Introduction to Python

we follow here:

https://docs.python.org/2/tutorial/

for alternatives, see:

1) https://wiki.python.org/moin/BeginnersGuide/Programmers

2) http://nbviewer.ipython.org/github/jrjohansson/scientific-python-lectures/tree/master/

why Python?

- to search-and-replace over a large number of files in a complicated way
- test suite for a library
- simple to use, available on Windows, Unix, OS X
- real programming language
- high-level, concise programming
- modular
- interpreter (has pros and cons)
- extensible
- named after the BBCs „Monty Python's Flying Circus“ series (no British humour intended)

Python interpreter

Look at:

```
th-ea-lswtb02:~ Lode.Pollet$ python
Python 2.7.8 (default, Jul 13 2014, 17:11:32)
[GCC 4.2.1 Compatible Apple LLVM 5.1 (clang-503.0.40)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> help
Type help() for interactive help, or help(object) for help about object.
>>> the_world_is_flat = 1
>>> if the_world_is_flat:
...     print "Be careful not to fall off!“ # note the indentation
...
Be careful not to fall off!
>>> exit()
```

it’s also a pocket calculator:

(here I used ipython instead of python, also note the comment syntax and how to recall the last variable)

```
In [6]: 2+4
Out[6]: 6

In [7]: 5/3
Out[7]: 1

In [8]: 5/3.   # so mixed int-float types automatically convert to float
Out[8]: 1.6666666666666667

In [9]: 5 % 3
Out[9]: 2

In [10]: 2**4
Out[10]: 16

In [11]: 2**4.
Out[11]: 16.0
```

```
In [15]: 5 // 3.0
Out[15]: 1.0

In [16]: 2 + _
Out[16]: 3.0
```

Note that Unicode can also be used — see the documentation

Python strings

Python strings are so-called *sequence* types

you can use single or double quotes, and use *print* for nicer syntax as can be seen from the examples below from the website:

```
>>> 'spam eggs'  # single quotes
'spam eggs'
>>> 'doesn\'t'  # use \' to escape the single quote...
"doesn't"
>>> "doesn't"  # ...or use double quotes instead
"doesn't"
>>> '"Yes," he said.'
'"Yes," he said.'
>>> "\"Yes,\" he said."
'"Yes," he said.'
>>> '"Isn\'t," she said.'
'"Isn\'t," she said.'
```

```
>>> print 'C:\some\name'  # here \n means newline!
C:\some
ame
>>> print r'C:\some\name'  # note the r before the quote
C:\some\name
```

(a *raw* string)

```
print """\
Usage: thingy [OPTIONS]
     -h                        Display this usage message
     -H hostname               Hostname to connect to
"""
```

(for including end-of-lines)

```
>>> '"Isn\'t," she said.'
'"Isn\'t," she said.'
>>> print '"Isn\'t," she said.'
"Isn't," she said.
>>> s = 'First line.\nSecond line.'  # \n means newline
>>> s  # without print(), \n is included in the output
'First line.\nSecond line.'
>>> print s  # with print, \n produces a new line
First line.
Second line.
```

concatenation:

```
>>> # 3 times 'un', followed by 'ium'
>>> 3 * 'un' + 'ium'
'unununium'
```

```
>>> prefix = 'Py'
>>> prefix + 'thon'
'Python'
```

```
>>> text = ('Put several strings within parentheses '
            'to have them joined together.')
>>> text
'Put several strings within parentheses to have them joined together.'
```

Python strings

Indexing is as follows (note that the counting starts by 0)

```
>>> word = 'Python'
>>> word[0]  # character in position 0
'P'
>>> word[5]  # character in position 5
'n'
```

Perhaps more surprising is that the following is also acceptable:

```
>>> word[-1]  # last character
'n'
>>> word[-2]  # second-last character
'o'
>>> word[-6]
'P'
```

```
 +---+---+---+---+---+---+
 | P | y | t | h | o | n |
 +---+---+---+---+---+---+
 0   1   2   3   4   5   6
-6  -5  -4  -3  -2  -1
```

Slicing is also supported:

```
>>> word[0:2]  # characters from position 0 (included) to 2 (excluded)
'Py'
>>> word[2:5]  # characters from position 2 (included) to 5 (excluded)
'tho'
```

```
>>> word[:2] + word[2:]
'Python'
>>> word[:4] + word[4:]
'Python'
```

```
>>> word[-2:] # characters from the second-last (included) to the end
'on'
```

Python strings

Python strings are however *immutable*

```
>>> word[0] = 'J'
  ...
TypeError: 'str' object does not support item assignment
>>> word[2:] = 'py'
  ...
TypeError: 'str' object does not support item assignment
```

if you want to change it, a new one needs to be created:

```
>>> 'J' + word[1:]
'Jython'
>>> word[:2] + 'py'
'Pypy'
```

len(.) returns the length of a string:

```
>>> s = 'supercalifragilisticexpialidocious'
>>> len(s)
34
```

```
In [17]: word = 'Python'

In [18]: len(word)
Out[18]: 6
```

Python lists

The *list* is a *compound* data type, written as a list of comma-separated values (items) between square brackets. Lists may contain items of different types

```
>>> squares = [1, 4, 9, 16, 25]
>>> squares
[1, 4, 9, 16, 25]
```

Like all other *sequence* types they can be indexed and sliced:

```
>>> squares[0]  # indexing returns the item
1
>>> squares[-1]
25
>>> squares[-3:]  # slicing returns a new list
[9, 16, 25]
```

This provides a shallow copy of the list:

```
>>> squares[:]
[1, 4, 9, 16, 25]
```

concatenation is also possible:

```
>>> squares + [36, 49, 64, 81, 100]
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

unlike strings, lists are *mutable*:

```
>>> cubes = [1, 8, 27, 65, 125]  # something's wrong here
>>> 4 ** 3  # the cube of 4 is 64, not 65!
64
>>> cubes[3] = 64  # replace the wrong value
>>> cubes
[1, 8, 27, 64, 125]
```

Python lists

Values at the end can be added by *append(.):*

```
>>> cubes.append(216)  # add the cube of 6
>>> cubes.append(7 ** 3)  # and the cube of 7
>>> cubes
[1, 8, 27, 64, 125, 216, 343]
```

(more about methods later)

assignment to slices is possible, and can change the size of the list:

```
>>> letters = ['a', 'b', 'c', 'd', 'e', 'f', 'g']
>>> letters
['a', 'b', 'c', 'd', 'e', 'f', 'g']
>>> # replace some values
>>> letters[2:5] = ['C', 'D', 'E']
>>> letters
['a', 'b', 'C', 'D', 'E', 'f', 'g']
>>> # now remove them
>>> letters[2:5] = []
>>> letters
['a', 'b', 'f', 'g']
>>> # clear the list by replacing all the elements with an empty list
>>> letters[:] = []
>>> letters
[]
```

the function len(.) also applies

```
>>> letters = ['a', 'b', 'c', 'd']
>>> len(letters)
4
```

Python lists

nested lists can also be created:

```
>>> a = ['a', 'b', 'c']
>>> n = [1, 2, 3]
>>> x = [a, n]
>>> x
[['a', 'b', 'c'], [1, 2, 3]]
>>> x[0]
['a', 'b', 'c']
>>> x[0][1]
'b'
```

(note the indexing)

A little quiz

What is the output of the following program:

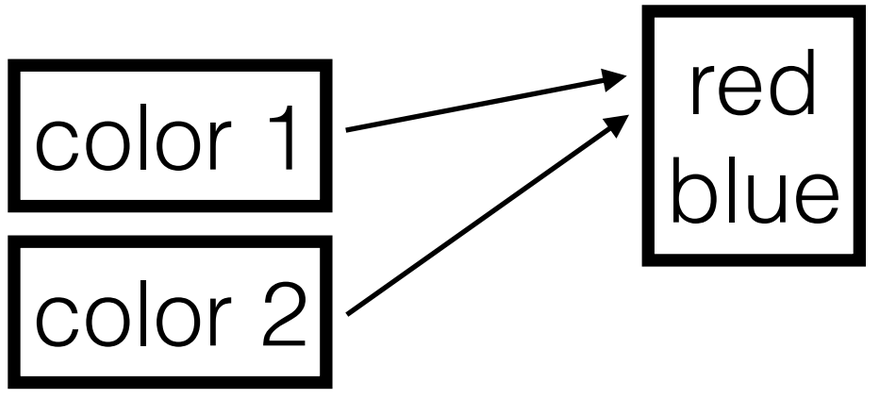
```
>>> color1 = ["red", "blue"]
>>> color2 = color1
>>> print color1
['red', 'blue']
>>> print color2
['red', 'blue']
>>> color2[1] = "green"
>>> print color2
['red', 'green']
>>> print color1
```

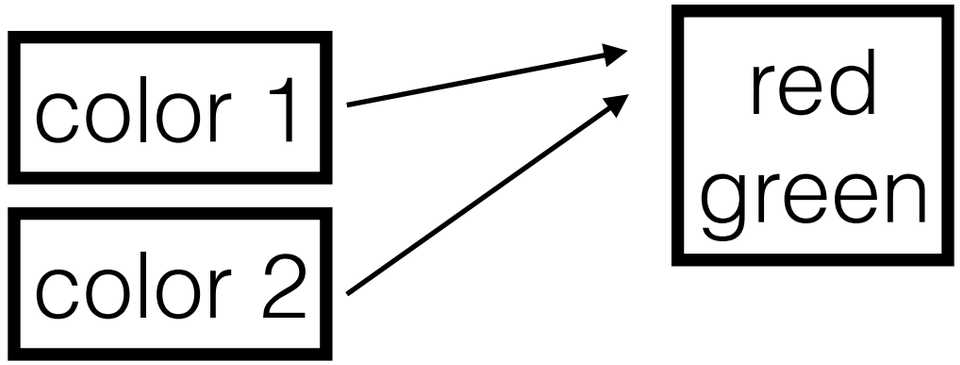
A. [‘red’, ‘blue’]
B. [‘red’, ‘green’]
C. gives an error
D. none of the above

Vote now!

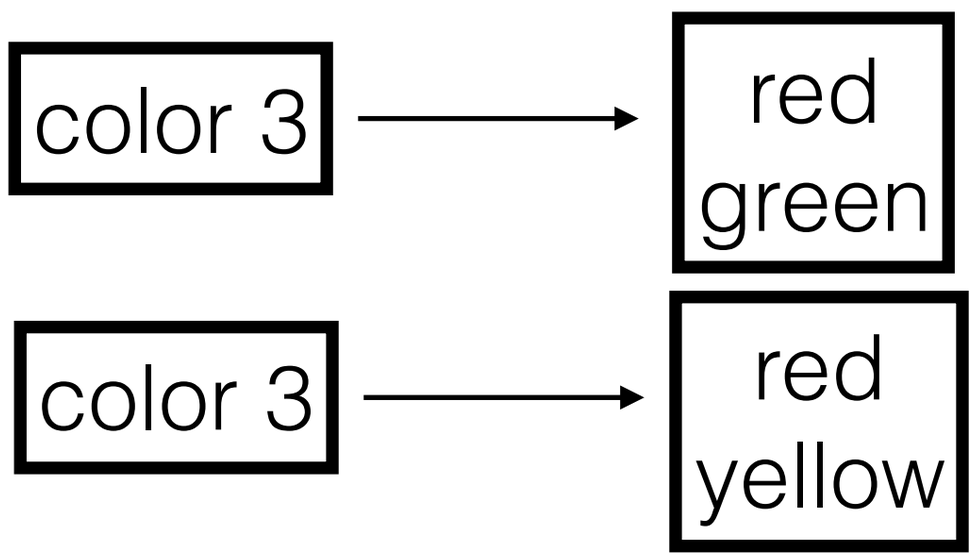
A little quiz

The reason is what is called a “shallow” copy





a “deep” copy could be achieved as follows:




```
>>> from copy import deepcopy
>>> color3 = deepcopy(color1)
>>> print color1
['red', 'green']
>>> print color3
['red', 'green']
>>> color3[1] = 'yellow'
>>> print color3
['red', 'yellow']
>>> print color1
['red', 'green']
>>> print color2
['red', 'green']
```

A simple program

```
>>> # Fibonacci series:
... # the sum of two elements defines the next
... a, b = 0, 1
>>> while b < 10:
...     print b
...     a, b = b, a+b
...
1
1
2
3
5
8
```

- note the double input separated by the comma
- note the indentation
- <, >, ==, <=, >= are used as in C
- the while loop gets executed as long as its argument remains true (non-zero)
- a „return“ in the print statement can be avoided by a comma:

```
>>> a, b = 0, 1
>>> while b < 1000:
...     print b,
...     a, b = b, a+b
...
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
```

The if statement

```
>>> x = int(raw_input("Please enter an integer: "))
Please enter an integer: 42
>>> if x < 0:
...      x = 0
...      print 'Negative changed to zero'
... elif x == 0:
...      print 'Zero'
... elif x == 1:
...      print 'Single'
... else:
...      print 'More'
...
More
```

else is optional;
elif prevents indentation problems with successive *else if* statements
there is no need for brackets (as in C/C++)

The for statement

The *for* statement iterates over a sequence. This syntax differs from C/C++

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

For an iteration over numbers, *range* is useful:

```
>>> 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]
>>> range(-10, -100, -30)
[-10, -40, -70]
```

break, continue and else clauses on loops

Break causes a termination of the inner *for/while* loop (as in C). An *else* clause in a loop gets executed after termination of the loop (eg exhaustion of the list with *for* or a false statement with *while*) but not after *break*:

```
>>> for n in range(2, 10):
...     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'
...
2 is a prime number
3 is a prime number
4 equals 2 * 2
5 is a prime number
6 equals 2 * 3
7 is a prime number
8 equals 2 * 4
9 equals 3 * 3
```

Continue continues with the next iteration of the loop (as in C):

```
>>> for num in range(2, 10):
...     if num % 2 == 0:
...         print "Found an even number", num
...         continue
...     print "Found a number", num
Found an even number 2
Found a number 3
Found an even number 4
Found a number 5
Found an even number 6
Found a number 7
Found an even number 8
Found a number 9
```

Defining functions

The syntax is as follows:

```
>>> def fib(n):    # write Fibonacci series up to n
...     """Print a Fibonacci series up to n."""
...     a, b = 0, 1
...     while a < n:
...         print a,
...         a, b = b, a+b
...
>>> # Now call the function we just defined:
... fib(2000)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597
```

note the function definition, the docstring (line 2), and pay attention to the scope of variables. A function produces a new symbol table for the local variables. Function arguments are passed using *call by value*

```
>>> 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 a < n:
...         result.append(a)    # see below
...         a, b = b, a+b
...     return result
...
>>> f100 = fib2(100)    # call it
>>> f100                # write the result
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
```

note the use of methods — more on this when we talk about classes

Default function arguments

```
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
```

the function can be called with only the mandatory input, or with the optional input as well.
Also note the *in* keyword.
Default values are evaluated at the point of function definition in the defining scope.

```
In [27]: ask_ok('Do you really want to quit?')
Do you really want to quit?y
Out[27]: True

In [28]: ask_ok('Do you really want to quit?')
Do you really want to quit?n
Out[28]: False

In [29]: ask_ok('Do you really want to quit?')
Do you really want to quit?zzz
Yes or no, please!
Do you really want to quit?zzz
Yes or no, please!
Do you really want to quit?zzz
Yes or no, please!
Do you really want to quit?zzz
Yes or no, please!
Do you really want to quit?zzz
---------------------------------------------------------------------------
IOError                                   Traceback (most recent call last)
<ipython-input-29-82a04d7ba61a> in <module>()
----> 1 ask_ok('Do you really want to quit?')

<ipython-input-26-72bc66f26afa> in ask_ok(prompt, retries, complaint)
      8                 retries = retries - 1
      9                 if retries < 0:
---> 10                         raise IOError('refusenik user')
     11                 print complaint
     12

IOError: refusenik user
```

A little quiz

what is the output of the following program?

```
def f(a, L=[]):
    L.append(a)
    return L

print f(1)
print f(2)
```

A. 1 and 2
B. [1] and [2]
C. gives an error
D. none of the above

Vote now!

Default function arguments

this is a common pitfall in Python. The answer is:

```
def f(a, L=[]):
    L.append(a)
    return L

print f(1)
print f(2)
print f(3)
```

```
[1]
[1, 2]
[1, 2, 3]
```

contrast this with

```
def f(a, L=None):
    if L is None:
        L = []
    L.append(a)
    return L

print f(1)
print f(2)
print f(3)
```

```
[1]
[2]
[3]
```

The reason is that default values are evaluated at the point of function definition in the defining scope. They are evaluated only once.

keyword arguments

They have the form kwarg=value:

```
def parrot(voltage, state='a stiff', action='voom', type='Norwegian Blue'):
    print "-- This parrot wouldn't", action,
    print "if you put", voltage, "volts through it."
    print "-- Lovely plumage, the", type
    print "-- It's", state, "!"
```

There is one required argument (voltage) and 3 optional ones (state, action, and type).
Accepted calls are:

```
parrot(1000)                                          # 1 positional argument
parrot(voltage=1000)                                  # 1 keyword argument
parrot(voltage=1000000, action='VOOOOOM')             # 2 keyword arguments
parrot(action='VOOOOOM', voltage=1000000)             # 2 keyword arguments
parrot('a million', 'bereft of life', 'jump')         # 3 positional arguments
parrot('a thousand', state='pushing up the daisies')  # 1 positional, 1 keyword
```

```
In [48]: parrot(voltage=1000)
-- This parrot wouldn't voom if you put 1000 volts through it.
-- Lovely plumage, the Norwegian Blue
-- It's a stiff !

In [49]: parrot(voltage=100000, action='VOOOOM')
-- This parrot wouldn't VOOOOM if you put 100000 volts through it.
-- Lovely plumage, the Norwegian Blue
-- It's a stiff !

In [50]: parrot(action='VOOOOM', voltage=100)
-- This parrot wouldn't VOOOOM if you put 100 volts through it.
-- Lovely plumage, the Norwegian Blue
-- It's a stiff !

In [51]: parrot('a million', 'bereft of life', 'jump')
-- This parrot wouldn't jump if you put a million volts through it.
-- Lovely plumage, the Norwegian Blue
-- It's bereft of life !

In [52]: parrot('a million', 'bereft of life', 'jump', 'Canadian goose')
-- This parrot wouldn't jump if you put a million volts through it.
-- Lovely plumage, the Canadian goose
-- It's bereft of life !
```

keyword arguments

Keyword arguments must follow positional arguments; their order is not important. If the last argument is ***name* then it receives a *dictionary* containing all *keyword* arguments. This may be combined with a **name* preceding it which receives a *tuple* containing the *positional* arguments beyond the parameter list:

```
def cheeseshop(kind, *arguments, **keywords):
    print "-- Do you have any", kind, "?"
    print "-- I'm sorry, we're all out of", kind
    for arg in arguments:
        print arg
    print "-" * 40
    keys = sorted(keywords.keys())
    for kw in keys:
        print kw, ":", keywords[kw]
```

```
cheeseshop("Limburger", "It's very runny, sir.",
           "It's really very, VERY runny, sir.",
           shopkeeper='Michael Palin',
           client="John Cleese",
           sketch="Cheese Shop Sketch")
```

```
-- Do you have any Limburger ?
-- I'm sorry, we're all out of Limburger
It's very runny, sir.
It's really very, VERY runny, sir.
----------------------------------------
client : John Cleese
shopkeeper : Michael Palin
sketch : Cheese Shop Sketch
```

more on function arguments

To unpack from a tuple (when positional arguments are required), use also *

Eg, range() expects separate start and stop values,

```
In [53]: range(3,6)
Out[53]: [3, 4, 5]

In [54]: args = [3,6]

In [55]: range(args)
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-55-cf3b519380c5> in <module>()
----> 1 range(args)

TypeError: range() integer end argument expected, got list.

In [56]: range(*args)
Out[56]: [3, 4, 5]
```

In the same fashion, dictionaries can deliver keyword arguments by the ** operator

```
>>> def parrot(voltage, state='a stiff', action='voom'):
...     print "-- This parrot wouldn't", action,
...     print "if you put", voltage, "volts through it.",
...     print "E's", state, "!"
...
>>> d = {"voltage": "four million", "state": "bleedin' demised", "action": "VOOM"}
>>> parrot(**d)
-- This parrot wouldn't VOOM if you put four million volts through it. E's bleedin' demised !
```

lambda expressions

Small anonymous functions can be made with the *lambda* keyword:
they are restricted syntactically to a single line.

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

Here we used lambda to return a function (cf nested function)

They can also be used to pass a small function as a argument:

```
>>> pairs = [(1, 'one'), (2, 'two'), (3, 'three'), (4, 'four')]
>>> pairs.sort(key=lambda pair: pair[1])
>>> pairs
[(4, 'four'), (1, 'one'), (3, 'three'), (2, 'two')]
```

```
In [69]: pairs
Out[69]: [(1, 'one'), (2, 'two'), (3, 'three'), (4, 'four')]

In [70]: pairs.sort(key=lambda comp_rel:comp_rel[1])

In [71]: pairs
Out[71]: [(4, 'four'), (1, 'one'), (3, 'three'), (2, 'two')]

In [72]: pairs.sort(key=lambda comp_rel:comp_rel[0])

In [73]: pairs
Out[73]: [(1, 'one'), (2, 'two'), (3, 'three'), (4, 'four')]
```

Data structures

Lists are last-in, first-out or stack type structures:

methods:
- append
- extend
- insert
- remove
- pop
- index
- count
- sort
- reverse

```
>>> a = [66.25, 333, 333, 1, 1234.5]
>>> print a.count(333), a.count(66.25), a.count('x')
2 1 0
>>> a.insert(2, -1)
>>> a.append(333)
>>> a
[66.25, 333, -1, 333, 1, 1234.5, 333]
>>> a.index(333)
1
>>> a.remove(333)
>>> a
[66.25, -1, 333, 1, 1234.5, 333]
>>> a.reverse()
>>> a
[333, 1234.5, 1, 333, -1, 66.25]
>>> a.sort()
>>> a
[-1, 1, 66.25, 333, 333, 1234.5]
>>> a.pop()
1234.5
>>> a
[-1, 1, 66.25, 333, 333]
```

(for first-in, first out or queue structures, use *collections.deque*)

useful are *filter(function, sequence)* , *map(function, sequence), sum(sequence)* and *reduce(function, sequence)*

```
>>> def f(x): return x % 2 != 0 and x % 3 != 0
...
>>> filter(f, range(2, 25))
[5, 7, 11, 13, 17, 19, 23]
```

```
>>> def cube(x): return x*x*x
...
>>> map(cube, range(1, 11))
[1, 8, 27, 64, 125, 216, 343, 512, 729, 1000]
```

```
>>> seq = range(8)
>>> def add(x, y): return x+y
...
>>> map(add, seq, seq)
[0, 2, 4, 6, 8, 10, 12, 14]
```

List comprehensions

```
>>> squares = []
>>> for x in range(10):
...     squares.append(x**2)
...
>>> squares
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

is equivalent but conciser

```
squares = [x**2 for x in range(10)]
```

```
squares = map(lambda x: x**2, range(10))
```

A list comprehension consists of brackets containing an expression followed by a `for` clause, then zero or more `for` or `if` clauses

```
>>> [(x, y) for x in [1,2,3] for y in [3,1,4] if x != y]
[(1, 3), (1, 4), (2, 3), (2, 1), (2, 4), (3, 1), (3, 4)]
```

Example:

```
>>> vec = [-4, -2, 0, 2, 4]
>>> # create a new list with the values doubled
>>> [x*2 for x in vec]
[-8, -4, 0, 4, 8]
>>> # filter the list to exclude negative numbers
>>> [x for x in vec if x >= 0]
[0, 2, 4]
>>> # apply a function to all the elements
>>> [abs(x) for x in vec]
[4, 2, 0, 2, 4]
>>> # call a method on each element
>>> freshfruit = ['  banana', '  loganberry ', 'passion fruit  ']
>>> [weapon.strip() for weapon in freshfruit]
['banana', 'loganberry', 'passion fruit']
>>> # create a list of 2-tuples like (number, square)
>>> [(x, x**2) for x in range(6)]
[(0, 0), (1, 1), (2, 4), (3, 9), (4, 16), (5, 25)]
>>> # the tuple must be parenthesized, otherwise an error is raised
>>> [x, x**2 for x in range(6)]
  File "<stdin>", line 1
    [x, x**2 for x in range(6)]
               ^
SyntaxError: invalid syntax
>>> # flatten a list using a listcomp with two 'for'
>>> vec = [[1,2,3], [4,5,6], [7,8,9]]
>>> [num for elem in vec for num in elem]
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

A little quiz

```
>>> matrix = [
...     [1, 2, 3, 4],
...     [5, 6, 7, 8],
...     [9, 10, 11, 12],
... ]
```

What is the output of:

```
>>> [[a[k] for a in matrix] for k in range(4)]
```

A. it copies the first column 4 times
B. it copies the matrix
C. it transposes the matrix
D. it reverses the order of elements in every rwo

Vote now!

Nested list comprehensions

```
>>> matrix = [
...     [1, 2, 3, 4],
...     [5, 6, 7, 8],
...     [9, 10, 11, 12],
... ]
```

you can try to unravel the meaning of:

```
>>> [[row[i] for row in matrix] for i in range(4)]
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
```

```
>>> transposed = []
>>> for i in range(4):
...     # the following 3 lines implement the nested listcomp
...     transposed_row = []
...     for row in matrix:
...         transposed_row.append(row[i])
...     transposed.append(transposed_row)
...
>>> transposed
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
```

del statement

you can remove elements without specifying the value as follows:

```
>>> a = [-1, 1, 66.25, 333, 333, 1234.5]
>>> del a[0]
>>> a
[1, 66.25, 333, 333, 1234.5]
>>> del a[2:4]
>>> a
[1, 66.25, 1234.5]
>>> del a[:]
>>> a
[]
```

```
>>> del a
```

Tuples

A tuple consists of a number of values separated by commas

```
>>> 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))
>>> # Tuples are immutable:
... t[0] = 88888
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
>>> # but they can contain mutable objects:
... v = ([1, 2, 3], [3, 2, 1])
>>> v
([1, 2, 3], [3, 2, 1])
```

note the brackets. Tuples are immutable and usually contain a heterogeneous sequence of elements that are accessed after unpacking. Lists are mutable, usually homogeneous and elements are accessed by iterating.

Tuples with zero or one element require special syntax.

```
>>> empty = ()
>>> singleton = 'hello',    # <-- note trailing comma
>>> len(empty)
0
>>> len(singleton)
1
>>> singleton
('hello',)
```

```
t = 12345, 54321, 'hello!'
```

this is called tuple packing

inverse is sequence unpacking

```
>>> x, y, z = t
```

Sets

A set is an unordered collection with no duplicate elements.

Curly braces or the set() function can be used to create sets. Note: to create an empty set you have to use set(), not {}; the latter creates an empty dictionary

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

>>> # Demonstrate set operations on unique letters from two words
...
>>> 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 both
set(['r', 'd', 'b', 'm', 'z', 'l'])
```

also set comprehensions are supported:

```
>>> a = {x for x in 'abracadabra' if x not in 'abc'}
>>> a
set(['r', 'd'])
```

Dictionaries

Dictionaries are associative arrays; it is an unordered set of key:value pairs where the keys are unique. Dictionaries are indexed by keys, which can be any immutable type; strings and numbers can always be keys, tuples only if the members are immutable. Lists cannot be used as keys.

```
>>> 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']
>>> 'guido' in tel
True
```

constructor:

```
>>> dict([('sape', 4139), ('guido', 4127), ('jack', 4098)])
{'sape': 4139, 'jack': 4098, 'guido': 4127}
```

if simple:

```
>>> dict(sape=4139, guido=4127, jack=4098)
{'sape': 4139, 'jack': 4098, 'guido': 4127}
```

dict comprehensions:

```
>>> {x: x**2 for x in (2, 4, 6)}
{2: 4, 4: 16, 6: 36}
```

Modules

A module is a text file with python statements and definitions. The file name is the module name + the extension .py The module's name is available as __name__

create fibo.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
```

in the interpreter

```
>>> import fibo
```

```
>>> fibo.fib(1000)
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
>>> fibo.fib2(100)
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
>>> fibo.__name__
'fibo'
```

Modules

there are 2 other variants:

```
>>> from fibo import *ib, fib2
>>> fib(500)
1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

```
>>> from fibo import *
>>> fib(500)
1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

Modules can also be run as scripts:

```
if __name__ == "__main__":
    import sys
    fib(int(sys.argv[1]))
```

```
$ python fibo.py 50
1 1 2 3 5 8 13 21 34
```

The dir function

dir() is used to find which names a module defines:

```
>>> import fibo, sys
>>> dir(fibo)
['__name__', 'fib', 'fib2']
>>> dir(sys)
['__displayhook__', '__doc__', '__excepthook__', '__name__', '__package__',
 '__stderr__', '__stdin__', '__stdout__', '_clear_type_cache',
 '_current_frames', '_getframe', '_mercurial', 'api_version', 'argv',
 'builtin_module_names', 'byteorder', 'call_tracing', 'callstats',
 'copyright', 'displayhook', 'dont_write_bytecode', 'exc_clear', 'exc_info',
 'exc_traceback', 'exc_type', 'exc_value', 'excepthook', 'exec_prefix',
 'executable', 'exit', 'flags', 'float_info', 'float_repr_style',
 'getcheckinterval', 'getdefaultencoding', 'getdlopenflags',
 'getfilesystemencoding', 'getobjects', 'getprofile', 'getrecursionlimit',
 'getrefcount', 'getsizeof', 'gettotalrefcount', 'gettrace', 'hexversion',
 'long_info', 'maxint', 'maxsize', 'maxunicode', 'meta_path', 'modules',
 'path', 'path_hooks', 'path_importer_cache', 'platform', 'prefix', 'ps1',
 'py3kwarning', 'setcheckinterval', 'setdlopenflags', 'setprofile',
 'setrecursionlimit', 'settrace', 'stderr', 'stdin', 'stdout', 'subversion',
 'version', 'version_info', 'warnoptions']
```

without arguments it lists all names (variables, modules, functions,…) currently defined. If you want to see all built-in names instead:

```
>>> import __builtin__
>>> dir(__builtin__)
['ArithmeticError', 'AssertionError', 'AttributeError', 'BaseException',
 'BufferError', 'BytesWarning', 'DeprecationWarning', ,EOFError', …
```

For *packages* : see the documentation

input and output

Fancier output is possible with the *string.format* method

```
>>> for x in range(1,11):
...     print '{0:2d} {1:3d} {2:4d}'.format(x, x*x, x*x*x)
...
 1   1    1
 2   4    8
 3   9   27
 4  16   64
 5  25  125
 6  36  216
 7  49  343
 8  64  512
 9  81  729
10 100 1000
```

```
>>> print 'We are the {} who say "{}!"'.format('knights', 'Ni')
We are the knights who say "Ni!"
```

```
>>> print '{0} and {1}'.format('spam', 'eggs')
spam and eggs
>>> print '{1} and {0}'.format('spam', 'eggs')
eggs and spam
```

```
>>> import math
>>> print 'The value of PI is approximately {0:.3f}.'.format(math.pi)
The value of PI is approximately 3.142.
```

```
>>> print 'This {food} is {adjective}.'.format(
...       food='spam', adjective='absolutely horrible')
This spam is absolutely horrible.
```

the *zfill* method will pad to the left:

```
>>> '12'.zfill(5)
'00012'
>>> '-3.14'.zfill(7)
'-003.14'
>>> '3.14159265359'.zfill(5)
'3.14159265359'
```

Reading and writing files

open(filename, mode) opens a file named filename. mode can be ‚r‘ (read, the default), ‚w‘ (write), ‚a‘ (append), and ‚r+‘ (read and write).

```
>>> f = open('workfile', 'w')
>>> print f
<open file 'workfile', mode 'w' at 80a0960>
```

to read a file’s contents, use f.*read(size)* which reads some quantity of data and returns it as a string.

```
>>> f.read()
'This is the entire file.\n'
>>> f.read()
''
```

readline() reads a single line of the file

```
>>> f.readline()
'This is the first line of the file.\n'
>>> f.readline()
'Second line of the file\n'
>>> f.readline()
''
```

for reading lines, you can loop over the file object:

```
>>> for line in f:
        print line,

This is the first line of the file.
Second line of the file
```

if you want all lines of the file f, use *list(f)* or *f.readlines()*

Reading and writing files

f.write(string) writes the contents of the string to the file returning None

```
>>> f.write('This is a test\n')
```

To write something else than a string, it needs to be converted first:

```
>>> value = ('the answer', 42)
>>> s = str(value)
>>> f.write(s)
```

f.tell() returns an integer giving the file object's current position in the file, measured in bytes from the beginning of the file.

To change the position, use *f.seek(offset, from_what)*

```
>>> f = open('workfile', 'r+')
>>> f.write('0123456789abcdef')
>>> f.seek(5)     # Go to the 6th byte in the file
>>> f.read(1)
'5'
```

good style is to use the keyword *with* (automatically closes the file, also in case of exception)

```
>>> with open('workfile', 'r') as f:
...     read_data = f.read()
>>> f.closed
True
```

for json : see the documentation

Classes

```
class MyClass:
    """A simple example class"""
    i = 12345                                   # data attribute (called member in C++)
    def f(self):                                # method; note the (self)
        return 'hello world'
```

then MyClass.i and MyClass.f are valid attribute references, as well as MyClass.__doc__

For instantiation, type

```
x = MyClass()
```

and look at the output of *x.i* , *x.f()* , and *x.__doc__*
Remark: x.f() seems to violate the arguments of function f; in fact it is MyClass.f(x)
A new *instance* of the class has been created. Automatic creation of objects is possible via __init__():

```
>>> class Complex:
...     def __init__(self, realpart, imagpart):
...         self.r = realpart
...         self.i = imagpart
...
>>> x = Complex(3.0, -4.5)
>>> x.r, x.i
(3.0, -4.5)
```

Classes

You don’t have to call the method right away, the following is allowed:

```
xf = x.f
while True:
    print xf()
```

which will print „hello world“ indefinitely

Instance variables are identical to all instances of the class:

```
class Dog:
    kind = 'canine'         # class variable shared by all instances

    def __init__(self, name):
        self.name = name    # instance variable unique to each instance

>>> d = Dog('Fido')
>>> e = Dog('Buddy')
>>> d.kind                  # shared by all dogs
'canine'
>>> e.kind                  # shared by all dogs
'canine'
>>> d.name                  # unique to d
'Fido'
>>> e.name                  # unique to e
'Buddy'
```

Classes

Be careful with mutable objects (lists, dictionaries,…)

```
class Dog:

    tricks = []             # mistaken use of a class variable

    def __init__(self, name):
        self.name = name

    def add_trick(self, trick):
        self.tricks.append(trick)

>>> d = Dog('Fido')
>>> e = Dog('Buddy')
>>> d.add_trick('roll over')
>>> e.add_trick('play dead')
>>> d.tricks                # unexpectedly shared by all dogs
['roll over', 'play dead']
```

Correct design is

```
class Dog:

    def __init__(self, name):
        self.name = name
        self.tricks = []    # creates a new empty list for each dog

    def add_trick(self, trick):
        self.tricks.append(trick)

>>> d = Dog('Fido')
>>> e = Dog('Buddy')
>>> d.add_trick('roll over')
>>> e.add_trick('play dead')
>>> d.tricks
['roll over']
>>> e.tricks
['play dead']
```

Inheritance

Syntax for a derived class:

```
class DerivedClassName(BaseClassName):
    <statement-1>
    .
    .
    .
    <statement-N>
```

It proceeds all naturally. Methods of the base class may be overwritten by the derived class. In C++ parlance, all methods are *virtual*.

There are 2 built-in functions that work with inheritance

- isinstance(obj, dtype) checks if obj is of type dtype and returns a boolean
- issubclass(class1, class2) checks is class1 is a subclass of class 2 and returns a bool

Python supports a little bit of multiple inheritance:

```
class DerivedClassName(Base1, Base2, Base3):
    <statement-1>
    .
    .
    .
    <statement-N>
```

Private variables

They don’t exist.

There is some convention: names starting with an underscore (eg _spam) should be treated as non-public

To avoid clashes with subclasses, there is *name mangling:* names starting with 2 underscores (eg __bacon) are textually replaced by _classname__egs

```
class Mapping:
    def __init__(self, iterable):
        self.items_list = []
        self.__update(iterable)

    def update(self, iterable):
        for item in iterable:
            self.items_list.append(item)

    __update = update   # private copy of original update() method

class MappingSubclass(Mapping):

    def update(self, keys, values):
        # provides new signature for update()
        # but does not break __init__()
        for item in zip(keys, values):
            self.items_list.append(item)
```

Iterators

Most sequences can be looped over with a *for* statement

```
for element in [1, 2, 3]:
    print element
for element in (1, 2, 3):
    print element
for key in {'one':1, 'two':2}:
    print key
for char in "123":
    print char
for line in open("myfile.txt"):
    print line,
```

Internally, an iterator is created and iterates over the sequence. You can also use iterators explicitly

```
>>> s = 'abc'
>>> it = iter(s)
>>> it
<iterator object at 0x00A1DB50>
>>> it.next()
'a'
>>> it.next()
'b'
>>> it.next()
'c'
>>> it.next()
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
    it.next()
StopIteration
```

Iterators

From the following example you can learn how to add iterator functionality to your own class

```
class Reverse:
    """Iterator for looping over a sequence backwards."""
    def __init__(self, data):
        self.data = data
        self.index = len(data)
    def __iter__(self):
        return self
    def next(self):
        if self.index == 0:
            raise StopIteration
        self.index = self.index - 1
        return self.data[self.index]
>>>
>>> rev = Reverse('spam')
>>> iter(rev)
<__main__.Reverse object at 0x00A1DB50>
>>> for char in rev:
...     print char
...
m
a
p
s
```

Generators

Generators make life easy to use iterators. They are written as regular functions, but use the *yield* statement whenever they want to return data. Every time *next()* is called on it, the generator resumes where it left off

```
def reverse(data):
    for index in range(len(data)-1, -1, -1):
        yield data[index]

>>> for char in reverse('golf'):
...     print char
...
f
l
o
g
```

The same functionality could have been achieved with the previously discussed *__iter__* and *next()* methods. Also note the absence of the *self* keyword here.

Simple generator expressions can be coded in a way similar to list comprehensions but with parentheses instead of brackets. They are designed for situations where the generator is used right away by an enclosing function:

Generator expressions

Examples:

```
>>> sum(i*i for i in range(10))                 # sum of squares
285

>>> xvec = [10, 20, 30]
>>> yvec = [7, 5, 3]
>>> sum(x*y for x,y in zip(xvec, yvec))         # dot product
260

>>> from math import pi, sin
>>> sine_table = dict((x, sin(x*pi/180)) for x in range(0, 91))

>>> unique_words = set(word  for line in page  for word in line.split())

>>> valedictorian = max((student.gpa, student.name) for student in graduates)

>>> data = 'golf'
>>> list(data[i] for i in range(len(data)-1,-1,-1))
['f', 'l', 'o', 'g']
```

What next

See the Standard Python Library

https://docs.python.org/2/library/index.html#library-index

Additional modules:

- Numpy http://wiki.scipy.org/Tentative_NumPy_Tutorial
- Scipy http://www.scipy.org/scipylib/index.html
- Matplotlib http://matplotlib.org

and a lot of practice…