

## Introduction into Python

Daniel Polani

### Properties:

- minimalistic syntax
- powerful
- high-level data structures built in
- scripting, rapid applications, and — as we will see — AI
- widely in use (a plus for you)

### What we will mainly need:

- flexible data and algorithm structures
- functionality
- procedurality

## Where to get Python and Learn More?

**Web Site:** <http://www.python.org/> (also main source for this introduction)

**UH Availability:** in PC labs A,B,C

## Introduction: Remarks

### Note:

- you will be taught the essentials for it to be useful and to solve the problems posed later
- intro is not meant to be complete
- read for yourselves if you want to learn more (there is no way around it)

- To start: type `python` in command line
- it will look like

```
Python 2.4 (#1, Mar  8 2005, 19:08:08)
[GCC egcs-2.91.66 19990314/Linux (egcs-1.1.2 release)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

- you can now type commands in the line denoted by `>>>`
- To leave: type end-of-file character (`ctrl-D` on Unix, `ctrl-z` on Windows)
- We call this: *interactive mode*

```
>>> the_world_is_flat = True
>>> if the_world_is_flat:
...     print "Be careful not to fall off!"
...
Be careful not to fall off!
```

## Lessons:

- prompt `>>>` allows to enter command
- command is ended by newline
- variables (`the_world_is_flat`) need not be initialized or declared
- a colon ":" opens a *block*
- `...` prompt denotes that block is expected
- a block is indented
- by ending indentation, block is ended

# Notation

## Notation:

- `>>>` prompt means, your input
- `...` prompt means, indentation (block) expected or possible
- no prompt means, python output

# Differences to Java or C

- interactive work possible
- no declaration of variables
- dynamic typing
- no brackets denote block, just indentation
  - this needs getting used to
  - a good editor (e.g. emacs) supports the style

**Remark:** a comment begins with a #. Everything after that is ignored

```
# this is a comment
foobar = 1 # this is setting 'foobar' to 1, followed by a comment
mystring = "# not a comment, just a hash"
```

```
>>> 2 + 2
4
>>> (50 - 5*6)/4
5
>>> 7/3 # integer division returns floor
2
>>> 7/-3
-3
>>> 7.0 / 2 # floating point is recognized dynamically
3.5
>>> width = 20
>>> height = 5 * 9
>>> width * height
900
>>> x = y = z = 0
```

## Strings

```
>>> 'spam eggs'
'spam eggs'
>>> 'doesn\'t'
"doesn't"
>>> "doesn't"
"doesn't"
>>> '"Yes," he said.'
'"Yes," he said.'
>>> """Here you can do
... everything"""
'Here you can do\neverything'
```

## Strings II

```
>>> 'a' + 'b'
'ab'
>>> 'and again ' * 5
'and again and again and again and again and
>>> '   lost all spaces   '.strip()
'lost all spaces'
```

## Strings III

```
>>> mystring = 'abcd'
>>> mystring[1]
'b'
>>> mystring[0:2]
'ab'
>>> mystring[2:4]
'cd'
```

## Strings IV

```
>>> mystring[4]
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
IndexError: string index out of range
>>> mystring[:2]
'ab'
>>> mystring[2:]
'cd'
```

## Strings V

**Note:** individual characters in a Python string cannot be changed

```
>>> mystring[0] = 'z'
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: object does not support item assignment
```

**Instead use:**

```
>>> 'z' + mystring[1:]
'zbcd'
```

## Strings VI

```
>>> mystring[2:1]
''
>>> mystring[-1]
'd'
>>> mystring[-2]
'c'
>>> mystring[-2:]
'cd'
>>> mystring[:-2]
'ab'
>>> s = 'supercalifragilisticexpialidocious'
>>> len(s)
34
>>> len(mystring[1:3])
2
```

## Lists

```
>>> mylist = ['foo', 'bar', 3, 1.5]
>>> mylist
['foo', 'bar', 3, 1.5]
>>> mylist[1] # counting starts from 0
'bar'
>>> mylist[-1] # negative index from end of list
1.5
>>> mylist[1:-1] # sublist from second element to second last
['bar', 3]
>>> mylist[:2] + ['bingo', 2*2] # concatenate and evaluate
['foo', 'bar', 'bingo', 4]
>>> 3 * mylist[:2] # list with 3 copies of mylist
['foo', 'bar', 'foo', 'bar', 'foo', 'bar']
```

## Lists II

```
>>> mylist[2] += 2 # it is possible to change elements
>>> mylist
['foo', 'bar', 5, 1.5]
>>> mylist[:2] = ['foobar'] # or even to replace sublists
>>> mylist
['foobar', 5, 1.5]
>>> mylist[1:1] = ['foo', 'bar'] # or to insert
>>> mylist
['foobar', 'foo', 'bar', 5, 1.5]
>>> len(mylist) # len works also for lists
5
```

## Lists III

```
>>> innerlist = ['in', 'ner']
>>> mylist[3] = innerlist # one can nest lists
>>> mylist
['foobar', 'foo', 'bar', ['in', 'ner'], 1.5]
>>> mylist[3]
['in', 'ner']
>>> mylist[3].append('list') # and operate on the inner lists
>>> mylist
['foobar', 'foo', 'bar', ['in', 'ner', 'list'], 1.5]
>>> innerlist # mylist[3] and innerlist are the same object
['in', 'ner', 'list']
```

## Programming

```
>>> # Fibonacci series:
... # the sum of two elements defines the next
... a, b = 0, 1
>>> while b < 10:
...     print b
...     a, b = b, a+b
... 
```

## Features:

1. multiple assignment: rhs evaluated before anything on the left, and (in rhs) from left to right
2. while loop executes as long as condition is True (non-zero, not the empty string, not None)
3. block indentation must be the same for each line of block
4. need empty line in *interactive* mode to indicate end of block (not required in edited code)
5. use of print

```
>>> i = 256*256
>>> print 'The value of i is', i
The value of i is 65536
>>> a, b = 0, 1
>>> while b < 1000:
...     print b,          # comma prevents newline
...     a, b = b, a+b
...
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
```

# Flow Control

```
>>> x = int(raw_input("Please enter an integer: "))
>>> if x < 0:
...     x = 0
...     print 'Negative changed to zero'
... elif x == 0:
...     print 'Zero'
... elif x == 1:
...     print 'Single'
... else:
...     print 'More'
...
...
```

**Note:** use `elif` instead of `else if` to avoid indentation

**Note:** use `as` as a switch statement

# Iteration

**Remark:** Python `for` iterates over sequence (string, list, tuple, range, etc.)  
generated sequence)

```
>>> # Measure some strings:
... a = ['cat', 'window', 'defenestrate']
>>> for x in a:
...     print x, len(x)
...
cat 3
window 6
defenestrate 12
```

**Note:** It is not safe to modify sequence while iterating over it. If necessary, create a copy:

```
>>> for x in a[:]: # make a slice copy of the entire list
...     if len(x) > 6: a.insert(0, x)
...
>>> a
['defenestrate', 'cat', 'window', 'defenestrate']
```

```
>>> range(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> range(5, 10)
[5, 6, 7, 8, 9]
>>> range(0, 10, 3)
[0, 3, 6, 9]
>>> mylist
['foobar', 'foo', 'bar', ['in', 'ner', 'list'], 1.5]
>>> for i in range(len(mylist)): print i, mylist[i]
...
0 foobar
1 foo
2 bar
3 ['in', 'ner', 'list']
4 1.5
```

## Further Flow Control

## The pass Statement

- `break` ends the innermost `for` or `while` loop
- `continue` continues the next iteration of the loop, ignoring the remaining rest of the present iteration
- `else` as second clause of an iteration is performed when the loop ends regularly (not by a `break`)

```
for n in range(2, 10): # notation as in program
    for x in range(2, n):
        if n % x == 0:
            print n, 'equals', x, '*', n/x
            break
    else:
        # loop fell through without finding a factor
        print n, 'is a prime number'
```

```
while True:
    pass # the block must not be empty,
        # but if we do not want
        # to do anything, do pass
```

```
def fib(n):    # write Fibonacci series up to n
    """Print a Fibonacci series up to n.""" # help string
    a, b = 0, 1
    while b < n:
        print b,
        a, b = b, a+b

>>> # Now call the function we just defined:
... fib(2000)
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597
```

## Function:

- definition by `def`
- name and argument list
- optional documentation string
- statements must be indented with respect to `def`

## Properties:

- all variables are local, unless named `global` or `system` variables
- always a *call by reference*

# Functions are Objects

```
>>> fib
<function fib at 10042ed0>
>>> f = fib
>>> f(100)
1 1 2 3 5 8 13 21 34 55 89
```

**Note:** functions are objects and can be assigned to variables

**Note:** `fib` is a procedure, i.e. a function that returns `None`.

**Return Value:** to return a value from a function, use the keyword `return`

# Example

```
>>> def fib2(n): # return Fibonacci series up to n
...     """Return a list containing the Fibonacci series up to n."""
...     result = []
...     a, b = 0, 1
...     while b < n:
...         result.append(b)    # see below
...         a, b = b, a+b
...     return result
...
>>> f100 = fib2(100)    # call it
>>> f100                # write the result
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
```

**Note:** `append` is a list method. It applies to the previous object.

### Example:

```
def ask_ok(prompt, retries=4, complaint='Yes or no, please!'):
    while True:
        ok = raw_input(prompt)
        if ok in ('y', 'ye', 'yes'): return True
        if ok in ('n', 'no', 'nop', 'nope'): return False
        retries = retries - 1
        if retries < 0: raise IOError, 'refusenik user'
        print complaint
```

### Use e.g. as:

```
ask_ok('Do you really want to quit?') or
ask_ok("I am impatient, give me a quick repl:
```

### Note:

- `in` is used outside of an iteration, as predicate identifying whether an object is in a list
- `raise` to raise exceptions

### Note:

- call first the obligatory arguments, then the optional (even if optionals are named)
- only one value assignment per argument

## Arbitrary Argument Lists

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

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

## Unpacking Argument Lists

```
>>> range(3, 6) # normal call with separate arguments
[3, 4, 5]
>>> args = [3, 6]
>>> range(*args) # call with arguments unpacked
[3, 4, 5]
```

### Create Anonymous (Lambda) Functions:

```
>>> def make_incrementor(n):
...     return lambda x: x + n
...
>>> f = make_incrementor(42)
>>> f(0)
42
>>> f(1)
43
```

**Note:** `help(list)` will give you a list of possible operations on the `list` structure, e.g.

- `append(x)` appends an item to the list `a` (same as `a[len(a):] = [x]`)
- `extend(L)` is equivalent to `a[len(a):] = L` (what exactly does it do?)
- `insert(i, x)` inserts `x` in front of the `i`-th member of the list
- `remove(x)` removes the first occurrence of `x` from the list (what happens if `x` is not in list?)

## More List Functions II

- `pop([i])` removes the item at position `i` and returns it
- `index(x)` returns the index of the first item with value `x`
- `count(x)`: how often does `x` appear in list?
- `sort(cmp=None, key=None, reverse=False)` sorts the elements of the list, in-place.  
`cmp(x, y)` should return `-1`, `0` or `1`, depending on whether `x < y`, `x = y` or `x > y`. You can replace `cmp` by your own function (or lambda expression).
- `reverse()` reverses list, in-place

## Lists as Stacks

```
>>> stack = [3, 4, 5]
>>> stack.append(6)
>>> stack.append(7)
>>> stack
[3, 4, 5, 6, 7]
>>> stack.pop()
7
>>> stack
[3, 4, 5, 6]
>>> stack.pop()
6
>>> stack.pop()
5
>>> stack
[3, 4]
```

## Lists as Queues

```
>>> queue = ["Eric", "John", "Michael"]
>>> queue.append("Terry")           # Terry arrived
>>> queue.append("Graham")        # Graham arrived
>>> queue.pop(0)
'Eric'
>>> queue.pop(0)
'John'
>>> queue
['Michael', 'Terry', 'Graham']
```

**Note:** *yes, it's Monty Python, not the snake after which the language's named!*

## Filtering Tools

```
def even(x):
    return x % 2 == 0

>>> filter(even, range(0, 10))
[0, 2, 4, 6, 8]

def cube(x): return x * x * x

>>> map(cube, range(0, 10))
[0, 1, 8, 27, 64, 125, 216, 343, 512, 729]
```

## List Comprehensions

**Allows:** elegant construction of lists

```
vec = range(10)

print vec
print [2*x for x in vec]

prints

[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
```

## List Comprehensions with Conditions

**Conditions:** use if

```
print [[x, x * x] for x in vec]
print [[x, x * x] for x in vec if x % 2 == 0]

to obtain

[[0, 0], [1, 1], [2, 4], [3, 9], [4, 16], [5, 25], [6, 36],
 [7, 49], [8, 64], [9, 81]]

[[1, 1], [3, 9], [5, 25], [7, 49], [9, 81]]
```

**Note:** almost like lists, but fixed structure. Items separated by list

```
>>> t = 12345, 54321, 'hello!'
>>> t[0]
12345
>>> t
(12345, 54321, 'hello!')
>>> # Tuples may be nested:
... u = t, (1, 2, 3, 4, 5)
>>> u
((12345, 54321, 'hello!'), (1, 2, 3, 4, 5))
>>> empty = ()
>>> singleton = 'hello', # <-- note trailing comma
>>> singleton
('hello',)
```

**Note:** this ugly syntax for singletons cannot be avoided

**Operator:** del removes a member from a list

```
>>> a = [1,2,3,4,5]
>>> del a[0]
>>> a
[2,3,4,5]
>>> del a[1:3]
>>> a
[2,5]
```

and can be used to delete a full variable: del a

**Def.:** a set is an unordered collection without duplicates.

```
>>> basket = ['apple', 'orange', 'apple', 'pear', 'orange', 'banana']
>>> fruits = set(basket) # create a set without duplicates
>>> fruits
set(['orange', 'pear', 'apple', 'banana'])
>>> 'orange' in fruits # fast membership testing
True
>>> 'crabgrass' in fruits
False
```

```
>>> a = set('abracadabra')
>>> b = set('alacazam')
>>> a # unique letters in a
set(['a', 'r', 'b', 'c', 'd'])
>>> a - b # letters in a but not in b
set(['r', 'd', 'b'])
>>> a | b # letters in either a or b
set(['a', 'c', 'r', 'd', 'b', 'm', 'z', 'l'])
>>> a & b # letters in both a and b
set(['a', 'c'])
>>> a ^ b # letters in a or b but not
set(['r', 'd', 'b', 'm', 'z', 'l'])
```

**Def.:** dictionaries, also *maps*, *associative arrays* map (almost) arbitrary objects to other objects: like arrays with not-necessarily numeric indices

**Note:** in Python, indices can only be immutable objects: e.g. strings, numbers or immutable tuples (i.e. tuples that recursively contain only immutable objects) — no lists

**Concept:** unordered set of key:value pairs

**Empty Dict:** {}

**Use:** like an array/list

**Further Functions:** `keys()` returns a list of dict keys.  
`has_key()` checks whether a key is in the dictionary.

```
>>> tel = {'jack': 4098, 'sape': 4139}
>>> tel['guido'] = 4127
>>> tel
{'sape': 4139, 'guido': 4127, 'jack': 4098}
>>> tel['jack']
4098
>>> del tel['sape']
>>> tel['irv'] = 4127
>>> tel
{'guido': 4127, 'irv': 4127, 'jack': 4098}
>>> tel.keys()
['guido', 'irv', 'jack']
>>> tel.has_key('guido')
True
```

## Dict Constructor

```
>>> dict([('sape', 4139), ('guido', 4127), ('jack', 4098)])
{'sape': 4139, 'jack': 4098, 'guido': 4127}
>>> dict([(x, x**2) for x in vec])      # use a list comprehension
{2: 4, 4: 16, 6: 36}
```

## Looping in Dicts

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

## Enumerating Iteration

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

## Combining Iterations Over Two Sequences

```
>>> 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

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

>>> print my_function.__doc__
Do nothing, but document it.
```

**Construction:** write program in file `fibonacci.py`

```
# 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 fibonacci` in the main module. This defines the module name. To call the functions, you need to qualify: `fibonacci.fib2(10)`

**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__)`

## Files — I/O

**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", 1.8))
```

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

**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
```

**Task:** check out the methods for strings

## Iterators and yield