

The Python Language



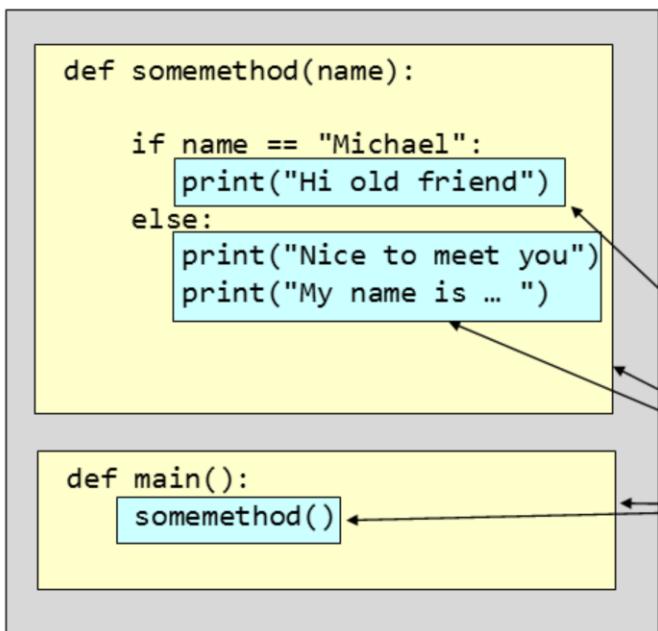
Global Knowledge®

Objectives

- Learn the basics of the Python language
- Define code blocks
- Work with variables
- Test for true / false using various conditionals
- Write loops to work with sets of data
- Consume external code via packages and modules
- Create isolated development environments
- Use deterministic clean up for recovering resources

The 'shape' of a Python program

- Python defines code blocks (known as **suites** in Python) using whitespace and colons.



Things to note:

- No semicolons
- Code blocks start with `:
- Whitespace really really matters
- There are no braces
- There are no parentheses
- Tabs are not your friend

Code suites

Variables

- Declaring variables is no-nonsense

```
name = "Jeff"  
print( "Hi there " + name )  
name = 42  
print( name )
```

Note:

- You do not declare the type
- Variables are not strongly-typed (but types are)
- Discover current type via **type(var)**
- Compare references with **id(var)** and **var1 is var2**
- Compare values with **var1 == var2**

Variables [scope]

- Variable scope
 - Python does not have strict block scope like many C-based languages
 - Not restricted to the declaring scope
 - Scope is global or function level
 - Initialize first => function scope
 - Use first => global scope (must be explicit)

```
num1 = 40

if num1 > 10:
    num2 = 2
    print("Num from if: " + str(num2))

print("Looks like the number is " + str(num1 + num2))

# prints 'Looks like the number is 42'
```

<http://docs.python.org/3.3/faq/programming.html>

What are the rules for local and global variables in Python?

In Python, variables that are only referenced inside a function are implicitly global. If a variable is assigned a new value anywhere within the function's body, it's assumed to be a local. If a variable is ever assigned a new value inside the function, the variable is implicitly local, and you need to explicitly declare it as 'global'.

Though a bit surprising at first, a moment's consideration explains this. On one hand, requiring `global` for assigned variables provides a bar against unintended side-effects. On the other hand, if `global` was required for all global references, you'd be using `global` all the time. You'd have to declare as `global` every reference to a built-in function or to a component of an imported module. This clutter would defeat the usefulness of the `global` declaration for identifying side-effects.

Variables [global scope]

- Variable scope
 - **global** keyword can promote scope
 - **nonlocal** keyword can allows us to share scopes (for closures)

```
sharedVal = 3

def method1():
    global sharedVal
    if sharedVal == 3:
        sharedVal = 7

    sharedVal += 1

method1()
print( sharedVal ) # prints 8
```

Nonlocal is only in Python 3, not 2.

Comments

- Comments are indicated with the **# character**
- They last for rest of a single line
- Class / function documentation comments use an alternate syntax (details later)

```
name = "Jeff"  
print( "Hi there " + name ) # use string concat  
num = 42  
# I wouldn't try this one!  
print( "Hi there " + num) # this is an error!
```

Conditionals: Truthiness

- The following are considered **False**
 - `None`
 - `False`
 - zero of any numeric type, for example, `0`, `0L`, `0.0`, `0j`.
 - any empty sequence, for example, `''`, `()`, `[]`.
 - any empty mapping, for example, `{}`.
 - instances of user-defined classes, if the class defines a `__nonzero__()` or `__len__()` method, which returns **False**
- Everything else is **True**

Conditionals: if statements

- **if** statements are simple suites
- Additional tests are done using **elif** (not **else if**)
- **and** and **or** are words (not symbols, e.g. **&&** and **||**)
- **else** statements can appear at the end

```
if len(name) > 5:  
    print('Oh you have a long name!')  
elif len(name) == 5:  
    print('Let me guess, your name is Sarah?')  
elif len(name) == 4 and name[0] == 'T':  
    print('Let me guess, your name is Todd?')  
else:  
    print('Filling out forms must be quick for you!')
```

There is no switch statement in Python.

Conditionals: Ternary statements

- Ternary statements are compressed **if** / **else** suites
- They are meant to be readable rather than concise

```
name = "Jeff"
val = "short name" if len(name) < 5 else "long name"
print(val) # prints 'short name'
```

Empty code suites (blocks): Pass statement

- Sometimes you want an empty block
 - maybe you commented out some code
 - maybe you're sketching out the structure
- The **pass** keyword keeps things running

```
if len(name) > 5:  
    pass
```

Empty code blocks are not allowed.

While loops

- While loops run until a condition becomes false

```
num1 = 1
num2 = 2

while num1 < 100:
    num1 = num1 * num2
    print( num1 )

# prints 2,4,8,16,32,64,128
```

For in loops

- For loops in Python fundamentally work on iterable sets
 - There is no index-based looping construct!
 - Many types are iterable
 - lists, sets, dictionaries, strings, files, classes, ...

```
name = "Jeff"

for ch in name:
    print( ch, end=', ' )

# prints 'J', 'e', 'f', 'f',
```

For in loops [with indexes]

- For loops *can* use an index
 - Uses **range** function
 - But it's less Pythonic
 - range was considered harmful in Python 2.7 (it's not in 3)

```
name = "Jeff"

for i in range( len(name) ):
    print( name[i] )

# prints 'J', 'e', 'f', 'f',
```

In Python 2 range creates a (potentially large) list to iterate over. Instead, use xrange which has the same behavior as range in Python 3.

For in loops [with indexes (better)]

- Index + element for loops can be combined
 - uses **enumerate** function
 - returns tuples of (index, element)
 - more Pythonic

```
name = "Jeff"

for t in enumerate(name):
    print( t )

# prints (0, 'J'), (1, 'e'), (2, 'f'), (3, 'f')
```

Loops and else statements

- All looping constructs support a final clause using **else**
- Only runs if
 - the loop completes **without** early breaks
 - the loop completes but never runs
- Don't forget the key line in the Zen of Python
 - "... that way may not be obvious at first unless you're Dutch."

```
print("Else loop test")
v = 7
while v < 10:
    v += 1
    print(v)
    if v == 9:
        break;
else:
    print("else v is now " + str(v))
```

Consuming libraries

- Python can access functionality from other modules, packages, and libraries using the **import** statement.

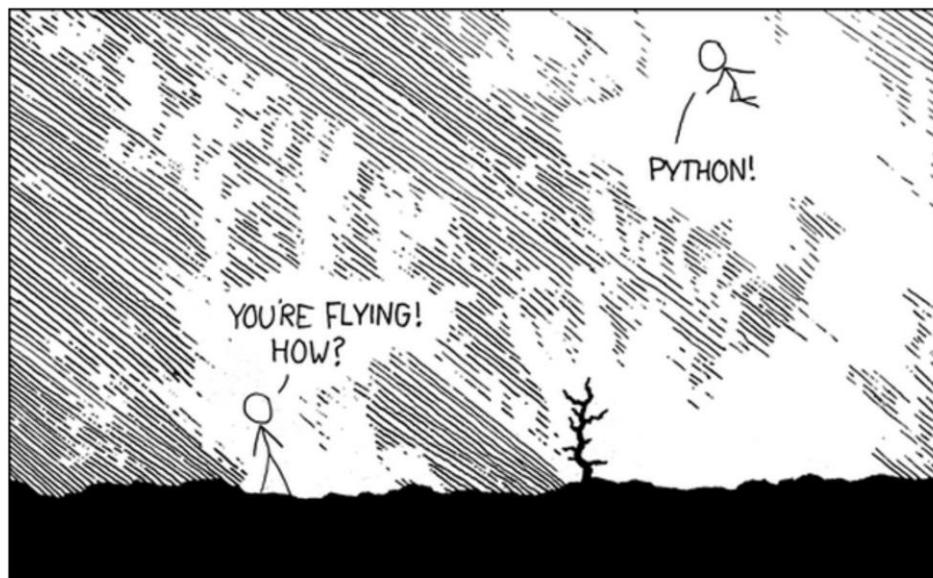


Image credit: XKCD: <http://xkcd.com/353/>

Consuming libraries [import keyword]

- Python can access functionality from other modules, packages, and libraries using the **import** statement.
- Import gives you access to
 - other scripts you have written
 - modules and packages from third parties
 - components of the standard library

Consuming libraries [standard library]

- Accessing the standard library

```
import sys

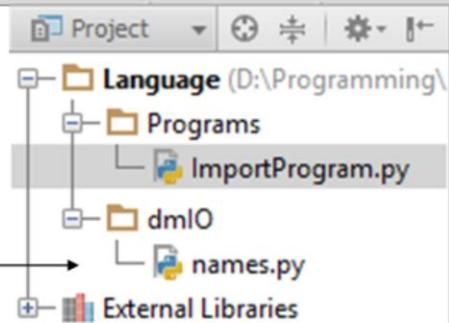
print("Please enter your name: ", end=' ')
sys.stdout.flush()
name = sys.stdin.readline()

print("Nice to meet you " + name)
```

Consuming libraries [other scripts]

- Accessing other scripts

Include relative paths for working folder



```
import dmIO.names

userName = dmIO.names.queryUserName()
print("Nice to meet you " + userName)

# Sample execution:
# Please enter your name: Jeff
# Nice to meet you Jeff
```

Consuming libraries [importing your script]

- When your scripts are imported, they may run code you did not intend to run
 - Use the `__name__` convention to test if your script is the main script.

```
if __name__ == "__main__":
    # your code here
```

Consuming libraries [import details]

- Import has several forms

```
import dmIO.names # default: keep namespace

userName = dmIO.names.queryUserName()
print("Nice to meet you " + userName)
```

```
from dmIO.names import queryUserName # single method / class
import

userName = queryUserName()
print("Nice to meet you " + userName)
```

```
from dmIO.names import * # import everything, no namespaces

userName = queryUserName()
print("Nice to meet you " + userName)
```

The last one is considered bad style because it may accidentally overwrite names in your namespace.

Consuming libraries [third-party packages]

- Python has several package managers which install and upgrade third-party packages
 - **setuptools** (download and run [ez setup.py](#))
 - **pip** (default since 3.4, else download and run [get-pip.py](#), requires setuptools)
- These are similar to
 - NPM from node.js [\[1\]](#)
 - NuGet from .NET [\[2\]](#)
 - Gems from Ruby [\[3\]](#)

Pip is now default since Python 3.4.

Consuming libraries [third-party packages]

- Installing packages
 - `pip install <packagename>`

Install [requests](#) package



```
Developer Command Prompt for VS2012
(env) C:\pip install requests
Downloading/unpacking requests
  Downloading requests-2.0.1.tar.gz (412kB): 412kB downloaded
  Running setup.py egg_info for package requests

Installing collected packages: requests
  Running setup.py install for requests

Successfully installed requests
Cleaning up...
```

Use package



```
Developer Command Prompt for VS2012 - python
(env) C:\Users\Michael Kennedy\Documents\Python\SecondProj>python
>>> import requests
>>> r = requests.get("http://www.develop.com")
>>> len(r.text)
31635
>>>
```

Consuming libraries [virtual environments]

- Virtual environments allow multiple Python projects that have different (and potentially conflicting) requirements, to coexist on the same computer.
- With virtual environments you can
 - Store multiple versions of a package
 - Store packages in non-admin users folders (sudo install not required)
 - Clearly express dependencies for a deployment

Consuming libraries [using virtualenv]

- Install virtual environments
 - `sudo pip install virtualenv`
- Create a virtual environment in working folder
 - `virtualenv .\env`
- Install your packages (e.g. requests)
 - `.\env\scripts\pip install requests`
- Activate your terminal / cmd session
 - `.\env\scripts\activate.bat` or `source (on OS X)`
- Run python scripts and commands as usual
 - but will target the virtual environment
- Run 'deactivate' to turn off the virtual environment

Note: Python 3.3+ includes a built-in way to do this:
<http://docs.python.org/dev/library/venv.html>

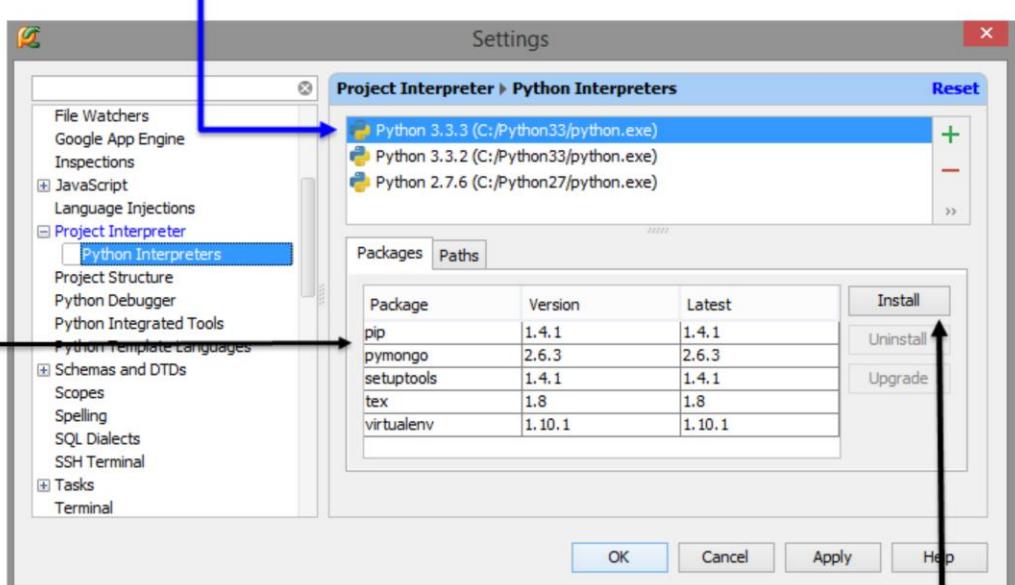
Use deactivate to turn off the virtualenv.

Use pyvenv for Python 3.3+

Consuming libraries [PyCharm]

- PyCharm has support for package management

For a selected interpreter or virtual environment

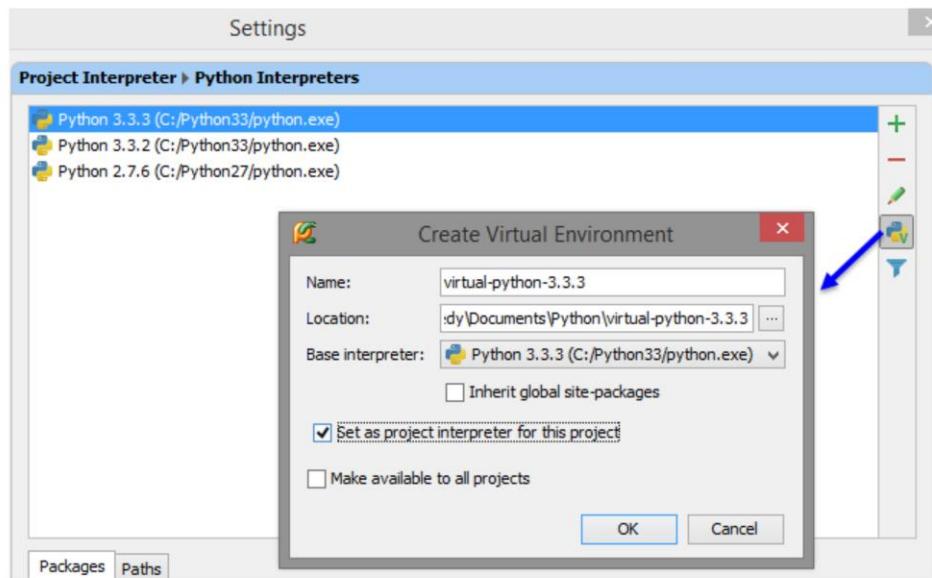


Manage / update installed packages

Install new packages

Consuming libraries [PyCharm]

- PyCharm has support for virtual environments
 - Can isolate environment
 - Can inherit global package settings



Deterministic clean up

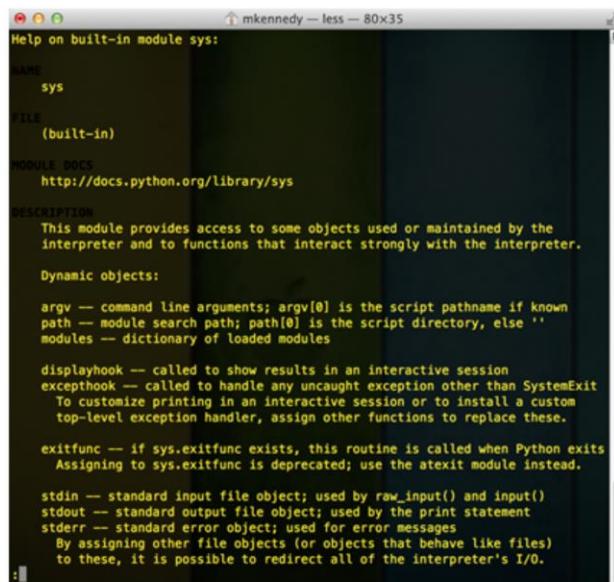
- Python memory management uses
 - Eager reference counting
 - Secondary generational garbage collector (for cycles)
- To ensure timely clean up use the **with** statement

```
with open('foo.txt') as fin:  
    # perform some action with fin  
    pass # <-- work with fin  
    # cleanup happens on exit code suite
```

'with' uses a context manager, which is an object that supports `_enter_` and `_exit_`

Getting help

- Python has good documentation with examples
 - Visit <http://docs.python.org/3.3/contents.html>
 - Just Google it (typically fastest access to docs.python.org)
 - Type **help(class)** or **help(namespace)** (hint: q to quit)



The screenshot shows a terminal window titled "mkennedy — less — 80x35". The content is the help documentation for the built-in module "sys". The text is in white on a black background. It includes sections for NAME, FILE, MODULE DOCS, and DESCRIPTION, followed by a list of dynamic objects and their descriptions.

```
Help on built-in module sys:

NAME
    sys

FILE
    (built-in)

MODULE DOCS
    http://docs.python.org/library/sys

DESCRIPTION
    This module provides access to some objects used or maintained by the
    interpreter and to functions that interact strongly with the interpreter.

    Dynamic objects:

    argv -- command line arguments; argv[0] is the script pathname if known
    path -- module search path; path[0] is the script directory, else ''
    modules -- dictionary of loaded modules

    displayhook -- called to show results in an interactive session
    excepthook -- called to handle any uncaught exception other than SystemExit
        To customize printing in an interactive session or to install a custom
        top-level exception handler, assign other functions to replace these.

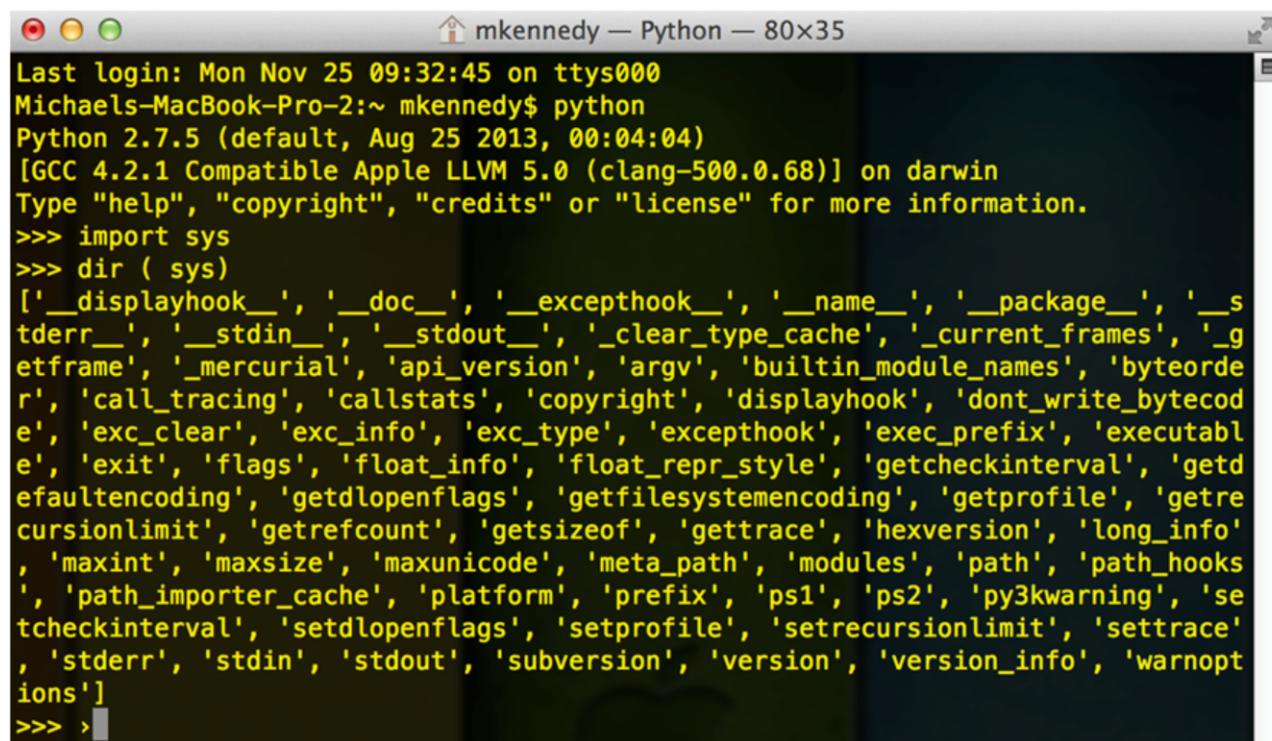
    exitfunc -- if sys.exitfunc exists, this routine is called when Python exits
        Assigning to sys.exitfunc is deprecated; use the atexit module instead.

    stdin -- standard input file object; used by raw_input() and input()
    stdout -- standard output file object; used by the print statement
    stderr -- standard error object; used for error messages
        By assigning other file objects (or objects that behave like files)
        to these, it is possible to redirect all of the interpreter's I/O.

:
```

Getting help

- Not everything is documented, but you can still browse it with the `dir(namespace)` command.



```
Last login: Mon Nov 25 09:32:45 on ttys000
Michaels-MacBook-Pro-2:~ mkennedy$ python
Python 2.7.5 (default, Aug 25 2013, 00:04:04)
[GCC 4.2.1 Compatible Apple LLVM 5.0 (clang-500.0.68)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import sys
>>> dir( sys )
['__displayhook__', '__doc__', '__excepthook__', '__name__', '__package__', '__stderr__', '__stdin__', '__stdout__', '__clear_type_cache__', '__current_frames__', '__getframe__', '__mercurial__', 'api_version', 'argv', 'builtin_module_names', 'byteorder', 'call_tracing', 'callstats', 'copyright', 'displayhook', 'dont_write_bytecode', 'exc_clear', 'exc_info', 'exc_type', 'excepthook', 'exec_prefix', 'executable', 'exit', 'flags', 'float_info', 'float_repr_style', 'getcheckinterval', 'getdefaultencoding', 'getdlopenflags', 'getfilesystemencoding', 'getprofile', 'getrecursionlimit', 'getrefcount', 'getsizeof', 'gettrace', 'hexversion', 'long_info', 'maxint', 'maxsize', 'maxunicode', 'meta_path', 'modules', 'path', 'path_hooks', 'path_importer_cache', 'platform', 'prefix', 'ps1', 'ps2', 'py3kwarning', 'setcheckinterval', 'setdlopenflags', 'setprofile', 'setrecursionlimit', 'settrace', 'stderr', 'stdin', 'stdout', 'subversion', 'version', 'version_info', 'warnoptions']
>>> >
```

Summary

- Python uses whitespace and colons to define blocks
- Variables do not require type definitions
- There are two types of conditionals
- Python does not have an index for loop
- Packages are imported using the import keyword
- Virtual environments allow us to work in clean envs
- With statement allows for safe use of recourse-based data