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 to write our own modules, and the relationship between modules and programs.

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`:

```
1  def add(a: int, b: int) -> int:
2      return a + b
3
4  def sub(a: int, b: int) -> int:
5      return a - b
6
7  def mul(a: int, b: int) -> int:
8      return a * b
9
10 def div(a: int, b: int) -> int:
11     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` as a script by entering `python3 arithmetic.py` in your OS command shell.
  - What happened?
- Add the following to the bottom of your `arithmetic.py` file:

```
1 import sys
2 ops = {'+': add, '-': sub, '*': mult, '/': div}
3 op = ops[sys.argv[2]]
4 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

- Replace the free-standing code at the bottom of your `arithmetic.py` file with this (adding 'if **name**=='**main**': above and indenting suite):

```
1     if __name__ == '__main__':
2         import sys
3         ops = {'+': add, '-': sub, '*': mult, '/': div}
4         op = ops[sys.argv[2]]
5         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:

```
1 #!/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:

```
1 $ python3 foo.py
2 $ ./foo.py
```

Notes: - This form of the shebang line (`#!/usr/bin/env...`) also works on Windows. - You can specify a more specific version of Python, e.g., `#!/usr/bin/env python3.10`.

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
```

Conclusion

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

  Typical macOS/Linux/Unix command shell:

```
drcs@horand ~ $
```

  Typical Windows Powershell:

```
PS C:>
```

  Python REPL:

```
>>>
```

  iPython REPL:

```
In [1]:
```

- Follow `if __name__=='__main__'` and `main` function conventions when writing scripts.