Core Python: Getting Started

MODULARITY

Austin Bingham
COFOUNDER - SIXTY NORTH

@austin_bingham

Robert Smallshire
COFOUNDER - SIXTY NORTH

@robsmallshire

1) Modularity is an important property as it gives us the power to make self contained reusable pieces, which can be combined in new ways to solve different problems.

2) As with most programming languages the most fine grained modularization facility is the definition of reasonable functions.

3) **Collections of related functions** are typically grouped into source code files called **modules**

4) Modules can be used from other modules. So long as we take care not to introduce circular dependencies.

Reusable functions

Source code files called modules

Modules can be used from other modules

Importing modules

Programs or scripts

Python execution model

Make programs executable

Starting Code

```
from urllib.request import urlopen
story = urlopen('http://sixty-north.com/c/t.txt')
story_words = []
for line in story:
    line_words = line.decode('utf-8').split()
    for word in line_words:
        story_words.append(word)
story.close()
```

Open Text Editor

1. Python syntax highlighting
2. Four space indentation
3. UTF-8 encoding

We should use a text editor , preferably one with syntax highlighting support for python and configure it to insert four spaces per indent level when you press the tab key.

You should also check that your editor saves the file using the UTF eight encoding, as that's what the Python3 runtime expects by default.

Run Python from the Shell

```
$ cd corepy
$ python words.py
$
```

All Python source files used the .py extension.

We can execute our modules simply by calling Python and passing the module file name.

Run Changes

```
that
some
of
its
noisiest
authorities
insisted
on
its
being
received
for
good
or
for
evil
in
the
superlative
degree
of
comparison
only
$
```

Importing Modules into the REPL

```
that
some
of
its
noisiest
authorities
insisted
on
its
being
received
for
good
or
for
evil
in
the
superlative
degree
of
comparison
only
>>>
```

Our module can also be imported into the REPL.
Start the REPL and import your module by typing ***"import words".***

Note how, when importing we omit the file extension. The code on your module is executed immediately when imported. This may not be the expected behavior.

To give us more control over when our code is executed and to allow it to be reused, we'll ***need to put our code in a function***

Defining Functions

```
>>> def square(x):
...     return x * x
...
>>> square(5)
25
>>> def launch_missiles():
...     print("Missiles launched!")
...
>>> launch_missiles()
Missiles launched!
>>>
```

Functions are defined using the **def keyword** followed by the function name and argument list in parenthesis and a colon to start a new block.

The code inside the function block must be indented.

We use the **return keyword** to return a value from the function

We call functions by providing the actual arguments in parenthesis after the function Name.

Functions aren't required to explicitly return a value though, perhaps they produce side effects.

It's good practice to prefer functions which

Early Return

```
>>> def even_or_odd(n):
...     if n % 2 == 0:
...         print("even")
...         return
...     print("odd")
...
>>> w = even_or_odd(31)
odd
>>> w is None
True
>>>
```

1) We can return early for a function by using the return key word with no parameter.

Both the return statement without a parameter as well as the implicit return at the end of a function, actually causes the function to return **None.**

2) REPL doesn't display none results, so we don't see them.

By capturing the returned object into a named variable. ,we can test for None.

```
def nth_root(radicand, n):
    return radicand ** (1/n)
```

```
>>> nth_root(16, 2)
4.0
>>> nth_root(27, 3)
3.0
>>>
```

```
def nth_root(radicand, n):
    return radicand ** (1/n)


def ordinal_suffix(value):
    s = str(value)
    if s.endswith('11'):
        return 'th'
    elif s.endswith('12'):
        return 'th'
    elif s.endswith('13'):
        return 'th'
    elif s.endswith('1'):
        return 'st'
    elif s.endswith('2'):
        return 'nd'
    elif s.endswith('3'):
        return 'rd'
    return 'th'


def ordinal(value):
    return str(value) + ordinal_suffix(value)


def display_nth_root(radicand, n):
    root = nth_root(radicand, n)
    message = "The " + ordinal(n) + " root of " \
              + str(radicand) + " is " + str(root)
    print(message)
```

- Calculate ordinal suffixes
- ST suffix for 1
- ND suffix for 2
- RD suffix for 3
- Define ordinal()
- Decomposition
- Display function
- implicit returns

```
def display_nth_root(radicand, n):
  root = nth_root(radicand, n)
  message = "The " + ordinal(n) + " root of " + str(radicand) + " is " + str(root)
  print(message)
```

```
>>> display_nth_root(64, 4)
The 4th root of 64 is 2.8284271247461903
>>>
```

Naming Special Functions

__feature__

Hard to prounounce!

dunder

In python, many language features are implemented or controlled using specially named objects and functions. These special names generally have **two leading and two trailing underscores.** This has the benefit of making them visually distinct, fairly easy to remember and unlikely to collide with other names.

To pronounce these names, the term dunder is used. Dunder is a portmanteau of the term double underscore and is used to refer to any name with leading and trailing Double underscores.

Our way of pronouncing special names

A portmanteau of "double underscore"

Instead of "underscore underscore name underscore underscore" we'll say "dunder name"

Defining a Function

```
def fetch_words():
    story = urlopen('http://sixty-north.com/c/t.txt')
    story_words = []
    for line in story:
        line_words = line.decode('utf8').split()
        for word in line_words:
            story_words.append(word)
    story.close()

    for word in story_words:
        print(word)
```

Here we place all our code into a function. When we import the module using "import words" the module imports, but now the words are not fetched until we call the fetch.

To call the function we use "words.fetch_words()".This use of the DOT is called qualifying the function name with the module name.

Alternatively, we could import a specific function using a different form of import statement:
from words import fetch_words.

Having imported the fetch_words function directly into our REPL session, which is itself a module we can invoke fetch words using its unqualified name.

```
of
its
noisiest
authorities
insisted
on
its
being
received
for
good
or
for
evil
in
the
superlative
degree
of
comparison
only
>>>
$ python words.py
$
```

When we try to run our module directly from the operating system,no words are printed. That's because all the module does now is defined a function and then exit. The function is never called.

To make a module from which we can usefully import functions into the REPL and which can be run as a script, we need to learn a new python idiom __name__.

It give us the means to detect whether our module has been run as a script or imported into another module or the REPL.

__name__

Specially named variable allowing us to detect whether a module is run as a script or imported into another module.

Print Name of Module

```
def fetch_words():
  story = urlopen('http://sixty-north.com/c/t.txt')
  story_words = []
  for line in story:
      line_words = line.decode('utf8').split()
      for word in line_words:
          story_words.append(word)
  story.close()

  for word in story_words:
      print(word)

print(__name__)
```

We add print(__name__) to the end of the module file.

```
$ python
Python 3.7.4 (default, Oct 17 2019, 14:41:32)
[Clang 10.0.1 (clang-1001.0.46.4)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import words
words
>>> import words
>>>
$ python words.py
__main__
$
```

When we 1st import the module into REPL, the Dunder name does indeed evaluate to the modules name.

Imp: if you import the module again in the same REPL, the print statement will not be executed. Module code is only executed once on first import.

When we try running the module as a script from the operating system shell with "python words.py" ,the special Dunder name variable is equal to the string "__main__".
Thus python sets the value of __name__ differently, depending on how our module is being used.

Import or Execute

```
from urllib.request import urlopen

def fetch_words():
  story = urlopen('http://sixty-north.com/c/t.txt')
  story_words = []
  for line in story:
      line_words = line.decode('utf8').split()
      for word in line_words:
          story_words.append(word)
  story.close()

  for word in story_words:
      print(word)

if __name__ == '__main__':
    fetch_words()
```

Our module can use the behavior of __name__, to decide how it should behave. We replaced the print statement with an if statement, which tests the value of __name__.

If __name__ is equal to the string "__main__", we execute our function. On the other hand, if __name__ is not equal to "__main__", the module knows it's being imported into another module, not executed, and so only defines the fetch_words function without executing it. We can now safely import our module without unduly executing our function, and we can usefully run our module as a script

```
that
some
of
its
noisiest
authorities
insisted
on
its
being
received
for
good
or
for
evil
in
the
superlative
degree
of
comparison
only
$
```

The Python Execution Model

`def` is a statement.

Top-level functions are defined when a module is imported or run.

When modules are imported or run, all of the top level statements are run. And this is the means by which the function within the module namespace are defined.

Module, Script, or Program

Python module

Convenient import with API

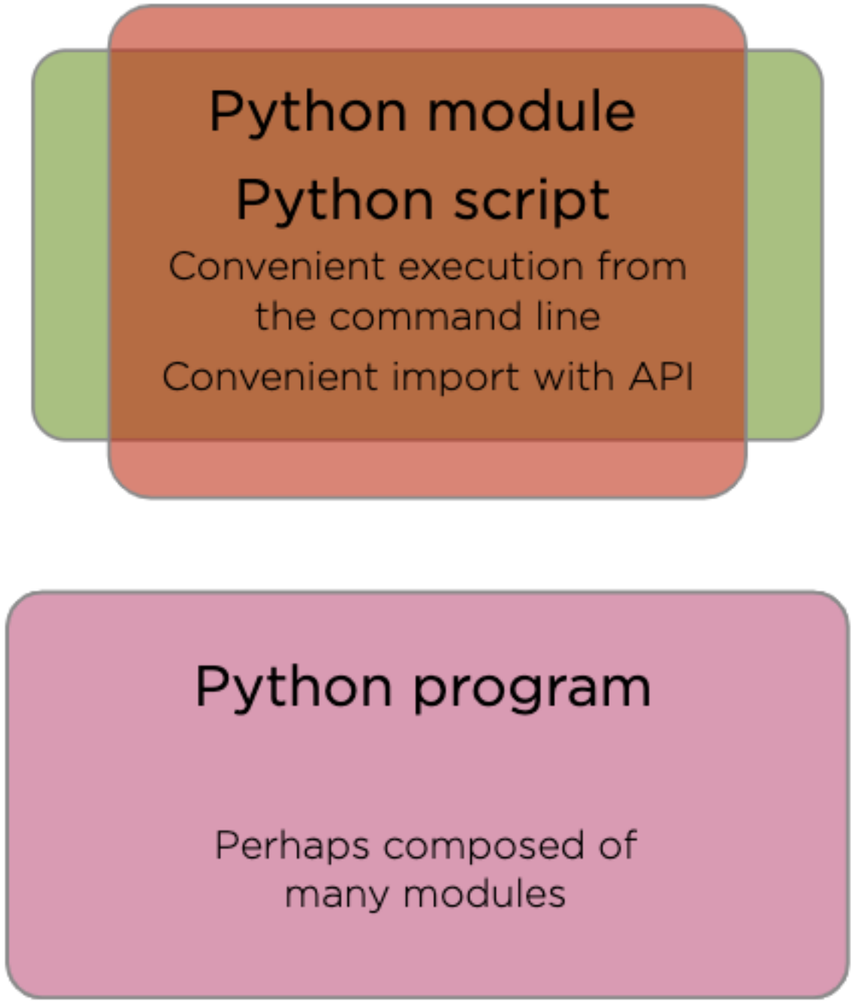
Python script

Convenient execution from the command line

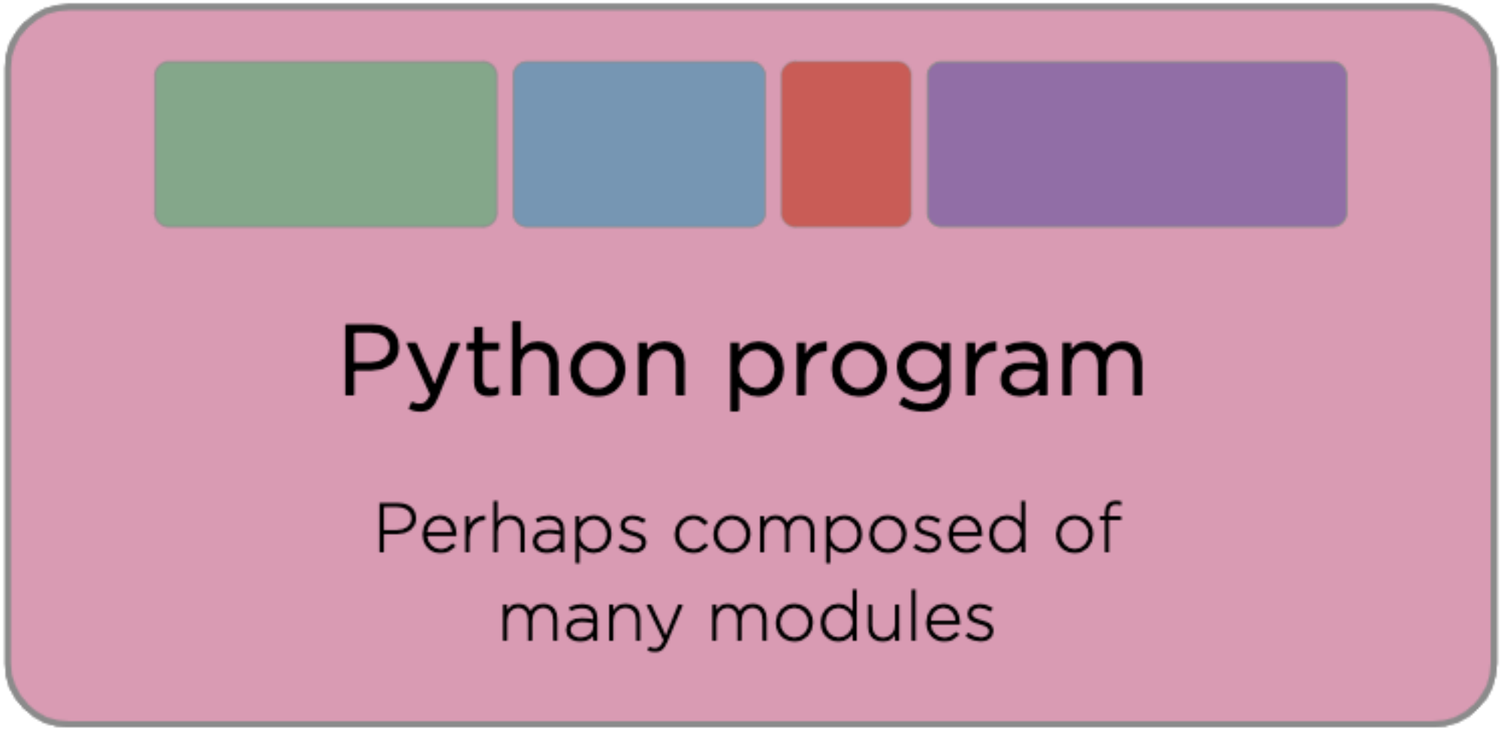
Python program

Perhaps composed of many modules

Module, Script, or Program



Module, Script, or Program



When defining the main block of the module code, we should use the following method:

```
def main():
   ..............
   ..............

if __name__ == '__main__':
     main()
```

This is better than specifying the main functionality under the if block, because it allows us to test the main function from REPL.

Command Line Arguments

Test in the REPL

```
t
r
i
n
g
s

a
r
e

i
t
e
r
a
b
l
e

t
o
o
>>>
```

Command Line and REPL

```
for
good
or
for
evil
in
the
superlative
degree
of
comparison
only
$ python
Python 3.7.4 (default, Oct 17 2019, 14:41:32)
[Clang 10.0.1 (clang-1001.0.46.4)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> from words import *
>>> main()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/Users/sixty-north/corepy/slide_spec/repl-2/words.py", line 22, in main
    url = sys.argv[1]
IndexError: list index out of range
>>>
```

Types of importing:

1) from words import (fetch_words, print_words)
The first new form imports multiple objects from a module using a comma separated list. The parentheses are optional, but they do allow us to break this list over multiple lines if it gets too long. This form is perhaps the most widely used form of the import statement.

2) from words import *
The second new form imports everything from a module using an asterisk while card. This latter form is recommended only for casual use at the REPL. It could wreak havoc in programs, since what is imported is now potentially beyond your control, opening yourself up to potential names space clashes at some future time

Test in the REPL

```
that
some
of
its
noisiest
authorities
insisted
on
its
being
received
for
good
or
for
evil
in
the
superlative
degree
of
comparison
only
>>>
```

Command line arguments:
Access to command line arguments in Python is through the attribute of the sys module called Argv, which is a list of strings to use it.

We must first import this this module at the top of our program.

```
import sys

url = sys.argv[1]   //Zero indexed.
```

Moment of Zen

Sparse is better than dense

Two between functions
That is the number of lines
PEP8 recommends

"Two between functions"- That is the number of lines Pep8 recommends.
According to the Pep8 style guide, it's customary to use to blank lines between module level functions.

We use single blank lines for logical breaks within functions.

Docstrings

docstrings

1) Docstrings are used to add the self documenting capability. This allows us to ask help for python functions, modules.

2) API Documentation in Python uses a facility called docstrings. The docstrings are literal strings, which **occur as the first statement within a named block, such as a function or module.**

3) Normally we use t**riple quoted strings**, even for single line doc strings. This is because they can be easily expanded to add more detail.

Literal strings which document functions, modules, and classes.

They must be the first statement in the blocks for these constructs.

Sphinx

Tool to create HTML documentation from Python docstrings.

One Python convention for Doc Strings is documented in PEP 257 Although it is not widely adopted.

Various tools such a **sphinx** are available to build HTML document from Python Doc Strings, and each tool mandates its preferred docstring format.

Our preference is to use the form presented in Google's Python style guide, since it is amenable to being machine parsed, while still remaining readable at the console.

Docstrings

```
>>> from words import *
>>> help(fetch_words)
Help on function fetch_words in module words:

fetch_words(url)
    Fetch a list of words from a URL.

    Args:
        url: The URL of a UTF-8 text document.

    Returns:
        A list of strings containing the words from
        the document.
(END)
```

Docstrings

```
Help on module words:

NAME
    words - Retrieve and print words from a URL.

DESCRIPTION
    Usage:

        python3 words.py <URL>

FUNCTIONS
    fetch_words(url)
        Fetch a list of words from a URL.

        Args:
            url: The URL of a UTF-8 text document.

        Returns:
            A list of strings containing the words from
            the document.

    main(url)
        Print each word from a text document from at a URL.
:
```

Note that docstrings can be added for modules and functions.

Module docstrings should be placed at the beginning of the module before any statements.

When we request help on the module, we get the useful info including the module docstring and each function docstring.

Comments

Comments

Code is ideally clear enough without ancillary explanation

Sometimes you need to explain why your code is written as it is

Comments in Python start with # and extend to the end of the line

Make Script Executable

Since Python 3.3 Python on Windows also supports the use of the shebang to make Python scripts directly executable with the correct version of the Python interpreter, even to the extent that shebangs that look like they should only work on a UNIX like system will work as expected on Windows.

This works because W**indows Python distributions now use a program called PyLauncher**.
Pylauncher, the executable for which is called simply py.exe will parse the shebang and locate the appropriate version of Python.

You can read more about pie launcher in pep 397

This allows execution of code without using the python

```
that
some
of
its
noisiest
authorities
insisted
on
its
being
received
for
good
or
for
evil
in
the
superlative
degree
of
comparison
only
$
```

It's common on UNIX like systems that have the first line of a script include a special comment called a shebang. This begins with the usual hash as for any other comment, followed by an exclamation mark.

#! /usr/bin/env python

This allows the program loader to identify which interpreter should be used to run the program.

Shebangs have an additional purpose of conveniently documenting at the top of a file, whether the python code there Python2 or Python3. The exact details of your shebang command depend on the location of python on your system.

Typical Python3 Shebangs use the unix env program to locate Python3 on your path environment variable, which importantly, is compatible with python virtual environments.

On MAC or Linux, we must mark our script as executeable using the command chmod +x words.py before the shebang will have any effect.

Pylauncher

Pylauncher

1. Associated with `*.py` files
2. Executable is `py.exe` and is on the `PATH`
3. Parse the shebang and locate Python

Windows Command Prompt

```
> words.py http://sixty-north.com/c/t.txt
```

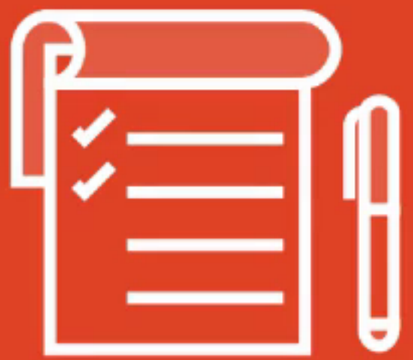
Windows PowerShell

```
PS> .\words.py http://sixty-north.com/c/t.txt
```

PEP 397

Describes PyLauncher

Summary



Python code is generally placed in *.py files

Execute modules by passing them as the first argument to Python

All top-level statements are executed when a module is imported

Define functions with the `def` keyword

Return objects from functions with the `return` keyword

`return` without an argument returns `None`, as does the implicit return

Summary

Use `__name__` to determine how a module is being used

`if __name__ == '__main__'` lets our module be executable and importable

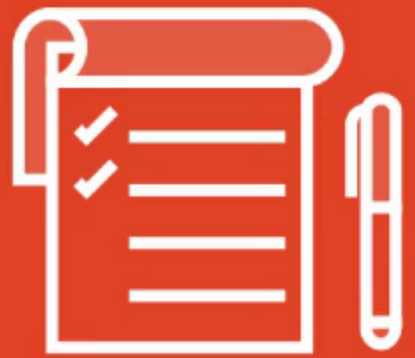
A module is executed once, on first import

`def` is a statement which binds code to a name

`sys.argv` contains command line arguments

Dynamic typing supports generic programming

Summary



Functions can have docstrings

`help()` can retrieve docstrings

Modules can have docstrings

Python comments start with #

Program loaders can use `#!` to determine which Python to run