Practical Programming
in Python

Inspired by 'Practical Programming' by Paul Gries, Jennifer Campbell, Jason Montojo

Lecture 6: Program Organization

Modules, Name Spaces, Main Programs, Libraries

Kurt Rinnert, Kate Shaw

Physics Without Frontiers

The Abdus Salam
ICTP International Centre
for Theoretical Physics

Abstract

"You can't write perfect software."
– Andrew Hunt, The Pragmatic Programmer

Almost no program is written by one programmer alone. It is much more common – and productive – to make use of the code that other programmers have written before. We also frequently work in teams on the same program. Both, using existing code and team work, require some means of program organization. The main tool for this in Python are modules.

Overview

- Importing modules
- Importing objects from modules
- Writing modules
- Module execution
- Main programs
- Libraries

Modules are Python's way of organizing programs.

Modules

- A *module* is a collection of expressions grouped together in a single file
- Typically the expressions define functions and variables
- The variables and functions are usually related to one another
- For example, the `math` module contains the variable `pi` and the function `sqrt` (square root)

Python comes with hundreds of modules, we explore some of them in this lecture.

Importing Modules

- To access variables and functions from a module, we have to *import* it:

```
>>> import math
```

- Importing a module creates a new variable with the same name as the module:

```
>>> type(math)
<class 'module'>
```

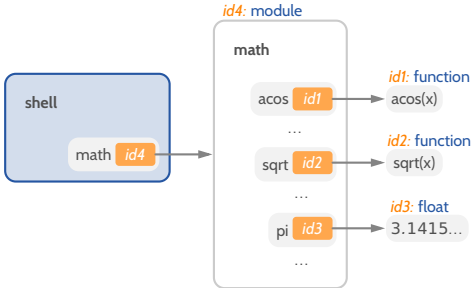
- You can use the built-in function `help` to learn what a module contains:

```
>>> help(math)

Help on module math:

NAME
    math

DESCRIPTION
    This module is always available.  It
    provides access to the mathematical
    functions defined by the C standard.

FUNCTIONS
    acos(...)
[...]
```

Try `help(math)` in the Python shell and have a look yourself.

After Importing the `math` Module

- The statement **`import`** `math` creates a variable called `math`
- The variable `math` refers to a module object
- This object contains all the names defined in the module
- The names refer to function or variable objects



Great! Our programs can now use all kinds of mathematical functions!

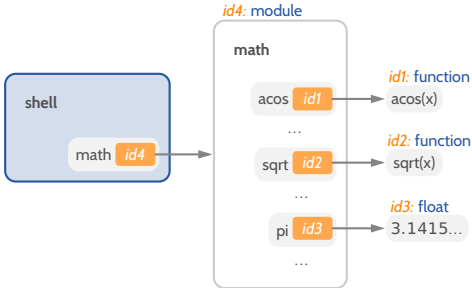
Using Functions from the `math` Module

- Let's try to calculate a square root:

```
>>> import math
>>> sqrt(9.0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'sqrt' is not defined
```

- We need to access it via the `math` variable using the *dot operator*:

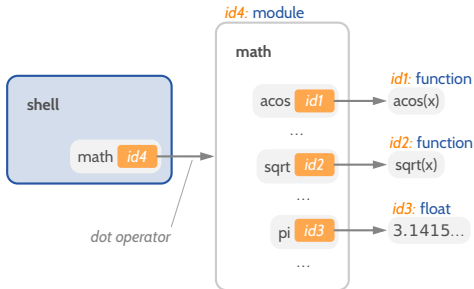
```
>>> import math
>>> math.sqrt(9.0)
3.0
```



Python didn't find the `sqrt` function in the first example. Why is this not surprising?

Namespaces

- `math` is a variable in the current *namespace*
- The `math` module has its own namespace
- The `sqrt` function lives in the `math` module's namespace
- We use the dot operator `.` to navigate namespaces

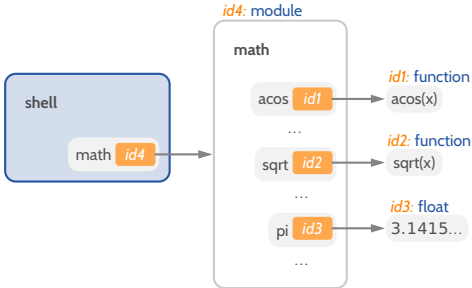


Namespaces avoid *name collisions*.

Using Variables from the `math` Module

- The `math` module also defines some variables:

```
>>> import math
>>> math.pi
3.141592653589793
>>> radius = 5.0
>>> print('area is', math.pi * radius ** 2)
area is 78.53981633974483
```



- Python does not have constants (a flaw?):

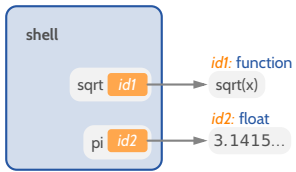
```
>>> import math
>>> math.pi = 3.0
>>> radius = 5.0
>>> print('area is', math.pi * radius ** 2)
area is 75.0
```

Never, we repeat, *never* re-define imported variables!

Direct Imports from a Module

- We can directly import objects into the current namespace:

```
>>> from math import sqrt
>>> from math import pi
>>> sqrt(9.0)
3.0
>>> radius = 5.0
>>> print('area is', pi * radius ** 2)
area is 78.53981633974483
```



- We can also import *all* objects from a module:

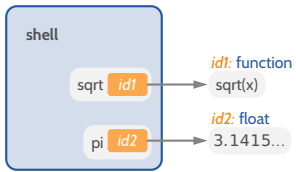
```
>>> from math import *
>>> sqrt(9.0)
3.0
```

Directly importing all objects is a bad idea. Why is that?

Direct Imports from a Module

- This does *not* create a variable `math` referring to the module:

```
>>> from math import sqrt
>>> from math import pi
>>> math.pi
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'math' is not defined
>>> pi
3.141592653589793
```

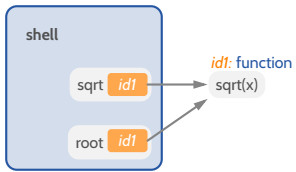


Direct imports act like assignments in the current scope. Use with care.

Import Aliases

- We can choose the name referring to an object or module we import:

```
>>> from math import sqrt
>>> from math import sqrt as root
>>> sqrt(9.0)
3.0
>>> root(16.0)
4.0
```



- There are community conventions for some libraries. For example:

```
import numpy as np
import matplotlib.pyplot as plt
```

This is explicit and readable. Follow established conventions.

Defining Your Own Modules

- In lecture 3 we learned how to write programs and save them to a `.py` file
- We already created a module then!
- Now we are going to extend this idea
- Create a file `temperature.py` containing the function definitions on the right
- Then **`import`** and use it:

```
>>> import temperature
>>> c = temperature.convert_to_celsius(33.3)
>>> temperature.above_freezing(c)
True
```

```
def convert_to_celsius(fahrenheit):
    """
    Convert Fahrenheit to Celsius.

    Examples:

        >>> convert_to_celsius(75)
        23.88888888888889
    """
    return (fahrenheit - 32.0) * (5.0/9.0)

def above_freezing(celsius):
    """
    Return True iff celsius is above freezing.

    Examples:

        >>> above_freezing(6.1)
        True
        >>> above_freezing(-5)
        False
    """
    return celsius > 0
```

Any file with Python expressions can act as a module.

What Happens During Import

- Let's try something else
- Create a file `experiment.py` with the following content:

```
print("The panda's scientific name is 'Ailuropoda melanoleuca'")
```

- Run the program and note the output
- Now import the file `experiment.py` in the Python shell:

```
>>> import experiment
The panda's scientific name is 'Ailuropoda melanoleuca'
```

Python *executes* modules when it imports them.

What Happens During Import

- What happens when we import the same module twice?
- Let's try it!
- Clear the Python shell
- Then try the following in the shell:

```
>>> import experiment
The panda's scientific name is 'Ailuropoda melanoleuca'
>>> import experiment
```

- The module is only executed on the *first* import

How might this work? Why is that a good idea?

Picking up Module Changes

- What if a module changes?
- Let's try it!
- Keep the shell from before
- Then change the contents of `experiment.py` to

```
print("The koala's scientific name is 'Phascolarctos cinereus'")
```

- Importing experiment again does not make difference
- We have to force a proper import to pick up the change:

```
>>> import experiment
>>> import imp
>>> imp.reload(experiment)
The koala's scientific name is 'Phascolarctos cinereus'
```

Under which circumstances might this be useful?

Selecting what to Run on Import

- Sometimes we want some module code to only run when we run the module directly (as opposed to importing it)
- Python defines a special string variable `__name__` in every module to help with this
- Create a module `echo.py`:

```
print('__name__ equals', __name__)
```

- Now run the above program:

```
__name__ equals __main__
```

- Import the module in a clean shell:

```
>>> import echo
__name__ equals echo
```

This is commonly used to run automatic tests.

Semiautomatic Testing

- We recommended adding examples to the docstring in our function design recipe
- Now we will unleash their true power
- Add the following to the `temperature.py` module:

```
if __name__ == "__main__":
    import doctest
    print('testing...')
    doctest.testmod()
```

- Now run the program

```
testing...
```

No output is good. All tests have passed.

When Tests Fail

- Let's introduce a *bug*

```
return fahrenheit - 32.0 * (5.0/9.0)
```

- And run the program again:

```
testing...
**********************************************************************
File "temperature.py", line 7, in __main__.convert_to_celsius
Failed example:
    convert_to_celsius(75.0)
Expected:
    23.88888888888889
Got:
    57.22222222222222
**********************************************************************
1 items had failures:
   1 of   1 in __main__.convert_to_celsius
***Test Failed*** 1 failures.
```

Make a habit of running automatic tests.

What Should Go Into a Module?

- A Module should define objects that have some logical connection
- What constitutes a logical connection is somewhat a matter of opinion
  - Should `math` contain matrix operations?
  - Or should they be in a linear algebra module?
- Don't bundle things just because you are the author

> **If you can't sum up the purpose of a module in a short docstring, there is probably something wrong.**

You have to develop experience and taste. Read other people's code!

Exercises Lecture 6