Modules and Programs

Python Programs

Python code organized in

- modules,
- packages, and
- scripts.

We've already used some modules, now we'll learn what they are, how they're orgainized in packages, and how to write Python programs that can be run on their own, not just entered in the Python command shell.

Importing Modules

To `import` a module means to get names from the module into scope, or add them to a namespace. When you import a module, you can access the module's members with the dot operator.

```
>>> import math   # Adds the math module to the current namespace
>>> math.sqrt(64) # Uses the sqrt function from the math module
8.0
```

You can also import a module and give it an alias: `import <module> as <local-name>`

```
>>> import math as m
>>> m.sqrt(64)
8.0
```

Importing into Local Scope

Importing brings names into the scope of the import. Here we import the math module into the scope of a single function:

```
>>> def hypotenuse(a, b):
...     import math
...     return math.sqrt(a*a + b*b)
...
>>> hypotenuse(3, 4)
5.0
```

But it's not available at the top level.

```
>>> math.sqrt(64)
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
NameError: name 'math' is not defined
```

Importing Names from a Module

You can choose to import only certain names from a module:

```
>>> from math import sqrt
>>> sqrt(64)
8.0
>>> floor(1.2)
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
NameError: name 'floor' is not defined
```

Or all names from a module:

```
>>> from math import *
>>> floor(1.2)
1
>>> sin(0)
0.0
>>> sin(.5 * pi)
1.0
```

Using this syntax adds the names from the module to your namespace so that you don't have to use a fully-qualified name, e.g., you can say `sqrt(64)` instead of `math.sqrt(64)`.

Namespace Pollution

It's usually better to import modules and access their members with dot notation. When you `import ...` `from ..` from several modules, especially if you use `*`, you "pollute" your namespace with many names and potentially cause problems.

Active Review

Evaluate the following, in order, in a Python REPL:

- `from logging import *`
- `log(WARN, 'A log message')`
- `from math import *`
- `log(WARN, 'A log message')`

What happened?

Writing Python Modules

A Python module is text file ending in `.py` – this is why you should always name your Python source files with a `.py` ending. A module typically includes classes, functions and variables.

Active Review

Save the following code in a file named `arithmetic.py`:

```
def add(a: int, b: int) -> int:
    return a + b

def sub(a: int, b: int) -> int:
    return a - b

def mul(a: int, b: int) -> int:
    return a * b

def div(a: int, b: int) -> int:
    return a / b
```

- In your Python REPL, evaluate `import arithemtic`.
  - Did you get an error? What caused the error?
- If you got an error when you tried to import your `arithmetic` module, fix it.
- Now use functions from your `arithmetic` module to make sure it works.

Python Scripts

A Python script is any text file containing executable Python code. Our `hello.py` script from Day 1 is an example of a Python script. Note that a module can be a Python script if it contains code that executes whenever the module is run by the Python interpreter.

Active Review

- Run `arithmetic.py` in "script mode" by entering `python3 arithmetic.py` in your OS command shell.
  - What happened?
- Add the following to the bottom of your `arithmetic.py` file:

```
import sys
ops = {'+': add, '-': sub, '*': mult, '/': div}
op = ops[sys.argv[2]]
print(op(int(sys.arg[1]), int(sys.arg[2])))
```

- Run `arithmetic.py` with `python3 arithmetic.py 6 + 2`.
- Restart your Python REPL and import your `arithmetic` module.
  - What happened?

`if __name__ == '__main__'`

To make a module a script that only evaluates definitions when imported and only runs the "script" parts when run by the Python interpreter, include an `if __name__ == '__main__'` block at the bottom. The code in the `if __name__ == '__main__'` block will only execute when the module is run as a script.

Active Review

Add the following to the bottom of your `arithmetic.py` module:

```
if __name__ == '__main__':
    import sys
    ops = {'+': add, '-': sub, '*': mult, '/': div}
    op = ops[sys.argv[2]]
    print(op(int(sys.arg[1]), int(sys.arg[2])))
```

- Run `arithmetic.py` in "script mode" with `python3 arithmetic.py`.
  - What happened?
- Run `arithmetic.py` with `python3 arithmetic.py 6 + 2`.
- Run `arithmetic.py` with `python3 arithmetic.py 6 / 2`.
- Run `arithmetic.py` with `python3 arithmetic.py 6 * 2`.
  - What happened?

Shebang!

Another way to run a Python program (on Unix) is to tell the host operating system how to run it. We do that with a "shebang" line at the beginning of a Python program:

```
#!/usr/bin/env python3
```

This line says "run python3 and pass this file as an argument." So if you have a script in `foo.py` with shebang line as above and which has been set executable (`chmod +x foo.py`), these are equivalent:

```
$ python3 foo.py
$ ./foo.py
```

Command-line Arguments

When you run a Python program, Python collects the arguments to the program in a variable called sys.argv. Given a Python program (`arguments.py`):

```
#!/usr/bin/env python3
import sys

print(sys.argv)

if len(sys.argv) < 2:
    print("You've given me nothing to work with.")
else:
    print(sys.argv[1] +"? Well I disagree!")
```

```
$ ./arguments.py Pickles
Pickles? Well I disagree!
$ ./arguments.py
You've given me nothing to work with.
```

Interactive Programs

The input() function Python reads all the characters typed into the console until the user presses ENTER and returns them as a string:

```
>>> x = input()
abcdefg1234567
>>> x
'abcdefg1234567'
```

We can also supply a prompt for the user:

```
>>> input('Give me a number: ')
Give me a number: 3
'3'
```

And remember, input() returns a string that may need to be converted.

```
>>> 2 * int(input("Give me a number and I'll double it: "))
Give me a number and I'll double it: 3
6
```

Module Search Path

Just as an operating system command shell searches for executable programs by searching the directories listed in the PATH environment variable, Python finds modules by searching directories. The module search path is stored in `sys.path`:

```
>>> import sys
>>> from pprint import pprint
>>> pprint(sys.path)
['',
 '/usr/lib/python38.zip',
 '/usr/lib/python3.8',
 '/usr/lib/python3.8/lib-dynload',
 '/home/chris/.local/lib/python3.8/site-packages',
 '/usr/local/lib/python3.8/dist-packages',
 '/usr/lib/python3/dist-packages']
```

Notice that the current directory, represented by the '' at the beginning of the search path, is part of `sys.path`, which is why you can import modules located in your current directory.

Also, note use of `pprint`.

Conclusion

- Be careful to distinguish between a Python RELP prompt, and an OS command shell prompt.

Arguments