

Python Tutorial

Day 2

Control: Whitespace

- in perl and C, blocking is controlled by curly-braces
- in shell, by matching block delimiters, `if...then...fi`
- in Python, blocking is controlled by indentation
- lots of people whine about this, remembering FOR-TRAN horrors
- well-written code should indent anyway

```
class Sysinfo:
    def __init__(self):
        self.db = {}
        self.getinfo()

    def getinfo(self):
        f = os.popen(SYSINFO, 'r')
        for line in f.readlines():
            m = line.split("|")
            # Only note lines with all the fields.
            if len(m) >= 4:
                self.db[m[2]] = m[4].strip()
        f.close()
```

Control: Decisions, decisions

- `if` works as expected
- a colon indicates a block is coming, so...

```
if spam > 3:  
    statement1  
elif eggs <= 42:  
    statement2  
    statement3  
else:  
    statement4
```

Control: Conditionals

- expected tests: `>` `>=` `<` `<=` `==` `!=`
- these work on numbers and strings
- comparing other sequences tests in parallel
- booleans: `and` `or` `not`
- sequence membership: `in`, `not in`
- identity (same location in memory): `is`, `is not`

```
>>> "wow" > "nee"  
True  
>>> "wow" < "nee" or 3 < 2  
False  
>>> (1, 2, 3) > (0, 1, 2)  
True  
>>> (1, 2, 3) < (0, 1, 2)  
False  
>>> [1, 2, 3] == [1, 2, 3]  
True  
>>> "spam" in [1, 2, "spam", 4]  
True  
>>> "a" in "spam"  
True  
>>> "f" not in "eggs"  
True
```

Special type of Nothing

- where perl has undefined, python often uses None
- many functions may return this
- normally, we don't use `is` or `is not` test
- one common idiom however:

```
data = some_function()
if data is not None:
    play_with(data)
else:
    print "Shut 'er down, Clancy!"
```

While

- works as expected, with one additional surprise
- optional `else` clause for when the condition is false

```
>>> f = 3
>>> while f > 0:
...     print f,
...     f -= 1      # same as: f = f - 1
... else:
...     print "done"
...
3 2 1 done
>>>
```

- Loop forever: `while True:`

Looping over Sequences

- again, works as expected, with the addition of an optional else clause

```
>>> for item in [3, 2, 1]:  
...     print item,  
...     else:  
...         print "done"  
...  
3 2 1 done
```

- no separate for and foreach syntax

Looping over Integers

- the `range` function produces an integer sequence
- technically, produces an iterator object, which is memory efficient — `range(100000)` not waste of memory
- `range([start,] stop[, step])`

```
>>> range(4)
[0, 1, 2, 3]
>>> range(4, 0, -1)
[4, 3, 2, 1]
```

Loops: Cutting out early

- both `for` and `while` loops may be exited early
- `continue` causes the rest of the block to be skipped, and the loop to go on to the next stage
- `break` cause control to jump out of the loop, **skipping** the `else` block

```
>>> for item in "spam":
...     if item == "a": break      # one-line if statement
...     print item,
...
s p
>>> for item in "spam":
...     if item == "a": continue
...     print item,
...
s p m
```

Functions

- defined with `def`
- without a `return` statement, the function returns `None`
- function arguments may be given default values

```
>>> def mult(x, y=1):  
...     return x * y  
...  
>>> mult(5, 3)  
15  
>>> mult(5)  
5  
>>>
```

Documenting Functions

- functions can (should) have doc strings
- by convention, triple quoted, may cross several lines
- first thing after def line

```
def frobnify(x, y, monkey):  
    """frobnify(x, y, monkey) -> list.  Cast monkey into 2D hilbert-space.  
  
    If monkey is less than zero, throws InfiniteMonkeys exception.  
    """  
    l2 = frobnosticate(x, monkey) ** dogma_dance(y, monkey)  
    ...
```

Input and Output: Terminal

- `input` reads from terminal but **evaluates** input
- `raw_input` does not evaluate input

```
>>> spam = input()
eggs
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "<string>", line 0, in ?
NameError: name 'eggs' is not defined
>>> spam = raw_input()
eggs
>>> spam
'eggs'
>>> viking = input("How many Vikings? ")
How many Vikings? 33
>>> viking
33
```

Input and Output: Files

- like unix: `open` takes file name and mode
- mode: `r`, `w`, `a` for read, write or append
- `open` returns a file object, with several methods
- `readline()` reads a single line (up to newline)
- `readlines()` for iterating over every line

- `write(str)` write the string to the file
- `writelines(seq)` write sequence of strings
- neither `write` form adds newlines
- `close()` to close file
- `flush()` to force buffered data into file

Silly Function Example: Counting The

```
def count_the(filename):
    """count_the(filename) -> integer.  Count word "the" in a file."""
    thes = 0
    f = open(filename, "r")
    for line in f.readlines():
        words = line.split()
        for word in words:
            # Not very smart: should clean up punctuation, or better,
            # this should be turned into a regular expression test.
            if word.lower() == "the":
                thes += 1
    f.close()
    return thes
```

Python's Libraries

- called “modules” in Python lingo
- many are hierarchical
- there are modules to do nearly everything
- modules have their own documentation trail
- many third-party tools

Import: Safest Version

- `import modulename` is safest to your namespace
- the module functions will be called `modulename.function()`
- requires a lot of typing
- good for infrequently called functions

Import: Into the Global Namespace

- when you define a function in your program, then function name is in the global namespace
- using `from MODULE import FUN1, FUN2, FUN3, ...` you import module functions into the global namespace
- there is a danger of name-clash
- but for functions you use a lot, it saves a lot of typing

Useful module: `sys`

- normally just use `import sys`
- useful function is `sys.exit(error_code)`, exits the program immediately (say, for a nonrecoverable error)
- the `input` and `raw_input` functions are crude
- full file methods available for `sys.stdin`, `sys.stdout` and `sys.stderr`

```
for line in sys.stdin.readlines():  
    do_something_with(line)
```

When things go bad: Exceptions

- when python encounters a fatal error, it “raises an exception”
- there are many sorts of exception
- exceptions are just classes — you can write your own
- there are control structures to watch for and clean up after exceptions

Exception Handling

- dangerous code goes into try block
- if anything in that fails, matching except clause run

```
try:  
    f = open(filename, "r")  
except:  
    sys.stdout.write("Can't open %s!\n" % filename)  
    sys.exit(1)
```

- you can check for particular exceptions
- the catch-all, default except clause must be last

```
try:
    f = open(filename, "r")
except IOError:
    sys.stdout.write("IO: can't open %s!\n" % filename)
    sys.exit(1)
except:
    sys.stdout.write("Mystery error!\n")
    sys.exit(2)
```

Many Exceptions

- you can extract more information
- exceptions have a class hierarchy, so an exception might have several matches
- e.g. any `OverflowError`, `ZeroDivisionError` or `FloatingPointError` is also an `ArithmeticError`
- I'll introduce more of these as we move along

A Taste of Class

- no real object orientation this week
- an object in python is just another namespace
- as with modules, you grab object elements with the dot notation, `f.close()`
- these are technically called “properties”
- you can create a dummy class just to hold properties

Class like C struct

```
# When a block is required, like in class, use
# 'pass' - a no-operation command - as a placeholder. Useful in loops
# and conditionals during development, too
>>> class info: pass
...
>>> m = info()
>>> spam = info()
>>> spam.name = "Michael Palin"
>>> spam.name
'Michael Palin'
>>> spam.goo
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
AttributeError: info instance has no attribute 'goo'
```

