

# Python Tutorial

Day 1

## Why Python

- high level language
- interpreted and interactive
- real data structures (structures, objects)
- object oriented all the way down
- rich library support

## The First Program

```
#!/usr/bin/env python2.4
```

```
print "Hello Kitty"      # comments like shell
```

- as usual `chmod +x FILENAME`

## Interactive Mode

- like shell, but unlike perl, python can be interactive

```
$ python
Python 2.4.1 (#1, Sep 21 2005, 14:17:33)
[GCC 4.0.0 20041026 (Apple Computer, Inc. build 4061)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

- built in debugger (will cover later)
- can preload code, then test: `python -i FILE.py`
- some platforms have line editing support (readline)

## Assignment

- variable names as in shell: foo bar\_3 ice9
- only letters, digits (cannot start) and underscore
- unlike shell or perl, no sign to reference:

```
>>> spam = 5
>>> spam * 3
15
```

## Types: Integers

- usual math operators: + - \* / \*\* %

```
>>> 7 / 2          # integer division returns floor
3
>>> 7 / -2
-4
```

- can force whole answer in division: //

```
>>> 2.3 / 2
1.1499999999999999
>>> 2.3 // 2
1.0
```

## Types: Integer Base

- usual Unix base conventions
- leading zero is octal (base eight):

```
>>> 033
27
>>> print oct(27)
033
```

- leading 0x is hex (base 16):

```
>>> 0x99
153
>>> print hex(153)
0x99
```

## Types: Floating Point, Complex

- usual Unix/C exponent notation:  $2.4e1 = 24.0$
- mixed types coerced up

```
>>> (3 + 4) / 2
3
>>> (3 + 4) / 2.0
3.5
>>> (3 + 4) / 2.0 + 2j
(3.5+2j)
```

- complex numbers created with function

```
>>> complex(2 ,1)
(2+1j)
```

## Types: more Complex

- complex literals also possible
- use properties `real` and `imag` to decompose:

```
>>> f = (2+1j) * (3+2j)
>>> f.real
4.0
>>> f.imag
7.0
```

## Types: Sequences

- all sequences take similar operations
- element reference: `s[1]`
- slice: `s[3:5]`
- length function: `len(s)`
- iteration: `for elt in s: print elt`

- finding element: `s.index('item')`
- testing membership: `if 'eggs' in s:`

## Sequences: Lists (arrays)

- list literal with square brackets: `[2, 3, 5]`
- any type can be an element: `["a", 2, 3.14]`
- append: `s.append("spam")`
- change item: `s[3] = "eggs"`
- empty list: `f = []`

## Sequences: Tuple

- arrays are mutable: you can change elements
- tuples are immutable
- (immutable sequences take less memory)
- library functions often return tuples
- access functions work the same
- tuple literal: `f = (2, 3, 66)`

## Sequences: String

- strings are another immutable sequence
- no difference between single and double quote, 'a', "a", "How's it going?"
- multi-line strings in triple quotes, '''wow''', """wow"""
- normal Unix/C string escapes work as expected

```
>>> print "a\tb\tc\n"  
a      b      c  
  
>>>
```

## Cutting up Strings

```
>>> s = "This is a test."
>>> s[0]
'T'
>>> s[-1]
'.'
>>> s[4:8]
' is '
>>> s[:4]
'This'
>>> s[-4:]
'est.'
>>> s.split()
['This', 'is', 'a', 'test.']
>>> '      moo      '.strip()
'moo'
```

## OOPy Digression

- all types in Python are object oriented
- methods are simply functions that apply to an object
- notation is like C/C++: a dot separating object and method, `s.split()`
- all types have methods; not all literals can call methods

- Python is introspective: you can ask types what they can do with function `dir`

```
>>> f = "spam"
>>> dir(f)
['__add__', '__class__', '__contains__', '__delattr__', '__doc__',
 '__eq__', '__ge__', '__getattr__', '__getitem__',
 '__getnewargs__', '__getslice__', '__gt__', '__hash__', '__init__',
 '__le__', '__len__', '__lt__', '__mod__', '__mul__', '__ne__',
 '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__rmod__',
 '__rmul__', '__setattr__', '__str__', 'capitalize', 'center',
 'count', 'decode', 'encode', 'endswith', 'expandtabs', 'find',
 'index', 'isalnum', 'isalpha', 'isdigit', 'islower', 'isspace',
 'istitle', 'isupper', 'join', 'ljust', 'lower', 'lstrip', 'replace',
 'rfind', 'rindex', 'rjust', 'rstrip', 'split', 'splitlines',
 'startswith', 'strip', 'swapcase', 'title', 'translate', 'upper',
 'zfill']
>>>
```

## Doc strings

- Functions and methods can have documentation strings
- Use the attribute `__doc__`
- returns a string, so use `print` to look at it

```
>>>  
>>> print f.strip.__doc__  
S.strip([chars]) -> string or unicode
```

Return a copy of the string `S` with leading and trailing whitespace removed.

If `chars` is given and not `None`, remove characters in `chars` instead. If `chars` is unicode, `S` will be converted to unicode before stripping

```
>>>
```

## Strings: Formatting

- like C `printf` format strings
- marked with `%` operator
- simplest format is `%s`, which converts all types to a string
- `string % [item | tuple ]`

```
>>> (a, b, c) = (1, 2, 3) # note tuple assignment
>>> "%s + %s = %s" % (a, b, a + b)
'1 + 2 = 3'
>>> print """ %s
... %s
... + %s
... ----
... %s""" % (a, b, c, a + b + c)
1
2
+ 3
----
6
>>> print "%s" % 'w00t'
w00t
>>>
```

## Types: Collections

- As of Python 2.4 there are two types of collections
- collections are unordered
- dictionary: a hash table
- set: a collection of unique items (2.4)

## Collections: Dictionaries

- literal: `{key:item, ...}`, empty `{}`
- access and set like a list: `d['name'] = 'Palin'`
- method `items` returns list of (key, item) tuples
- method `keys` returns list of only keys
- method `values` returns list of only values
- method `get` retrieves value, with default

```
>>> d = {'a':1, 'b':3, 'pi':3.14159}
>>> d['c'] = 42
>>> d
{'a': 1, 'pi': 3.1415899999999999, 'c': 42, 'b': 3}
>>> d['f']
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
KeyError: 'f'
>>> d.get('f', 22)
22
>>> d.keys()
['a', 'pi', 'c', 'b']
>>> d.items()
[('a', 1), ('pi', 3.1415899999999999), ('c', 42), ('b', 3)]
>>> "pi is %(pi)s" % d # note "%(KEY)s" format
'pi is 3.14159'
>>>
```

## Collections: Sets

- new in Python 2.4
- an unordered collection of unique items
- may not contain mutable items (no lists or dicts)
- requires function to create: `f = set()`
- set can take a single argument, any sequence type

- method `add` adds elements
- union: `s1 | s2`; intersection `s1 & s2`

```
>>> f = set()
>>> f
set([])
>>> f.add('ll')
>>> g = set("abcde")
>>> g
set(['a', 'c', 'b', 'e', 'd'])
>>> g = set("abcdeab")
>>> g
set(['a', 'c', 'b', 'e', 'd'])
>>> f | g
set(['a', 'c', 'b', 'e', 'd', 'll'])
>>>
```

## Type Conversion

- base types have conversion functions
- `int`, `float`, `complex`, `long`, `str`, `list`, `tuple`, `dict`, `set`
- most work as expected
- `dict` expects a sequence of sequences

- remove duplicate entries (order lost):

```
>>> f = [1, 2, 5, 2, 4, 1, 3, 6, 2, 6, 4, 2, 3]
>>> f = list(set(f))
>>> f
[1, 2, 3, 4, 5, 6]
```

- unlike perl, string to number conversion not automatic

```
>>> str(int("4") + float("3.14159e2"))
'318.159'
>>>
```

## Next Week

- control structures
- functions
- details on type methods
- basic exception handling