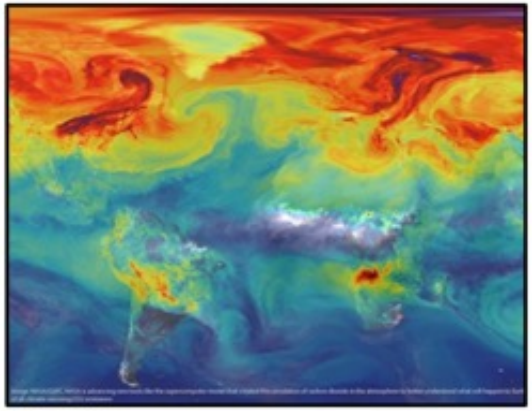





Geo Data Science with Python (GEOS-5984/4984)

Prof. Susanna Werth

Topic: Python Modules & Classes

Today's music is from: Carmen and me

Please keep sending me your song suggestions through Canvas!

Notes/Reminders

- **E04 on Statements and Functions**
  due Friday 30 September

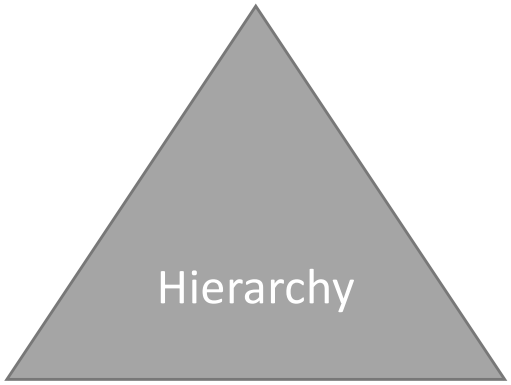
Today

- Modules
- Scripts

Modules

Python Conceptual Hierarchy

- Python program components
  - Programs are composed of ***modules***
  - Modules contain ***statements***
  - Statements contain ***expressions***
  - Expressions create and process ***objects***



- *Objects* are data elements (e.g. variables, functions, ...)
- *Expression* is a **combination of one or more objects** that the programming language interprets and computes to **produce another object**. They are embedded in statements.
- *Statements* code the larger logic of a program (e.g. iteration, assignment, selections, ...)
- ***Modules*** are highest-level organization unit, packages code for reuse

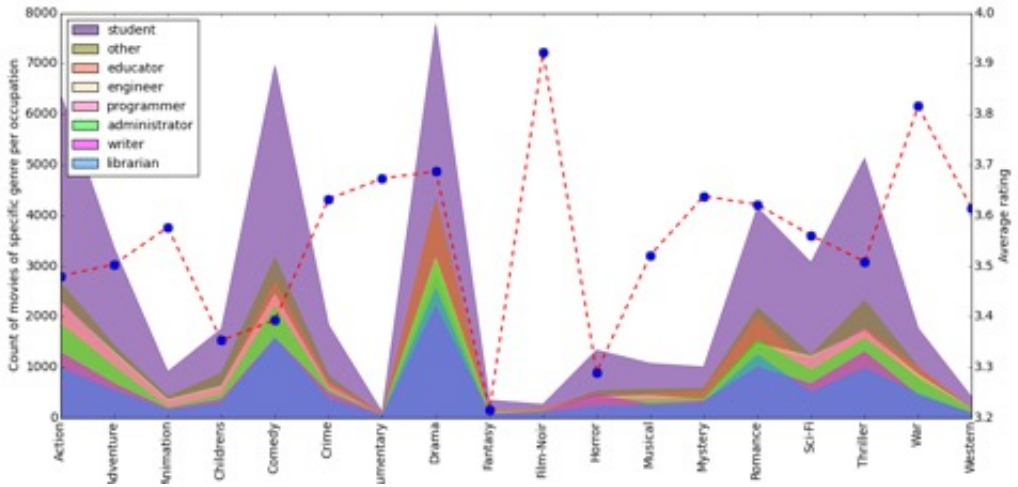
What is a Module?

- Highest-level program organization unit
- Packages code and data for reuse

- Core idea behind Python *program architecture*
- **A Library of Python code**

Useful Python Modules

- **IPython**: python prompt on steroids
- **NumPy**: advance math functionalities
- **SciPy**: library of algorithms and mathematical tools
- **Matplotlib**: numerical plotting library
- **Pandas**: data structure and analysis tools
- **Scikit-learn**: classification, regression and clustering algorithms, interoperates with NumPy and SciPy



Purpose of Modules

- **Code reuse**
  - Save code permanently, reload and rerun
  - Place to define names: attributes
  - Groups functionality into reusable units (modular design)
- **System namespace partitioning**
  - Self-contained: “Everything lives in a module”
  - Code and objects always enclosed in modules (similar to local scopes in functions)
  - Grouping system components
- **Implementing shared services or data**
  - Useful for Implementing components to be shared across systems
  - Single copy of a global object coded in a module can be imported by many users/clients

Core Object Types

Table 4-1. Built-in objects preview

Object type	Example literals/creation
Numbers	`1234, 3.1415, 3+4j, Decimal, Fraction`
Strings	`'spam', "guido's", b'a\x01c'`
Lists	`[1, [2, 'three'], 4]`
Dictionaries	`{'food': 'spam', 'taste': 'yum'}`
Tuples	`(1, 'spam', 4, 'U')`
Files	`myfile = open('eggs', 'r')`
Sets	`set('abc'), {'a', 'b', 'c'}`
Other core types	Booleans, types, `None`
Program unit types	Functions, modules, classes
Implementation-related types	Compiled code, stack tracebacks

Lutz, M. (2013). *Learning Python* (5th ed.). O'Reilly Media, Inc.

What defines a Module?

- **Each Python file is a module** (no special code needed, but usually *.py* extension)
- **Modules import other modules** to access names (=object references) they define

- Module must be **imported** explicitly to use it's names (content)

Standard Library Modules

Over 200...

Python Module Index: https://docs.python.org/3/py-modindex.html

Module	Description	Examples
os	Interacting with the operating system	os.system(command)
sys	Interpreter-related tools	sys.path, sys.exit()
string	constants and variables for processing strings (most available as methods)	string.capwords()
math	Floating point math functions	math.pow()
shutil	High-level file operations	shutil.copyfile(), shutil.move()
re	Regular expression pattern matching tools for advanced string processing	re.match('c', 'abcdef')
...		

Python Statements

Table 10-1. Python statements

Statement	Role	Example
Assignment	Creating references	`a, b = 'good', 'bad'`
Calls and other expressions	Running functions	`log.write("spam, ham")`
`print` calls	Printing objects	`print('The Killer', joke)`
`if/elif/else`	Selecting actions	`if "python" in text:` `  print(text)`
`for/else`	Iteration	`for x in mylist:` `  print(x)`
`while/else`	General loops	`while X > Y:` `  print('hello')`
`pass`	Empty placeholder	`while True:` `  pass`
`break`	Loop exit	`while True:` `  if exittest(): break`
`continue`	Loop continue	`while True:` `  if skiptest(): continue`
`def`	Functions and methods	`def f(a, b, c=1, *d):` `  print(a+b+c+d[0])`
`return`	Functions results	`def f(a, b, c=1, *d):` `  return a+b+c+d[0]`
`yield`	Generator functions	`def gen(n):` `  for i in n: yield i*2`

`global`	Namespaces	`x = 'old'`
		`def function():`
		`  global x, y; x = 'new'`

`nonlocal`	Namespaces (3.X)	`def outer():`
		`  x = 'old'`
		`  def function():`
		`  nonlocal x; x = 'new'`

`import`	Module access	`import sys`
`from`	Attribute access	`from sys import stdin`

`class`	Building objects	`class Subclass(Superclass):`
		`  staticData = []`
		`  def method(self): pass`

`try/except/ finally`	Catching exceptions	`try:`
		`  action()`
		`except:`
		`  print('action error')`

`raise`	Triggering exceptions	`raise EndSearch(location)`
`assert`	Debugging checks	`assert X > Y, 'X too small'`
`with/as`	Context managers (3.X, 2.6+)	`with open('data') as myfile:` `  process(myfile)`

`del`	Deleting references	`del data[k]`
		`del data[i:j]`
		`del obj.attr`
		`del variable`

Lutz (2013), Ch. 10, pp330-331

How to import a Module?

- Modules are imported with these keywords:

```
import module
import module as name

from module import name
```

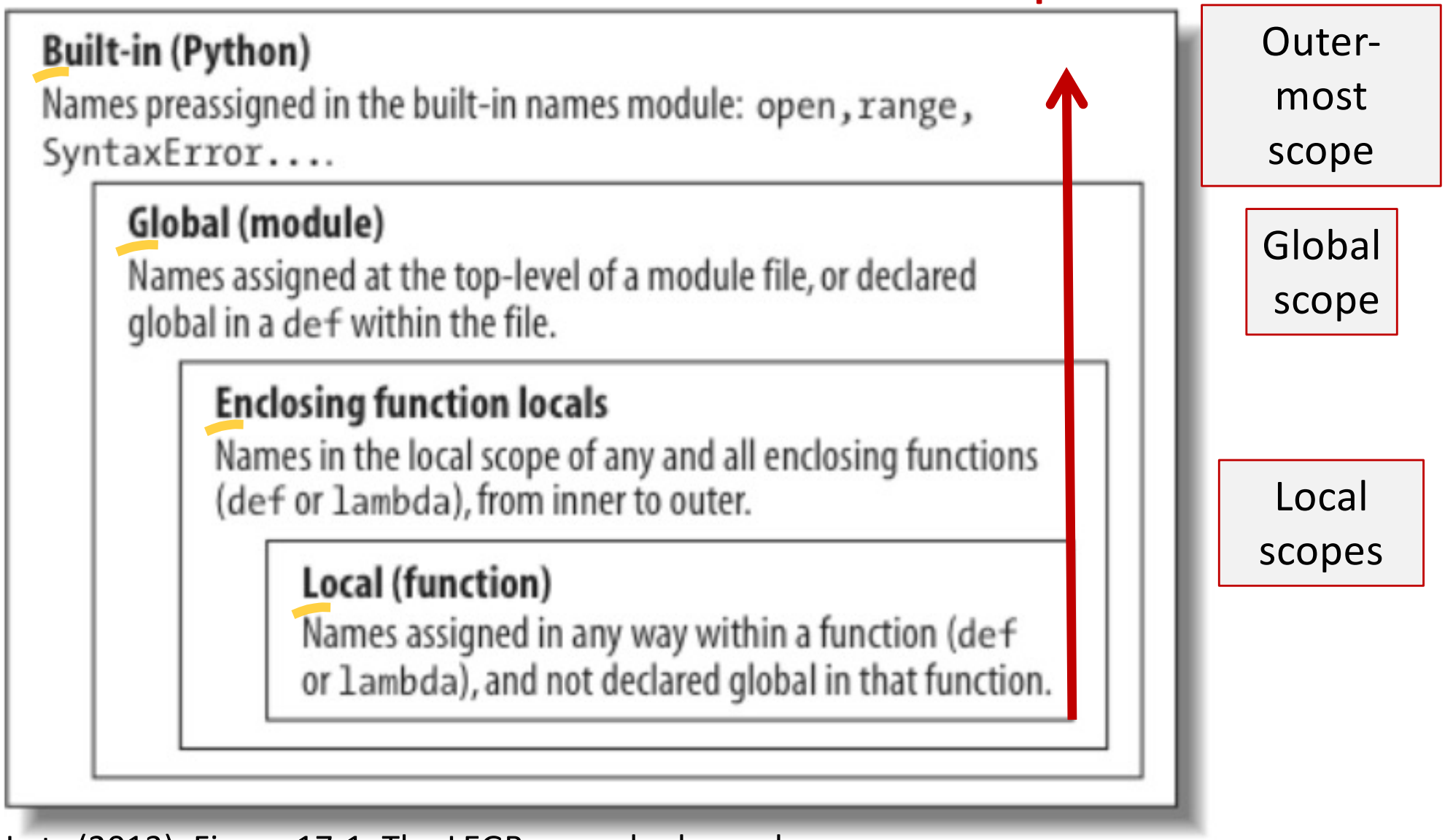
fetches module as a whole (makes its tools accessible)

fetches particular names from a module

Examples

- Import a module (math)
- Use objects defined in a module
- Other import options
  - Renaming
  - Importing single attributes or functions
- Importing sub-packages of a module
- Namespace of a module

LEGB Scope Lookup Rule



Lutz (2013), Figure 17-1. The LEGB scope lookup rule

Scopes of Modules

Don’t be fooled by the word “global”:

- Global scope spans a single file/module only
- Single, all-encompassing global file-based scope not existing
- Names at top level of a file are global to code within that file only
- **Module must be imported explicitly to use it’s names**
- Importing a module gives access to all names assigned at top level of a module’s file

Conclusion: “global” = “module”

Exception: in-place changes to objects (later, Lutz pg. 508)

Avoid the following!

- Using the ‘from’ statement often
  - from X import Y
  - Might cause conflict of attribute names
- Confusing names when renaming on import
  - import matplotlib as math
  - import matplotlib as m

Use Module Attributes

Attributes = Tools / content of a module

= Variable names attached to a module

Contents is made available through module name an the **period syntax:**

```
Modulename.attributename()
```

Syntax similar to **object-specific attributes** (“methods”)

Built-in Attributes

- Names automatically assigned to a module
- Predefined built-in names: with leading and trailing double underscores: __*attibute*__

- Special meaning to interpreter, for example:

__doc__	contains documentation string
__file__	contains full file name of module
__name__	contains module name
__package__	package name, empty for top-level module

Tutorial

Examples

- All attributes of the module math
- Built-in attributes and their content

- What are some attibutes of the module ‘string’?
- Which built-in attribute tells you the purpose of the module?

Tutorial

Write your own Module

Let's create a module of name *hello*

1. Create a python script file 'hello.py'
2. Open the file for editing
3. Ender the following:

```
title = 'The meaning of Life'
```

4. Save the file, and exit.
5. Follow the next steps in a jupyter notebook stored in the same folder as your file 'hello.py'

… Writing your own Module

Content of module/file hello.py:

```
title = 'The meaning of Life'
```

Importing Module:

```
>>> import hello
>>> hello.title
'The meaning of Life'
```

Qualifies use of module name

Alternative:

```
>>> import hello as ho
>>> ho.title
'The meaning of Life'
```

Import (copy) name from module:

```
>>> from hello import title
>>> title
'The meaning of Life'
```

Uses module name **unqualified**: **attribute copied to current namespace**

Namespace of *hello.py*

Which names are defined in the namespace of the module hello?

dir(hello)

Tutorial

Add a docstring

Let’s expand the module ‘hello.py’

6. Add a docstring to your module.
7. Check the content of the variable __doc__:

Tutorial

Expand *hello.py*

Let's expand the module 'hello.py'

8. Reopen the file for editing
9. Add a print statement: `print(title)`
10. Save the file, and exit.
11. Go back to notebook, restart Kernel and clear, or reimport the module:
    - *import importlib*
    - *importlib.reload(hello)*
12. What happens now at import?

Tutorial

Expand *hello.py* – Part II

Let's expand the module 'hello.py'

14. Add a function to the module file:

```
def world():
    print("Hello, World!")
```

15. Reimport the module, & call the function world()!
16. How did you call it?

Scripts

Python modules versus scripts

- Formally a script is also just a module.
- When writing a python program, usually multiple multiple module files are imported
- A code file / module coding an end-user (data analysis) problem is, including import of other modules, is a main or *top-level* file
- The top-level file is sometimes also called a “script”, starting the entire program to be executed.

Anatomy of a Script/Module file

1. **Notes at the top**

```
"""
Put description, function, version, time, author
information
@version  2016-01-22
@author  John @ ASU
"""
```

2. **Import statements**

```
import math
import numpy
import package name
```

3. **Functions**

```
def function(): ...    # Use meaningful names
```

4. **Main Code Sequence**

```
                       # Indent code to organize program
```

5.
```
# use comments to explain code
# use whitespace in expressions and statements
# (surround operators with single space on either side
```

Style Guide for Python Code : http://www.python.org/dev/peps/pep-0008/

Comments!

- # ALWAYS COMMENT!
  - **Best practice is to leave yourself some info**
  - **Or someone that you might share code with!**

Example A

```
print("this is a cool print statment!") #print
```

Example B

```
print("this is a cool print statment!")
#the above print statement writes out the
#important string to the console for your viewing
#pleasure. I did this so that you may read the
#output of the function I created above on line 90
```

Python modules versus scripts

- Some python files (modules) can be used as module to be imported AND as top-level script

For that

- We make use of the built-in attribute __name__

Tutorial

Built-in *__name__*

1. What is the content of the built-in attribute __name__ in a code cell (= main program level)?

2. What is the content of __name__ in another module?
   - Add to your hello.py module: `print(__name__)`

Python modules versus scripts

- Some python files (modules) can be used as module to be imported AND as top-level script

For that

- We make use of the built-in attribute __name__
- We define a code sequence that is called only, if the module is executed from the command line but not when the file is imported (as module).

Script or module import ?

- In a notebook, we are running code from the main program level!

  __name__ = “__main__”

- In a module we are running code at import

  __name__ = <modulename>

- We can check for content of the built-in variable __name__ and perform a selection based on the content:

```
if __name__ == "__main__”:
    do_something
```

main()

- We could define a function holding all the code that should be run if the file is run from the program level (as script)
- A main() function is often used for that purpose

= center code of a file, from which everything runs

```
if __name__ == "__main__":

    main()
```

BUT: Main function is not required and can have any other name

Tutorial

Write your own Script

Let's expand the module 'hello.py'

17. Add a main() function to the module file.
18. Move the 'print(title)' statement into the main function
19. At the bottom of the file add the line:
    ```
    if __name__ == "__main__": main()
    ```
20. Compare when you importing the file as module versus executing it as a script
    - Restart kernel, if needed to reimport the module
    - Executing the python file as a script/program with the magic !:

```
! python hello.py
```

Practice

- **E04 on Statements and Functions**
  due Friday 30 September

- Revise L09 notebook on Modules
- Optional: L08 Section C on Exceptions

Python Statements

Table 10-1. Python statements

Statement	Role	Example
Assignment	Creating references	`a, b = 'good', 'bad'`
Calls and other expressions	Running functions	`log.write("spam, ham")`
`print` calls	Printing objects	`print('The Killer', joke)`
`if/elif/else`	Selecting actions	`if "python" in text:` `  print(text)`
`for/else`	Iteration	`for x in mylist:` `  print(x)`
`while/else`	General loops	`while X > Y:` `  print('hello')`
`pass`	Empty placeholder	`while True:` `  pass`
`break`	Loop exit	`while True:` `  if exittest(): break`
`continue`	Loop continue	`while True:` `  if skiptest(): continue`
`def`	Functions and methods	`def f(a, b, c=1, *d):` `  print(a+b+c+d[0])`
`return`	Functions results	`def f(a, b, c=1, *d):` `  return a+b+c+d[0]`
`yield`	Generator functions	`def gen(n):` `  for i in n: yield i*2`

`global`	Namespaces	`x = 'old'`
		`def function():`
		`  global x, y; x = 'new'`

`nonlocal`	Namespaces (3.X)	`def outer():`
		`  x = 'old'`
		`  def function():`
		`  nonlocal x; x = 'new'`

`import`	Module access	`import sys`
`from`	Attribute access	`from sys import stdin`

`class`	Building objects	`class Subclass(Superclass):`
		`  staticData = []`
		`  def method(self): pass`

`try/except/ finally`	Catching exceptions	`try:`
		`  action()`
		`except:`
		`  print('action error')`

`raise`	Triggering exceptions	`raise EndSearch(location)`
`assert`	Debugging checks	`assert X > Y, 'X too small'`
`with/as`	Context managers (3.X, 2.6+)	`with open('data') as myfile:` `  process(myfile)`

`del`	Deleting references	`del data[k]`
		`del data[i:j]`
		`del obj.attr`
		`del variable`

Lutz (2013), Ch. 10, pp330-331

Why are Exceptions useful?

- Error handling
  - built-in
  - user defined
- Termination actions
- Event notification
- Special-case handling (handle rare conditions)
- Unusual control flows

Python Statements

Table 10-1. Python statements

Statement	Role	Example
Assignment	Creating references	`a, b = 'good', 'bad'`
Calls and other expressions	Running functions	`log.write("spam, ham")`
`print` calls	Printing objects	`print('The Killer', joke)`
`if/elif/else`	Selecting actions	`if "python" in text:` `  print(text)`
`for/else`	Iteration	`for x in mylist:` `  print(x)`
`while/else`	General loops	`while X > Y:` `  print('hello')`
`pass`	Empty placeholder	`while True:` `  pass`
`break`	Loop exit	`while True:` `  if exittest(): break`
`continue`	Loop continue	`while True:` `  if skiptest(): continue`
`def`	Functions and methods	`def f(a, b, c=1, *d):` `  print(a+b+c+d[0])`
`return`	Functions results	`def f(a, b, c=1, *d):` `  return a+b+c+d[0]`
`yield`	Generator functions	`def gen(n):` `  for i in n: yield i*2`

`global`	Namespaces	`x = 'old'`
		`def function():`
		`  global x, y; x = 'new'`

`nonlocal`	Namespaces (3.X)	`def outer():`
		`  x = 'old'`
		`  def function():`
		`  nonlocal x; x = 'new'`

`import`	Module access	`import sys`
`from`	Attribute access	`from sys import stdin`

`class`	Building objects	`class Subclass(Superclass):`
		`  staticData = []`
		`  def method(self): pass`

`try/except/ finally`	Catching exceptions	`try:`
		`  action()`
		`except:`
		`  print('action error')`

`raise`	Triggering exceptions	`raise EndSearch(location)`
`assert`	Debugging checks	`assert X > Y, 'X too small'`
`with/as`	Context managers (3.X, 2.6+)	`with open('data') as myfile:` `  process(myfile)`

`del`	Deleting references	`del data[k]`
		`del data[i:j]`
		`del obj.attr`
		`del variable`

Lutz (2013), Ch. 10, pp330-331