with `_`. These (can) have special meanings:

- If a name *starts* with a single `_`, that means it's supposed to be private to the module. The author is trying to tell you that there are no promises regarding this working, not changing, etc. in the future. Try to avoid using these names, because there are not guarantees.
- If a name *ends* with a single `_`, that's usually in scientific/AI applications and means that it was the results of running a model.
- If a name *both starts and ends* with double underscores (`__`), these are known in Python as "dunders." These names have special meaning for Python. If one is defined, and Python is looking for it, then you can really influence the way that the language works. Setting a dunder name without knowing what it does is asking for trouble.

## 5 Exercise: Number guessing

1. Load the `random` module into memory. We are (again) going to use `random.randint` in our program.
2. Generate a random number using that module. Assign the result to `number`.
3. Ask the user, repeatedly, to guess the random number:
   - If they get it, congratulate them and exit
   - If they are too low, tell them that
   - If they are too high, tell them that
4. Until the user guesses the number, the loop will continue.

[7]:
```
# import random

number = random.randint(0, 100)   # get a random number

while True:
    s = input('Enter guess: ').strip()
```

```
    if not s.isdigit():
        print(f'{s} is not numeric; try again')
        continue

    n = int(s)

    if n == number:
        print('You got it!')
        break
    elif n < number:
        print('Too low!')
    else:
        print('Too high!')
```

```
Enter guess:  50
Too low!
Enter guess:  75
Too low!
Enter guess:  90
Too high!
Enter guess:  85
Too high!
Enter guess:  80
Too high!
Enter guess:  77
Too low!
Enter guess:  78
You got it!
```

[ ]:
```
# AB

import random
rando_num = random.randint(0,100)

while True:
    number = int(input("what is the random number?"))
    if rando_num > number:
        print("You are too low!")
    elif rando_num < number:
        print("You are too high!")
```

```
    else:
        print("You have found the number!")
        break
```

The best way to check if the user gave us a legit integer as input is to let the function fail, and trap that failure, known as "exception handling."

Next best, which is what I showed in class, is to check the string to be sure it contains only digits. Python offers three string methods that are almost identical:

- `str.isdigit()`
- `str.isnumeric()`
- `str.isdecimal()`

For most cases, these will give identical results. I tend to use `isdigit`, just because it's shorter.

In Jupyter, you can stop the `input` process by going to the Kernel menu and choosing "interrupt." If that doesn't work, then you can choose "restart kernel," which erases all defined variables/functions since Jupyter started, but is more likely to succeed.

What if I am going to be calling `random.randint` many times? Can I just call `randint`?

[8]:
```
randint(0, 100)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[8], line 1
----> 1 randint(0, 100)

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

We can see that when we used `import random`, we defined the variable `random`. It, as as module, then gives us access to all of its attributes. So we can call `random.randint`. But `randint` by itself doesn't exist.

However, we can define it!

We can use a different syntax for `import`:

`from MODULE import name`

What this does is: - Imports the module, just like before, except that it doesn't define the variable - It does define the specific variable you asked about

So if you were to say

`from random import randint`

The `random` module would still be imported from disk in its entirety, and would be stored in Python's memory. However, there would be no `random` variable referring to the module. Rather, there would be the `randint` variable referring to what `random.randint` had previously referred to.

[1]:
```
random.randint(0, 100)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[1], line 1
----> 1 random.randint(0, 100)

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

[2]:
```
from random import randint   # this means: I want to define randint as a variable
```

[3]:
```
random
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[3], line 1
----> 1 random

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

[4]:
```
randint(0, 100)
```

[4]:
```
86
```

If you say `from random import randint`, then you don't have access to the other names in the `random` module. You can import them one by one:

```
from random import randint, choice, choose
```

Now those three names will be defined as variables, rather than as attributes under `random`.

[5]:
```
import random as r    # this gives the module an alias; it won't be known as "random"
```

[6]:
```
r.randint(0, 100)
```

[6]:
```
58
```

[7]:
```
r
```

[7]:
```
<module 'random' from
'/Users/reuven/.pyenv/versions/3.12.1/lib/python3.12/random.py'>
```

You might want to use `import .. as` in cases where the module is deep in a hierarchy of packages. Also, there are many modules that expect you to import them with an alias, out of convention.

For example, the `numpy` module assumes that you'll say `import numpy as np`

Finally, you can say `from MODULE import NAME as ALIAS`. This way, you get one imported name, with the alias you choose.

## 6 Ways to `import`

- `import MODNAME`
- `from MODNAME import NAME`
- `import MODNAME as ALIAS`
- `from MODNAME import NAME as ALIAS`

All of these work, and all are common in Python programs.

There is a fifth version of this:

- `from MODNAME import *`

What that means is: Load the module and everything defined in the module should then be defined as a variable in you program.

If `MODNAME` were to have 20 attributes defined in the module, you would now have 20 new global variables by those names.

**PLEASE DO NOT USE THIS SYNTAX!** It will just cause trouble and pain at some point down the road.

## 7 Where are modules loaded from?

We saw two things:

- When we use `import`, we don't specify a filename to load
- When we asked to see the printed representation of `random`, it told us from which file it was loaded

How does Python know how to find the module file based on the name I gave?

Answer: It takes the module name we gave, and adds `.py` to it to get a filename. It then looks for this filename in every directory in the special list `sys.path`. Python runs a `for` loop on `sys.path`, looking for our filename. The first match that it finds is the winner.

That means that if you have a file in the current directory whose name is the same as a module in a later directory, you'll never see the file in the later directory.

[9]:
```
r
```

[9]:
```
<module 'random' from
'/Users/reuven/.pyenv/versions/3.12.1/lib/python3.12/random.py'>
```

[10]:
```
import sys
```

[11]:
```
sys.path
```

[11]:
```
['/Users/reuven/Courses/Current/OReilly-2024-02February-python',
 '/Users/reuven/.pyenv/versions/3.12.1/lib/python312.zip',
 '/Users/reuven/.pyenv/versions/3.12.1/lib/python3.12',
 '/Users/reuven/.pyenv/versions/3.12.1/lib/python3.12/lib-dynload',
 '',
 '/Users/reuven/.pyenv/versions/3.12.1/lib/python3.12/site-packages']
```

## 8 Next up

- Develop a module
- Special names ("dunders") that are defined in modules
- Add functionality to our module
- Reloading them, as well

[12]:
```
# import some_name   # Python will look for some_name.py in all the sys.path␣
↪locations

# if I create a file called mymod.py in the current directory,
# then Python will load the module, and give me access to its contents.
```

[13]:
```
import mymod
```

[14]:
```
mymod
```

[14]:
```
<module 'mymod' from '/Users/reuven/Courses/Current/OReilly-2024-02February-
python/mymod.py'>
```

[15]:
```
# we loaded my empty module.
# what attributes will we see when we run "dir" on it?

dir(mymod)
```

[15]:
```
['__builtins__',
 '__cached__',
 '__doc__',
 '__file__',
 '__loader__',
 '__name__',
 '__package__',
 '__spec__']
```

## 9 Some of the dunders on our module

When we `import` a module, even an empty module, Python defines a bunch of dunders (names with double underscore before/after them):

- `__builtins__` – this is an alias to the "builtins" namespace, where things like `len` and `str` and `dict` are all defined.

- `__file__` – the filename that was loaded
- `__name__` – this is the name of the module itself. For `random`, it's the string `'random'`.

[16]:
```
mymod.__file__
```

[16]:
```
'/Users/reuven/Courses/Current/OReilly-2024-02February-python/mymod.py'
```

[18]:
```
mymod.__name__
```

[18]:
```
'mymod'
```

## 10 Loading and reloading

Normally, a Python program runs starting in an empty environment, without anything defined. This means that when you say `import`, the module is imported, and its contents are available until the program exits.

In Jupyter, we don't start Python many times. Rather, Python is always running. If you `import` once, you'll get the module loaded, as per usual. But if you modify the module and then `import` a second time, you'll get the old version, because `import` only works once per file in a Python session.

I have configured my Jupyter to automatically reload modules. But that is not the norm! Usually, you need to tell Python that you want to reload a module; otherwise, it won't do that.

We will use the `reload` function in the `importlib` module to reload things:

[20]:
```
import importlib            # provides import-related functionality
importlib.reload(mymod)     # reloads a module that was already loaded before
```

[20]:
```
<module 'mymod' from '/Users/reuven/Courses/Current/OReilly-2024-02February-
python/mymod.py'>
```

[21]:
```
# let's look at what our module includes now!

dir(mymod)
```

[21]:
```
['__builtins__',
 '__cached__',
 '__file__',
 '__loader__',
 '__name__',
 '__package__',
 '__spec__',
 'hello',
 'x',
 'y',
 'z']
```

[22]:
```
mymod.x
```

[22]: 100

[23]:
```
mymod.y
```

[23]: [10, 20, 30]

[24]:
```
mymod.z['a']
```

[24]: 10

## 11 Notice how variables (in the module) become attributes (out of the module)

In the module file, we defined `x`, `y`, `z`, and `hello` as global variables.

When we used `import` on `mymod`, we got all of those – but not as variables. Rather, we got them as attributes on the module object.

## 12 What are attributes?

In Python, we have two storage systems for values:

- Variables, which are names (`x`, `name`, and `s`)
- Attributes, which are names that belong to someone else. These all come after a `.`. So if we see `a.b`, then we say that `a` is a variable, but `b` is an attribute of `a`.

When we `import` a module, the module object gets a bunch of attributes, meaning names that we can access via the module, using `.`.

When we said `import mymod`, we now not only had the `mymod` variable (referring to the module object), but we also had four attributes – `mymod.x`, `mymod.y`, `mymod.z`, and `mymod.hello`. Just as variable names can be for data or functions, attributes can also be data or functions.

## 13 Exercise: `menu` module, first stage

We are going to write a module that lets programs present the user with a bunch of options. The user will need to choose one of those options. We'll then find out which option they chose.

This will all be in the `menu` module (i.e., `menu.py`). We'll start with a simpler function:

1. Create a file, `menu.py`, in the same directory as Jupyter.
2. Inside of `menu.py`, write a function, `get_input`. When this function is invoked, it asks the user to enter some input. It returns the user's input to the caller, but removing any whitespace on the start and end. In other words, it'll run `str.strip` on the user's input.

I should be able to, when this module is working/done, say:

```
import menu
s = menu.get_input()
print(f'You typed {s}')
```

[27]:
```
import menu
s = menu.get_input()
print(f'You typed {s}')
```

```
Enter something:          something

You typed something
```

## 14 Exercise: Get the user's choice

Now define a new function in our module, `get_choice`.

1. `get_choice` takes a list of strings as arguments.
2. It asks the user to enter one of those strings.
3. If the user enters one of those strings, it is returned to the caller.
4. If not, then the function scold the user and asks again.

So `get_choice` will be an infinite loop, which only exits when the user chooses one of the options that was passed as an argument to the `get_choice` function.

Example:

```
user_choice = get_choice(['a', 'b', 'c'])

Choose from a/b/c: d
d is not a legit option; try again
Choose from a/b/c: b
[function returns b]
```

[29]:
```
import menu
user_choice = menu.get_choice(['a', 'b', 'c'])
```

```
Choose from ['a', 'b', 'c']:  d

d is invalid; try again

Choose from ['a', 'b', 'c']:  b
```

[30]:
```
print(f'User chose {user_choice}')
```

```
User chose b
```

## 15 What happens when we `import`?

It turns out that when we `import` a module, we are actually causing the module file to be executed, from start to finish.

Normally, our module contains `def` (to define functions) and assignments to variables (with `=`).

Think about it: After I `import` a module, I want to use the functions in the module. They are only available if the `def` already executed.

[32]:
```
importlib.reload(mymod)
```

```
Hello from mymod!
Goodbye from mymod!
```

[32]:
```
<module 'mymod' from '/Users/reuven/Courses/Current/OReilly-2024-02February-
python/mymod.py'>
```

## 16 Next up

- `__name__` and its magical qualities
- Python's standard library
- Third-party modules, PyPI, and `pip`

[33]:
```
dir(mymod)
```

[33]:
```
['__builtins__',
 '__cached__',
 '__file__',
 '__loader__',
 '__name__',
 '__package__',
 '__spec__',
 'hello',
 'x',
 'y',
 'z']
```

[35]:
```
importlib.reload(mymod)
```

```
Hello from mymod!
Goodbye from mymod!
```

[35]:
```
<module 'mymod' from '/Users/reuven/Courses/Current/OReilly-2024-02February-
python/mymod.py'>
```

## 17 `__name__` ("dunder name")

The variable `__name__` always exists and is defined in Python.

If we have it in a module file, then it can have one of two values:

- If we have imported the module, the `__name__` is a string, the name of the module. So in `mymod.py`, its value would be the string `'mymod'`.
- If the module is run as a standalone program, then `__name__` is the string `'__main__'`. In fact, any Python program that we run has `__name__` equal to `'__main__'`. Any value that isn't this means that we have imported that module for use of its definitions.

The reason I'm telling you this is that there's a very famous line that is in nearly every Python module:

```
if __name__ == '__main__':
    # STUFF HERE
```

This is usually at the end of the module. It means: The below code will only run when you execute this file. It will *not* run when you `import` this file.

Many, *many* modules use this, but in different ways: - Some will provide an interactive interface to the functions that were defined in the module - Some will run automated tests on themselves - Some will demo the functionality they offer

[37]:
```
importlib.reload(menu)
```

[37]:
```
<module 'menu' from '/Users/reuven/Courses/Current/OReilly-2024-02February-
python/menu.py'>
```

## 18 Python standard library

When you install Python, you get a large number of modules with it. These are collectively known as the "standard library."

If you use a module from the standard library in your program, you stil need to `import` it. But if you give your program to a friend, and they're also running Python, then they don't need to install anything special. The standard library is available to anyone with Python installed. Given a particular version of Python (e.g., 3.12), all users, on all platforms, have access to the same standard library.

Let's look at the list of modules in the standard library!

`https://docs.python.org/3/library/index.html`

[38]:
```
import glob   # this module lets us get a list of files matching a pattern
```

[39]:
```
# module is glob
# function is also glob!
glob.glob('*.txt')
```

[39]:
```
['mini-access-log.txt',
 'nums.txt',
 'config.txt',
 'shoe-data.txt',
 'linux-etc-passwd.txt',
 'wcfile.txt',
 'myfile.txt']
```

[41]:
```
import os

os.stat('nums.txt')  # this returns info about that file, including the size
```

[41]:
```
os.stat_result(st_mode=33188, st_ino=213565303, st_dev=16777221, st_nlink=1,
st_uid=501, st_gid=20, st_size=42, st_atime=1709798364, st_mtime=1505818936,
```

```
st_ctime=1709798281)
```

[42]:
```
os.stat('nums.txt').st_size   # this returns the size of the file we named
```

[42]:
```
42
```

## 19 Exercise: File sizes

1. Ask the user to enter a globbing pattern, such as `*.txt`.
2. Iterate over the filenames that you got, and run `os.stat` on each file.
3. Then retrieve the `st_size` attribute from the result of `os.stat`.
4. Print the filename and its size on the screen.
5.

[43]:
```
import glob
import os

pattern = input('Enter a pattern: ').strip()

for one_filename in glob.glob(pattern):
    stat_results = os.stat(one_filename)
    print(f'{one_filename}: {stat_results.st_size}')
```

```
Enter a pattern:  *.txt
```

```
mini-access-log.txt: 36562
nums.txt: 42
config.txt: 15
shoe-data.txt: 1676
linux-etc-passwd.txt: 2683
wcfile.txt: 165
myfile.txt: 27
```

[ ]:
```
# AB

import glob
import os
patt = input("Enter a globbing pattern: ")
txt_list = glob.glob(patt)
for x in txt_list:
    print(f"{x}: {os.stat(x)} > Size: {os.stat(x).st_size}")
```

[44]:
```
glob.glob('*x*')
```

[44]:
```
['mini-access-log.txt',
 'nums.txt',
 'config.txt',
 'shoe-data.txt',
```

```
 'linux-etc-passwd.txt',
 'wcfile.txt',
 'myfile.txt']
```

[45]:
```
glob.glob('*j*')
```

[45]:
```
[]
```

[47]:
```
glob.glob('c*')
```

[47]:
```
['config.txt']
```

## 20 What if a module isn't in the standard library?

The standard library is huge – but there many, *many* modules that we might want to install that aren't part of the standard library. Where are they? How can I download and install them?

They are all on PyPI, the Python Package Index, at https://PyPI.org

## 21 Next up

1. Practice installing with `pip` and using something from PyPI
2. What next? Where do we go from here (with Python)?

## 22 To install from PyPI

1. Find the package on PyPI
2. On the command line (i.e., not in Jupyter!) run the `pip` command, as `pip install PACKAGE`.
    - If you already have the package installed, and there's a newer version on PyPI, the newer one will *not* be installed unless you add the `-U` (or `--update`) option: `pip install -U PACKAGE`.
    - The package will be installed (typically) in your `site-packages` directory, which is in `sys.path`. This means that right after installing the package, you can use `import` inside of Python and start to use it.

[48]:
```
# The "Rich" package allows us to display text in all sorts of amazing ways.

import rich
```

[49]:
```
rich.print('Hello out there!')
```

```
Hello out there!
```

[55]:
```
# if we want, we can colorize the text by using [COLOR] and [/COLOR] around
# areas we want colorized

```

```
rich.print('Hello [italic][bold][blue on yellow]out[/blue on yellow][/bold][/
↪italic] there!')
```

```
Hello out there!
```

## 23 Exercise: Colorizing text

1. Download and install `Rich` using `pip`. (Notice that it's a capital "R" when installing.)
2. `import rich`
3. Ask the user to enter a sentence.
4. If the sentence has an even number of characters, print it in green.
5. If the sentence has an odd number characters, print it in blue.

[58]:
```
import rich

s = input('Enter a sentence: ').strip()

if len(s) % 2 == 0:
    color = 'green'
else:
    color = 'blue'

rich.print(f'[{color}]{s}[/{color}]')
```

```
Enter a sentence:  hello!
```

```
hello!
```

## 24 Updating Python

A new version of Python comes out (nowadays) every October. The Python developers try really hard not to break existing functionality, so that moving to a new version of Python will be as smooth as possible.

Part of their way of making upgrades easy is announcing new features long in advance. This allows people to try them, give feedback, and get ready for them.

Sometimes, if it's a really big thing that they're adding or changing, they'll let you use it early. The mechanism for doing this is a special module in the standard library called `__future__`.

You can then say

```
from __future__ import print_function
```

BTW, `print_function` was there for people transitioning from Python 2 to Python 3. In the former, `print` wasn't a function, but nowadays, in Python 3, it is.

You gain nothing from using this particular `import`.

[59]:
```
import __future__
```

[61]:
```
dir(__future__)
```

[61]:
```
['CO_FUTURE_ABSOLUTE_IMPORT',
 'CO_FUTURE_ANNOTATIONS',
 'CO_FUTURE_BARRY_AS_BDFL',
 'CO_FUTURE_DIVISION',
 'CO_FUTURE_GENERATOR_STOP',
 'CO_FUTURE_PRINT_FUNCTION',
 'CO_FUTURE_UNICODE_LITERALS',
 'CO_FUTURE_WITH_STATEMENT',
 'CO_GENERATOR_ALLOWED',
 'CO_NESTED',
 '_Feature',
 '__all__',
 '__builtins__',
 '__cached__',
 '__doc__',
 '__file__',
 '__loader__',
 '__name__',
 '__package__',
 '__spec__',
 'absolute_import',
 'all_feature_names',
 'annotations',
 'barry_as_FLUFL',
 'division',
 'generator_stop',
 'generators',
 'nested_scopes',
 'print_function',
 'unicode_literals',
 'with_statement']
```

[63]:
```
print(__future__.all_feature_names)
```

```
['nested_scopes', 'generators', 'division', 'absolute_import', 'with_statement',
'print_function', 'unicode_literals', 'barry_as_FLUFL', 'generator_stop',
'annotations']
```

[64]:
```
help(__future__)
```

```
Help on module __future__:

NAME
    __future__ - Record of phased-in incompatible language changes.
```

```
MODULE REFERENCE
    https://docs.python.org/3.12/library/__future__.html

    The following documentation is automatically generated from the Python
    source files.  It may be incomplete, incorrect or include features that
    are considered implementation detail and may vary between Python
    implementations.  When in doubt, consult the module reference at the
    location listed above.

DESCRIPTION
    Each line is of the form:

        FeatureName = "_Feature(" OptionalRelease "," MandatoryRelease ","
                                  CompilerFlag ")"

    where, normally, OptionalRelease < MandatoryRelease, and both are 5-tuples
    of the same form as sys.version_info:

        (PY_MAJOR_VERSION, # the 2 in 2.1.0a3; an int
         PY_MINOR_VERSION, # the 1; an int
         PY_MICRO_VERSION, # the 0; an int
         PY_RELEASE_LEVEL, # "alpha", "beta", "candidate" or "final"; string
         PY_RELEASE_SERIAL # the 3; an int
        )

    OptionalRelease records the first release in which

        from __future__ import FeatureName

    was accepted.

    In the case of MandatoryReleases that have not yet occurred,
    MandatoryRelease predicts the release in which the feature will become part
    of the language.

    Else MandatoryRelease records when the feature became part of the language;
    in releases at or after that, modules no longer need

        from __future__ import FeatureName

    to use the feature in question, but may continue to use such imports.

    MandatoryRelease may also be None, meaning that a planned feature got
    dropped or that the release version is undetermined.

    Instances of class _Feature have two corresponding methods,
    .getOptionalRelease() and .getMandatoryRelease().
```

```
    CompilerFlag is the (bitfield) flag that should be passed in the fourth
    argument to the builtin function compile() to enable the feature in
    dynamically compiled code.  This flag is stored in the .compiler_flag
    attribute on _Future instances.  These values must match the appropriate
    #defines of CO_xxx flags in Include/cpython/compile.h.

    No feature line is ever to be deleted from this file.

DATA
    __all__ = ['all_feature_names', 'nested_scopes', 'generators', 'divisi…
    absolute_import = _Feature((2, 5, 0, 'alpha', 1), (3, 0, 0, 'alpha', 0…
    all_feature_names = ['nested_scopes', 'generators', 'division', 'absol…
    annotations = _Feature((3, 7, 0, 'beta', 1), None, 16777216)
    barry_as_FLUFL = _Feature((3, 1, 0, 'alpha', 2), (4, 0, 0, 'alpha', 0)…
    division = _Feature((2, 2, 0, 'alpha', 2), (3, 0, 0, 'alpha', 0), 1310…
    generator_stop = _Feature((3, 5, 0, 'beta', 1), (3, 7, 0, 'alpha', 0),…
    generators = _Feature((2, 2, 0, 'alpha', 1), (2, 3, 0, 'final', 0), 0)
    nested_scopes = _Feature((2, 1, 0, 'beta', 1), (2, 2, 0, 'alpha', 0), …
    print_function = _Feature((2, 6, 0, 'alpha', 2), (3, 0, 0, 'alpha', 0)…
    unicode_literals = _Feature((2, 6, 0, 'alpha', 2), (3, 0, 0, 'alpha', …
    with_statement = _Feature((2, 5, 0, 'alpha', 1), (2, 6, 0, 'alpha', 0)…

FILE
    /Users/reuven/.pyenv/versions/3.12.1/lib/python3.12/__future__.py
```

# 25 GM

1. Quick rundown of classes.

Advertisement: On May 6-7, I'll be doing a two-part "object-oriented programming bootcamp" in Python. It's meant for people who have finished this class!

The basic idea of objects is: You want to create new data types. Strings, lists, tuples, etc. are great, but what if you want to create a Person type? Or a Company type? Or a Phone type? Objects allow you to create such types, and to create your own methods that work on such types.

Even if such a "class" doesn't add lots of new functionality, the fact that you can think at a higher level, about a Person rather than a dict, is really helpful.

2. What are good next steps? Sites? Books?

- My OO bootcamp (via O'Reilly!)
- My data analysis with Pandas in 5 weeks
- My book, Python Workout
- My new book, Pandas Workout (once you've learned some Pandas)

You want to practice a *lot*. Use Python wherever you can. I have a site,

https://PracticeYourPython.com, where I list the places I know of where you can get exercises and pratice problems.

[ ]:
```
def mysum(numbers):
    total = 0

    for one_number in numbers:
        total *= one_number

    return total

print(mysum([10, 20, 30]))
```

[ ]:
```
# PySnooper -- the poor man's debugger
```