Introduction into Python

Python 4: Advanced Features

Daniel Polani

Looping in Dicts

Input

```
>>> knights = {'gallahad': 'the pure', 'robin': 'the brave'}
>>> for k, v in knights.iteritems():
...     print k, v
...
```

```
gallahad the pure
robin the brave
```


Enumerating Iteration

Input

```
>>> for i, v in enumerate(['tic', 'tac', 'toe']):
...     print i, v
...
```

```
0 tic
1 tac
2 toe
```


Combining Iterations Over Two Sequences

Input

```
>>> questions = ['name', 'quest', 'favorite color']
>>> answers = ['lancelot', 'the holy grail', 'blue']
>>> for q, a in zip(questions, answers):
...     print 'What is your %s?  It is %s.' % (q, a)
...
```

```
What is your name?  It is lancelot.
What is your quest?  It is the holy grail.
What is your favorite color?  It is blue.
```

System Features

Note: only minimalistic selection

Executable Script: put (on Unix) line with system call of python script `#! /usr/bin/python` at the front. It begins with comment, but bang `!` has special meaning.

Argument Passing: `sys.argv[0]` will contain call to script (or, in special cases the empty string `''` or the option `-c`, see Sec. 2.1.1 of Python tutorial

Documentation string

Code

```
def my_function():
    """Do nothing, but document it."""
    pass
```

Input

```
>>> print my_function.__doc__
```

```
Do nothing, but document it.
```

Input

```
>>> help(my_function)
```

```
Help on function my_function in module __main__:

my_function()
    Do nothing, but document it.
```


Arbitrary Argument Lists

Code

```
def all_args(*args):
    for a in args: print a
```

Input

```
>>> all_args('a', 'b', 'c')
```

```
a
b
c
```


Unpacking Argument Lists

Input

```
>>> range(3, 6) # normal call with separate arguments
```

```
[3, 4, 5]
```

Input

```
>>> args = [3, 6]
>>> range(*args) # call with arguments unpacked from a list
```

```
[3, 4, 5]
```

Lambda Forms

Create Anonymous (Lambda) Functions:

Input

```
>>> def make_incrementor(n):
...     return lambda x: x + n
...
>>> f = make_incrementor(42)
>>> f(0)
```

```
42
```

Input

```
>>> f(1)
```

```
43
```


Modules

Construction: write program in file `fibo.py`

Code

```
# Fibonacci numbers module

def fib(n):    # write Fibonacci series up to n
    a, b = 0, 1
    while b < n:
        print b,
        a, b = b, a+b

def fib2(n): # return Fibonacci series up to n
    result = []
    a, b = 0, 1
    while b < n:
        result.append(b)
        a, b = b, a+b
    return result
```

and call it then with `import fibo` in the main module. This defines the module name. To call the functions, you need to qualify: `fibo.fib2(10)`

Standard Modules

Sys Module: `import sys` makes many important functions available:

- stdin/out
- argument lists
- and other

The `dir()` Function

Question: What does `dir()` do?

Builtin Functions: `dir(__builtin__)`

Usage:

Input

```
>>> import __builtin__
>>> dir(__builtin__)
```

```
['ArithmeticError', 'AssertionError', ...,
'_', '__debug__', '__doc__', '__import__',
'__name__', 'abs', 'apply', 'basestring', ..
'set', 'setattr', 'slice', 'sorted', 'static
'str', 'sum', 'super', 'tuple', 'type', 'uni
'unicode', 'vars', 'xrange', 'zip']
```

Files — Input/Output

File Opening: `file("filename.dat")` opens a file for reading (also `file("filename.dat", "r")`) and `file("filename.dat", "w")` opens it for writing.

Use: if `f` is a file, use `f.write('ablabla')`

For Composed Output:

```
f.write("%s is %f feet tall\n" % ("Jack", 3.
```

Printing Codes:

- **`%s`:** for strings
- **`%d`:** for integers
- **`%f`:** for floats

See documentation for others.

Files – Input

File Reading: (as whole) `f.read()`

File Reading: `f.readline()` reads a single line from the file; *any later calls of the same opened file will only start from the following lines*.

File Reading: `f.readlines()` returns a list of all lines in the file (as strings)

File Reading by Iteration:

```
for line in f:
    print line
```


String Methods

Consider a string `x='abc'`

`__add__(...)`: `x.__add__(y)` is the same as `x+y`

Input

```
>>> s = "abc"
>>> s.__add__("def")
```

```
'abcdef'
```

Input

```
>>> s+"def"
```

```
'abcdef'
```

`__contains__(...)`: `x.__contains__(y)` is the same as `y in x`

Note: generally, double-underscores `'__'` indicate that a method has a special, easier syntax to call it with.

String Methods II

`split()`: splits the current string along whitespaces, returning a list of the splits

`find(sub)`: return the lowest (earliest) index where the substring `sub` is found in the current string

`join(words)`: concatenate the strings in the `words` list, using the present string as separator.

Note: There are many more methods, and even call arguments for the methods shown. Use `'help(str)'` for a full list of methods and call arguments