Practical Python

Richard P. Muller
May 18, 2000
Fundamentals
Assignment

- The key to understanding Python is understanding assignment
  - Similar to pointers in C
  - Assignment creates references
  - Functions are pass-by-assignment
  - Names are created when first assigned
  - Names must be assigned before being referenced
    - spam = 'Spam'  # basic assignments
    - spam, ham = 'yum','YUM'  # tuple assignment
    - spam = ham = 'lunch'  # multiple target
  - Can use the copy module for times when you want a new object rather than a pointer to an existing object
Naming rules

• Syntax: (underscore or letter) + (any number of digits or underscores)
  – _rick is a good name
  – 2_rick is not

• Case sensitive
  – Rick is different from rick

• Reserved words:

  and, assert, break, class, continue, def, del, elif, else, except, exec, finally, for, from, global, if, import, in, is, lambda, not, or, pass, print, raise, return, try, while
Expressions

• Function calls
  \[ \text{spam}(\text{ham, eggs}) \]

• List/dictionary reference
  \[ \text{spam}[\text{ham}] \]

• Method calls
  \[ \text{spam}.\text{ham} \]
  \[ \text{spam}.\text{ham}(\text{eggs}) \]

• Compound expressions
  \[ \text{spam} < \text{ham} \text{ and } \text{ham} \neq \text{eggs} \]

• Range tests
  \[ \text{spam} < \text{ham} < \text{eggs} \]
print

• The print command prints out variables to the standard output
  >>> print "a", "b"
  a b
  >>> print "a"+"b"
  ab
  >>> print "%s %s" % (a,b)
  a b

• Notes
  – Print automatically puts in a new line; use print ..., to suppress
  – print(string) is equivalent to sys.stdout(string + '\n')
if and truth testing
if tests

• General format:

    if <test1>:
        <statements1>
    elif <test2>:
        <statements2>
    else:
        <statements3>

• Example:

    x = 'killer rabbit'       # Assignment
    if x == 'roger':
        print 'How\'s Jessica?'
    elif x == 'bugs':
        print 'What\'s up, Doc?'
    else:
        print 'Run away! Run away!'
truth tests

• In general,
  – True means any nonzero number, or nonempty object
  – False means not true: zero number, empty object, or None
  – Comparisons and equality tests return 0 or 1
  – In addition
    
    \[
    \begin{align*}
    X \text{ and } Y & \quad \# \text{true if both } X \text{ and } Y \text{ is true} \\
    X \text{ or } Y & \quad \# \text{true if either } X \text{ or } Y \text{ is true} \\
    \text{not } X & \quad \# \text{true if } X \text{ is false}
    \end{align*}
    \]
  
  – Comparisons
    
    \[
    \begin{align*}
    2 < 3 & \quad \# \text{ true} \\
    3 \leq 4 & \quad \# \text{ true}
    \end{align*}
    \]
  
  – Equality versus identity
    
    \[
    \begin{align*}
    x == y & \quad \# \text{x and y have the same value} \\
    x \text{ is } y & \quad \# \text{x and y are the same object} \\
    & \quad \# \text{ or x points to y}
    \end{align*}
    \]
while and for
while loops

• General format:

  while <test1>:  # loop test
      <statements1>  # loop body
  else:          # optional else
      <statements2>  # run if loop didn't break

• Examples

  while 1:       # infinite loop
      print 'type Ctrl-C to stop me!'  

  a,b = 0,10
  while a < b:
      print a,
      a = a + 1
break, continue, pass, else

- **break**
  - Jumps out of the enclosing loop

- **continue**
  - Jumps to the end of the enclosing loop (next iteration)

- **pass**
  - Does nothing (empty statement place holder)

```python
while <test>:
  <statements>
  if <test2>: break
  if <test3>: continue
  <more statements>
else:
  <still more statements>
```
for loops

• for is a sequence iterator
  – Steps through items in a list, string, tuple, class, etc.

```python
for <target> in <object>:
    <statements>
else:  # optional, didn't hit a break
    <other statements>
```

– Can use break, continue, pass as in while
– Can be used with range to make counter loops
  ```python
  for i in range(10):
      print i
  ```
functions
Why use functions?

- Code reuse
  - Package logic you want to use in more than one place
- Procedural decomposition
  - Split complex task into series of tasks
  - Easier for reader to understand
functions

• `def` creates a function and assigns it a name
• `return` sends a result back to the caller
• Arguments are passed by assignment
• Arguments and return types are not declared

```python
def <name>(arg1, arg2, ..., argN):
    <statements>
    return <value>

def times(x, y):
    return x*y
```
Example function: intersecting sequences

```python
def intersect(seq1, seq2):
    res = []             # start empty
    for x in seq1:
        if x in seq2:
            res.append(x)
    return res
```
Scope rules for functions

• LGB rule:
  – Name references search at most 3 scopes: local, global, built-in
  – Assignments create or change local names by default
  – Can force arguments to be global with global command

• Example
  
  ```
  x = 99
  def func(Y):
    Z = X+Y  #X is not assigned, so it's global
    return Z
  func(1)
  ```
Passing arguments to functions

• Arguments are passed by assignment
  – Passed arguments are assigned to local names
  – Assignment to argument names don't affect the caller
  – Changing a mutable argument may affect the caller

```python
def changer (x,y):
    x = 2  #changes local value of x only
    y[0] = 'hi'  #changes shared object
```
Optional arguments

- Can define defaults for arguments that need not be passed

```python
def func(a, b, c=10, d=100):
    print a, b, c, d

>>> func(1, 2)
1 2 10 100

>>> func(1, 2, 3, 4)
1, 2, 3, 4
```
Modules
Why use modules?

• Code reuse
  – Routines can be called multiple times within a program
  – Routines can be used from multiple programs

• Namespace partitioning
  – Group data together with functions used for that data

• Implementing shared services or data
  – Can provide global data structure that is accessed by multiple subprograms
Modules

- Modules are functions and variables defined in separate files
- Items are imported using `from` or `import`
  
  ```python
  from module import function
  function()
  
  import module
  module.function()
  ```

- Modules are namespaces
  - Can be used to organize variable names, i.e.
    ```python
    atom.position = atom.position - molecule.position
    ```
Built-in functions and convenient modules
Data converters

- Most of these are fairly easy to understand
  - `str(obj)` Return the string representation of obj
  - `list(seq)` Return the list representation of a sequence object
  - `tuple(seq)` Return the tuple representation of a sequence object
  - `int(obj)` Return the integer representation of an object
  - `float(x)` Return the floating point representation of an object
  - `chr(i)` Return the character with ASCII code i
  - `ord(c)` Return the ASCII code of character c

- `min(seq)` Return the smallest element of a sequence
- `max(seq)`
string module

- string contain objects for manipulating strings
  - atof() Convert string to a float
  - atoi() Convert string to an integer
  - capitalize() Capitalize the first character in the string
  - capwords() Capitalize each word in string
  - replace() Replace a substring
  - split() Split string based on whitespace (default)
  - lower() Convert string to lowercase
  - upper() Convert string to uppercase
  - strip() Remove leading and trailing whitespace

- digits abcdefghijklmnopqrstuvwxyz
- uppercase ABCDEFGHIJKLMNOPQRSTUVWXYZ
- letters lowercase + uppercase
- whitespace \t\n\r\v
**re module**

- More advanced version of string, for regular expressions
  - . Match any character but newline
  - ^ Match the start of a string
  - $ Match the end of a string
  - * "Any number of what just proceeded"
  - + "One or more of what just proceeded"
  - | "Either the thing before me or the thing after me"
  - \w Matches any alphanumeric character
  - tomato Matches the string "tomato"
**os module**

- Generic operating system interface
  - `getcwd()` Get the current directory name
  - `listdir()` List the files in a directory
  - `chown()` Change the ownership of a file
  - `chmod()` Change the permissions of a file
  - `rename()` Rename a file
  - `remove()` Delete a file
  - `mkdir()` Create a new directory
  - `system()` Execute command in a subshell
### timing and profiling

- **General timings**
  - `time()` Seconds since first call of `time()`

- **Profile module**
  - `profile.run(func(arg1, arg2))`

<table>
<thead>
<tr>
<th>ncalls</th>
<th>tottime</th>
<th>percall</th>
<th>cumtime</th>
<th>percall</th>
<th>filename</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>8.541</td>
<td>0.086</td>
<td>8.574</td>
<td>0.086</td>
<td>makezer</td>
</tr>
<tr>
<td>100</td>
<td>0.101</td>
<td>0.001</td>
<td>0.101</td>
<td>0.001</td>
<td>one_mul</td>
</tr>
<tr>
<td>1</td>
<td>0.001</td>
<td>0.001</td>
<td>8.823</td>
<td>8.823</td>
<td>do_timi</td>
</tr>
</tbody>
</table>
Running Python scripts
Hello, World!

• Hello, world! with an error:

```plaintext
printf "Hello, world!"  #incorrect -- C function

% python hellof.py
File "hellof.py", line 1
  printf "Hello, World!"
     ^
SyntaxError: invalid syntax
```

• Correct the error:

```plaintext
print "Hello, world!"

% python hello.py
Hello, world!
```
Hello, Name

• Make a simple expansion of Hello, world!
  
  ```python
  name = raw_input("What is your name?")
  print "Hello ", name
  
  % python hello_name.py
  What is your name? Rick
  Hello, Rick
  ```