Introduction to Python for Scientific Computing

Bart Oldeman, Calcul Québec – McGill HPC
Bart.Oldeman@mcgill.ca

compute • calcul CANADA

Calcul Québec

Outline of the workshop

- What is Python? Why is it useful?
- How do we run Python codes, including on the Guillimin cluster?
- How to program with Python?
- How to use scientific packages in Python (NumPy, SciPy, Matplotlib)?
- Practical exercises on Guillimin
- Note: These slides are based on the lectures notes here: `http://scipy-lectures.github.io/`

Table of Contents

What is Python?

python™

- Python is a high-level, general-purpose programming language.
- Readable code.
- (Usually) Interpreted, not compiled.
- Dynamically typed.
- Original author and "Benevolent Dictator For Life (BFDL)": Guido van Rossum (originally at CWI, Amsterdam, now at Dropbox).
- Named after Monty Python, Van Rossum's favourite TV show.
- History: Started 1989, Python 1.0 (1994), Python 2.0 (2000), Python 3.0 (2008).

Why is it useful?

- Provides a general-purpose, open-source, high-level glue between computations.
- Alternative to programs such as MATLAB, R, IDL, etc., for many numerical computations. Python is less specialized, which has pros:
  - easier to interface with non-scientific code
  - can do almost everything

  and cons
  - slightly more cumbersome syntax for arrays
- Used in many disciplines, e.g. bioinformatics, mathematics, statistics, biology, economics, physics, electrical engineering, geosciences, signal processing.
- Groups on Guillimin use it, for instance, to study
  - Gene flow in animal populations
  - Climate change
  - Searching for pulsars
  - Computational Fluid Dynamics

Scientific Python stack

McGill HPC

Python	The base language. It has "batteries included" but no fast array type by default.
Numpy	Provides efficient powerful numerical array type with associated routines. Almost all scientific software written in Python uses Numpy.
Scipy	Higher level than Numpy, provides many data processing routines, for instance optimization, regression, interpolation.
Matplotlib	Most popular 2D (and basic 3D) plotting library.
Ipython	Advance Python shell that integrates well with scientific Python packages.
Mayavi	3D visualization.
Mpi4py	Use MPI from Python.
Cython	C-extensions for Python.
Sympy	Symbolic mathematics.

Exercise 1:

McGill HPC

Log in to Guillimin, setting up the environment

1) Log in to Guillimin:
   `ssh -X username@guillimin.clumeq.ca`
2) Check for loaded software modules:
   `guillimin> module list`
3) See all available modules:
   `guillimin> module av`
4) Load necessary modules:
   `guillimin> module add python/2.7.3`
5) Check loaded modules again
6) Verify that you have access to the correct Python package:
   `guillimin> which python`
   output: /software/CentOS-5/tools/python-2.7.3/bin/python

Interactive job submission

McGill HPC

1) Copy all workshop files to your home directory:

```
guillimin> cp -a /software/workshop/python/* .
```

2) Inspect the file "interactive.pbs":

```
guillimin> cat interactive.pbs
```

```
#!/bin/bash
#PBS -l nodes=1:ppn=1,walltime=08:00:00
#PBS -N python-interactive
#PBS -V
#PBS -X
#PBS -I
```


Interactive job submission

McGill HPC

3) `guillimin> msub -q class interactive.pbs`

```
qsub: waiting for job 12703020.gm-1r14-n05.guillimin.clumeq.ca to start

qsub: job 12703020.gm-1r14-n05.guillimin.clumeq.ca ready

----------------------------------------
Begin PBS Prologue Mon Aug  5 12:38:18 EDT 2013 1375720698
Job ID:         12703020.gm-1r14-n05.guillimin.clumeq.ca
Username:       classxx
Group:          classxx
Nodes:          sw-2r13-n30
End PBS Prologue Mon Aug  5 12:38:19 EDT 2013 1375720699
----------------------------------------
[classxx@sw-2r13-n30 ~]$
```

Run "Hello World"

4)
```
[classxx@sw-2r13-n30 ~]$ cat hello.py
print "Hello, world!"
[classxx@sw-2r13-n30 ~]$ python hello.py
Hello, world!
[classxx@sw-2r13-n30 ~]$ python
 Python 2.7.3 (default, Sep 17 2012, 10:35:59)
[GCC 4.1.2 20080704 (Red Hat 4.1.2-50)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> print "Hello, world!"
Hello, world!
>>> execfile("hello.py")
Hello, world!
>>> quit()
```

Note: for Python 3 (or Python 2.6+ with `from __future__ import print_function`) use `print("Hello, world!")` instead.

Run "Hello World" (IPython)

5)
```
[classxx@sw-2r13-n30 ~]$ ipython
 Python 2.7.3 (default, Sep 17 2012, 10:35:59)
Type "copyright", "credits" or "license" for more information.

IPython 0.13.2 -- An enhanced Interactive Python.
?         -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help      -> Python's own help system.
object?   -> Details about 'object', use 'object??' for extra details.

In [1]: print "Hello, world!"
Hello, world!
In [2]: run hello.py
Hello, world!
In [3]: quit
```


First steps

Start the Ipython shell as before.

To get started, type the following stack of instructions.

```
>>> a = 3
>>> b = 2*a
>>> type(b)
<type 'int'>
>>> print b
6
>>> a*b
18
```

```
>>> b = 'hello'
>>> type(b)
<type 'str'>
>>> b + b
'hellohello'
>>> 2*b
'hellohello'
```

- Two variables a and b have been defined.
- One does not declare the type of a variable (In C, one should write `int a = 3;`), but it is given at assignment time.
- The type of a variable may change (b changes from type int to type str).
- Additions and multiplications on strings amount to concatenation and repetition.

Numerical types

Python supports the following numerical, scalar types:

Integer:

```
>>> 1 + 1
2
>>> a = 4
>>> type(a)
<type 'int'>
```

Complex:

```
>>> a = 1.5 + 0.5j
>>> a.real
1.5
>>> a.imag
0.5
>>> type(1. + 0j )
<type 'complex'>
```

Floats:

```
>>> c = 2.1
>>> type(c)
<type 'float'>
>>> import math
>>> math.sqrt(2.1)
1.449137674618944
```

Booleans:

```
>>> 3 > 4
False
>>> test = (3 > 4)
>>> test
False
>>> type(test)
<type 'bool'>
```

Python as a calculator

McGill HPC

A Python shell can replace your pocket calculator, with the basic arithmetic operations +, -, *, /, % (modulo) natively implemented

```
>>> 7 * 3.
21.0
>>> 2**10
1024
>>> 8 % 3
2
```

Type conversion (casting):

```
>>> float(1)
1.0
```

Warning: Integer division

```
>>> 3 / 2
1
>>> from __future__ import division
>>> 3 / 2
1.5
```

Trick: use floats:

```
>>> 3 / 2.
1.5

>>> a = 3
>>> b = 2
>>> a / b
1
>>> a / float(b)
1.5
```

If you explicitly want integer division use //:

```
>>> 3.0 // 2
1.0
```

Note: Python 3 always returns floats for / division.

Lists

McGill HPC

A list is an ordered collection of objects, that may have different types.
For example:

```
>>> L = ['one', 'two', 'three', 'four', 'five']
>>> type(L)
<type 'list'>
```

Indexing: accessing individual objects contained in the list:

```
>>> L[2]
'three'
```

Counting from the end with negative indices:

```
>>> L[-1]
'five'
>>> L[-2]
'four'
```

Warning

Indexing starts at 0 (as in C), not at 1 (as in Fortran or Matlab)!

Slicing

McGill HPC

Obtaining sublists of regularly-spaced elements:

```
>>> L
['one', 'two', 'three', 'four', 'five']
>>> L[2:4]
['three', 'four']
```

Note that `L[start:stop]` contains the elements with indices `i` such as `start<= i <stop` (`i` ranging from `start` to `stop-1`).
Therefore, `L[start:stop]` has `(stop-start)` elements.
Slicing syntax: `L[start:stop:stride]`
All slicing parameters are optional:

```
>>> L
['one', 'two', 'three', 'four', 'five']
>>> L[3:]
['four', 'five']
>>> L[:3]
['one', 'two', 'three']
>>> L[::2]
['one', 'three', 'five']
```


More about lists

McGill HPC

Lists are **mutable** objects and can be modified:

```
>>> L[0] = 'yellow'
>>> L
['yellow', 'two', 'three', 'four', 'five']
>>> L[2:4] = ['gray', 'seven']
>>> L
['yellow', 'two', 'gray', 'seven', 'five']
```

The elements of a list may have different types:

```
>>> L = [3, -200, 'hello']
>>> L
[3, -200, 'hello']
>>> L[1], L[2]
(-200, 'hello')
```

Later: use NumPy arrays for more efficient collections of numerical data that all have the same type.

List methods

McGill HPC

Python offers a large number of functions to modify lists, or query them. Here are a few examples. For more information type:

```
>>> help(list)
```

Add and remove elements:

```
>>> L = ['one', 'two', 'three', 'four', 'five']
>>> L.append('six')
>>> L
['one', 'two', 'three', 'four', 'five', 'six']
>>> L.pop() # removes and returns the last item
'six'
>>> L
['one', 'two', 'three', 'four', 'five']
>>> L.extend(['six', 'seven']) # extend L, in-place
>>> L
['one', 'two', 'three', 'four', 'five', 'six', 'seven']
>>> L = L[:-2]
>>> L
['one', 'two', 'three', 'four', 'five']
```


More list methods

McGill HPC

Reverse:

```
>>> r = L[::-1]
>>> r
['five', 'four', 'three', 'two', 'one']
>>> r2 = list(L)
>>> r2
['one', 'two', 'three', 'four', 'five']
>>> r2.reverse() # in-place
>>> r2
['five', 'four', 'three', 'two', 'one']
```

Concatenate and repeat lists:

```
>>> r + L
['five','four','three','two','one','one','two','three','four','five']
>>> r * 2
['five','four','three','two','one','five','four','three','two','one']
```

Sort:

```
>>> sorted(r) # new object
['five', 'four', 'one', 'three', 'two']
>>> r
['five', 'four', 'three', 'two', 'one']
>>> r.sort()  # in-place
>>> r
['five', 'four', 'one', 'three', 'two']
```


Methods and Object-Oriented Programming

McGill HPC

The notation `r.method()` (`r.append(3)`, `L.pop()`, ...) is a first example of object-oriented programming (OOP).

The `list` object `r` owns the method function called using the notation `..`

In Ipython: use tab-completion (press tab) to find methods.

```
In [1]: r.<TAB>
r.__add__           r.__iadd__          r.__setattr__
r.__class__         r.__imul__          r.__setitem__
r.__contains__      r.__init__          r.__setslice__
r.__delattr__       r.__iter__          r.__sizeof__
r.__delitem__       r.__le__            r.__str__
r.__delslice__      r.__len__           r.__subclasshook__
r.__doc__           r.__lt__            r.append
r.__eq__            r.__mul__           r.count
r.__format__        r.__ne__            r.extend
r.__ge__            r.__new__           r.index
r.__getattribute__  r.__reduce__        r.insert
r.__getitem__       r.__reduce_ex__     r.pop
r.__getslice__      r.__repr__          r.remove
r.__gt__            r.__reversed__      r.reverse
r.__hash__          r.__rmul__          r.sort
```


Strings

McGill HPC

Different string syntaxes (simple, double or triple quotes):

```
s = 'Hello, how are you?'
s = "Hi, what's up"
s = '''Hello,                  # tripling the quotes allows the
       how are you'''          # the string to span more than one line
s = """Hi,
what's up?"""

>>> 'Hi, what's up'
  File "<stdin>", line 1
    'Hi, what's up'
               ^
SyntaxError: invalid syntax
```

The newline character is \n, and the tab character is \t.

String indexing and slicing

McGill HPC

Strings are collections like lists. Hence they can be indexed and sliced, using the same syntax and rules.

Indexing:

```
>>> a = "hello"
>>> a[0]
'h'
>>> a[1]
'e'
>>> a[-1]
'o'
```

(Remember that negative indices correspond to counting from the right.)

Slicing:

```
>>> a = "hello, world!"
>>> a[3:6] # 3rd to 6th (excluded) elements: elements 3, 4, 5
'lo,'
>>> a[2:10:2] # Syntax: a[start:stop:step]
'lo o'
>>> a[::3] # every three characters, from beginning to end
'hl r!'
```

Accents and special characters can also be handled in Unicode strings.

A string is an **immutable** object: not possible to modify its contents.

So, create new strings from the original ones.

String methods

McGill HPC

Useful methods are `replace`, `strip`, `split`, and `join`.

```
>>> a = "hello, world!"
>>> a[2] = 'z'
--------------------------------------------------------------------------
Traceback (most recent call last):
   File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment

>>> a.replace('l', 'z', 1)
'hezlo, world!'
>>> a.replace('l', 'z')
'hezzo, worzd!'
```

String substitution:

```
>>> 'An integer: %i; a float: %f; another string: %s' % (1, 0.1, 'string')
'An integer: 1; a float: 0.100000; another string: string'
>>> i = 102
>>> filename = 'processing_of_dataset_%d.txt' % i
>>> filename
'processing_of_dataset_102.txt'
```


Dictionaries

McGill HPC

A dictionary is an efficient table that **maps keys to values**. It is an **unordered** container.

```
>>> tel = {'emmanuelle': 5752, 'sebastian': 5578}
>>> tel['francis'] = 5915
>>> tel
{'sebastian': 5578, 'francis': 5915, 'emmanuelle': 5752}
>>> tel['sebastian']
5578
>>> tel.keys()
['sebastian', 'francis', 'emmanuelle']
>>> tel.values()
[5578, 5915, 5752]
>>> 'francis' in tel
True
```

It can be used to conveniently store and retrieve values associated with a name (a string for a date, a name, etc.).

A dictionary can have keys (resp. values) with different types:

```
>>> d = {'a':1, 'b':2, 3:'hello'}
>>> d
{'a': 1, 3: 'hello', 'b': 2}
```


Tuples and Sets

McGill HPC

Tuples are immutable lists. The elements of a tuple are written between parentheses, or only separated by commas:

```
>>> t = 12345, 54321, 'hello!'
>>> t[0]
12345
>>> t
(12345, 54321, 'hello!')
>>> u = (0, 2)
```

Sets: unordered, unique items:

```
>>> s = set(('a', 'b', 'c', 'a'))
>>> s
set(['a', 'c', 'b'])
>>> s.difference(('a', 'b'))
set(['c'])
>>> t = set(('a', 'b', 'd'))
>>> s & t
set(['a', 'b'])
>>> s | t
set(['a', 'c', 'b', 'd'])
```

Assignment operator

McGill HPC

Assignment statements are used to assign (or bind) **names** to values and to modify attributes or items of mutable objects.

```
# a single object can have
# several names bound to it:
>>> a = [1, 2, 3]
>>> b = a
>>> a
[1, 2, 3]
>>> b
[1, 2, 3]
>>> a is b
True
>>> b[1] = 'hi!'
>>> a
[1, 'hi!', 3]
```

```
# to change a list in place,
# use indexing/slices:
>>> a = [1, 2, 3]
>>> a
[1, 2, 3]
>>> a = ['a', 'b', 'c'] # Creates new object.
>>> a
['a', 'b', 'c']
>>> id(a)
138641676 # Yours will differ...
>>> a[:] = [1, 2, 3] # Modifies in place.
>>> a
[1, 2, 3]
>>> id(a)
138641676 # Same as id(a) output above
```

The key concept here is mutable vs. immutable:

mutable objects can be changed in place

immutable objects cannot be modified once created

Control Flow

McGill HPC

Controls the order in which the code is executed.

if/elif/else

```
>>> if 2**2 == 4:
...     print 'Obvious!'
...
Obvious!
```

Blocks are delimited by **indentation**. Convention in Python is to use 4 spaces, no hard Tabs. Type the following (press Enter 2x to leave block):

```
>>> a = 10
>>> if a == 1:
...     print(1)
... elif a == 2:
...     print(2)
... else:
...     print('A lot')
...
A lot
```

Indentation is compulsory in scripts as well.

Exercise 2 Re-type the previous lines with the same indentation in a script `condition.py`, then execute the script.

Conditional Expressions

McGill HPC

if <OBJECT >: **Evaluates to False**:

any number equal to zero (0, 0.0, 0+0j)

an empty container (list, tuple, set, dictionary, ...)

`False`, `None`

Evaluates to True: everything else

a == b: Tests equality, with logics:

```
>>> 1 == 1.
True
```

a is b: Tests identity:

both sides are the same object:

```
>>> 1 is 1.
False

>>> a = 1
>>> b = 1
>>> a is b
True
```

a in b:

For any collection `b`: `b` contains `a`

```
>>> b = [1, 2, 3]
>>> 2 in b
True
>>> 5 in b
False
```

If `b` is a dictionary, this tests that `a` is a key of `b`.

Loops

McGill HPC

for/range Iterating with an index:

```
>>> for i in range(4):
...     print(i)
0
1
2
3
```

But most often, it is more readable to iterate over values:

```
>>> for word in ('cool', 'powerful', 'readable'):
...     print('Python is %s' % word)
Python is cool
Python is powerful
Python is readable
```

while/break/continue Typical C-style while loop (Mandelbrot problem):

```
>>> z = 1 + 1j
>>> while abs(z) < 100:
...     z = z**2 + 1
>>> z
(-134+352j)
```

More on loops

```
# break out of enclosing
# for/while loop:
>>> z = 1 + 1j

>>> while abs(z) < 100:
...     if z.imag == 0:
...         break
...     z = z**2 + 1
```

```
# continue the next iteration
# of a loop:
>>> a = [1, 0, 2, 4]
>>> for element in a:
...     if element == 0:
...         continue
...     print 1. / element
1.0
0.5
0.25
```

Iterate over any sequence (string, list, keys in dictionary, lines in file, ...)
Can make code very readable, eliminating use of indices.

```
>>> vowels = 'aeiouy'

>>> for i in 'powerful':
...     if i in vowels:
...         print(i),
o e u
```

```
>>> message = "Hello how are you?"
>>> message.split() # returns a list
['Hello', 'how', 'are', 'you?']
>>> for word in message.split():
...     print word
...
Hello
how
are
you?
```


Modifying the sequence

Warning: Not safe to modify the sequence you are iterating over. Here we must keep track of the enumeration number.

```
>>> words = ('cool', 'powerful', 'readable')
>>> for i in range(len(words)):
...     print i, words[i]
0 cool
1 powerful
2 readable
```

Python also provides the `enumerate` keyword for this:

```
>>> for index, item in enumerate(words):
...     print index, item
0 cool
1 powerful
2 readable
```


More loops

Looping over a dictionary: use iteritems or not

```
>>> d = {'a': 1, 'b':1.2, 'c':1j}

>>> for key, val in d.iteritems():
...     print('Key: %s has value: %s' % (key, val))
Key: a has value: 1
Key: c has value: 1j
Key: b has value: 1.2
>>> for key in d: # Python 3 compatible.
...     print('Key: %s has value: %s' % (key, val[key]))
```

List Comprehensions

```
>>> [i**2 for i in range(4)]
[0, 1, 4, 9]
```

Exercise 3

Compute the decimals of Pi using the Wallis formula:

$$\pi = 2 \prod_{i=1}^{\infty} \frac{4i^2}{4i^2 - 1}$$

Functions

Function blocks must be indented as other control-flow blocks.

```
>>> def test():
...     print('in test function')
...
...

>>> test()
in test function
```

```
>>> def disk_area(radius):
...     return 3.14 * radius * radius
...

>>> disk_area(1.5)
7.0649999999999995
```

`return` statement: functions can optionally return values. By default, functions return `None`.

Note the syntax to define a function:

- the def keyword;
- is followed by the function's name, then
- the arguments of the function are given between parentheses followed by a colon.
- the function body;
- and `return object` for optionally returning values.

Exercise 4: Write a function that displays the n first terms of the Fibonacci sequence, defined by: $u_0 = 1; u_1 = 1; u_{n+2} = u_{n+1} + u_n$

Docstrings

McGill HPC

Documentation about what the function does and its parameters.

General convention:

```
In [1]: def funcname(params):
   ....:     """Concise one-line sentence describing the function.
   ....:
   ....:     Extended summary which can contain multiple paragraphs.
   ....:     """
   ....:     # function body
   ....:     pass
   ....:

In [2]: funcname?
Type:           function
Base Class:     type 'function'>
String Form:    <function funcname at 0xeaa0f0>
Namespace:      Interactive
File:           <ipython console>
Definition:     funcname(params)
Docstring:
    Concise one-line sentence describing the function.

    Extended summary which can contain multiple paragraphs.
```


The NumPy array object

McGill HPC

Python objects:

- high-level number objects: integers, floating point
- containers: lists (costless insertion and append), dictionaries (fast lookup)

NumPy provides:

- extension package to Python for multi-dimensional arrays
- closer to hardware (efficiency)
- designed for scientific computation (convenience)

```
>>> import numpy as np
>>> a = np.array([0, 1, 2, 3])
>>> a
array([0, 1, 2, 3])
```


NumPy array examples

McGill HPC

An array containing:

- values of an experiment/simulation at discrete time steps
- signal recorded by a measurement device, e.g. sound wave
- pixels of an image, grey-level or colour
- 3-D data measured at different X-Y-Z positions, e.g. MRI scan
- ...

Why it is useful: Memory-efficient container that provides fast numerical operations.

```
In [1]: L = range(1000)

In [2]: %timeit [i**2 for i in L]
1000 loops, best of 3: 403 us per loop

In [3]: a = np.arange(1000)

In [4]: %timeit a**2
100000 loops, best of 3: 12.7 us per loop
```


NumPy documentation and help

McGill HPC

Reference documentation: `http://docs.scipy.org/`

Interactive help:

```
In [5]: np.array?
String Form:<built-in function array>
Docstring:
array(object, dtype=None, copy=True, order=None, subok=False, ndmin=0, ...
```

```
>>> help(np.array)
Help on built-in function array in module numpy.core.multiarray:
array(...)
...
>>> np.lookfor('create array')
Search results for 'create array'
---------------------------------
numpy.array
    Create an array.
numpy.memmap
    Create a memory-map to an array stored in a *binary* file on disk.
...
```

```
In [6]: np.con*?
np.concatenate
np.conj ...
```

Creating arrays

McGill HPC

```
#1-D:
>>> a = np.array([0, 1, 2, 3])
>>> a
array([0, 1, 2, 3])
>>> a.ndim
1
>>> a.shape
(4,)
>>> len(a)
4
```

```
#2-D, 3-D, ...
# 2 x 3 array
>>> b = np.array([[0, 1, 2], [3, 4, 5]])
>>> b
array([[0, 1, 2],
       [3, 4, 5]])
>>> b.ndim
2
>>> b.shape
(2, 3)
>>> # returns size of first dimension
>>> len(b)
2
>>> c = np.array([[[1], [2]], [[3], [4]]])
>>> c
array([[[1],
        [2]],

       [[3],
        [4]]])
>>> c.shape
(2, 2, 1)
```


In practice, rarely enter items one by one

McGill HPC

Evenly spaced:

```
>>> a = np.arange(10) # 0 .. n-1  (!)
>>> a
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> b = np.arange(1, 9, 2) # start, end (exlusive), step
>>> b
array([1, 3, 5, 7])
```

or by number of points:

```
>>> c = np.linspace(0, 1, 6)   # start, end, num-points
>>> c
array([ 0. ,  0.2,  0.4,  0.6,  0.8,  1. ])
>>> d = np.linspace(0, 1, 5, endpoint=False)
>>> d
array([ 0. ,  0.2,  0.4,  0.6,  0.8])
```

Random numbers:

```
>>> a = np.random.rand(4)       # uniform in [0, 1]
>>> a
array([ 0.95799151,  0.14222247,  0.08777354,  0.51887998])
>>> b = np.random.randn(4)  # Gaussian
>>> b
array([ 0.37544699, -0.11425369, -0.47616538,  1.79664113])
>>> np.random.seed(1234)        # Setting the random seed
```


Common arrays

McGill HPC

```
>>> a = np.ones((3, 3))  # reminder: (3, 3) is a tuple
>>> a
array([[ 1.,  1.,  1.],
       [ 1.,  1.,  1.],
       [ 1.,  1.,  1.]])
>>> b = np.zeros((2, 2))
>>> b
array([[ 0.,  0.],
       [ 0.,  0.]])
>>> c = np.eye(3)
>>> c
array([[ 1.,  0.,  0.],
       [ 0.,  1.,  0.],
       [ 0.,  0.,  1.]])
>>> d = np.diag(np.array([1, 2, 3, 4]))
>>> d
array([[1, 0, 0, 0],
       [0, 2, 0, 0],
       [0, 0, 3, 0],
       [0, 0, 0, 4]])
```


Exercises

McGill HPC

5) Create the following arrays (with correct data types):

```
[[1, 1, 1, 1],
 [1, 1, 1, 1],
 [1, 1, 1, 2],
 [1, 6, 1, 1]]
```

```
[[0., 0., 0., 0., 0.],
 [2., 0., 0., 0., 0.],
 [0., 3., 0., 0., 0.],
 [0., 0., 4., 0., 0.],
 [0., 0., 0., 5., 0.],
 [0., 0., 0., 0., 6.]]
```

Par on course: 3 statements for each

Hint: Individual array elements can be accessed similarly to a list, e.g. `a[1]` or `a[1, 2]`.

Hint: Examine the docstring for diag.

6) Tiling for array creation

Skim through the documentation for np.tile, and use this function to construct the array:

```
[[4, 3, 4, 3, 4, 3],
 [2, 1, 2, 1, 2, 1],
 [4, 3, 4, 3, 4, 3],
 [2, 1, 2, 1, 2, 1]]
```

Basic data types

McGill HPC

Sometimes array elements are displayed with a trailing dot (e.g. `2.` vs `2`). They have different data-types:

```
>>> a = np.array([1, 2, 3])
>>> a.dtype
dtype('int64')

>>> b = np.array([1., 2., 3.])
>>> b.dtype
dtype('float64')
```

Different data-types allow for more compact storage in memory.
Mostly we simply work with floating point numbers.
By default NumPy auto-detects the data-type from the input.
You can explicitly specify which data-type you want:

```
>>> c = np.array([1, 2, 3], dtype=float)
>>> c.dtype
dtype('float64')
```


Types

McGill HPC

The default data type (without auto-detection) is floating point

```
>>> a = np.ones((3, 3))
>>> a.dtype
dtype('float64')
```

There are also other types:

Complex:

```
>>> d = np.array([1+2j, 3+4j, 5+6*1j])
>>> d.dtype
dtype('complex128')
```

Bool:

```
>>> e = np.array([True, False, False, True])
>>> e.dtype
dtype('bool')
```

Strings:

```
>>> f = np.array(['Bonjour', 'Hello', 'Hallo',])
>>> f.dtype     # <--- strings containing max. 7 letters
dtype('S7')
```

Much more:

`int32/int64...`

Basic visualization

McGill HPC

Now that we have our first data arrays, we are going to visualize them.
Start by launching IPython in pylab mode: `$ ipython --pylab`
Matplotlib is a 2D plotting package. Import its functions as follows:

```
>>> import matplotlib.pyplot as plt  # the tidy way
```

1D plotting:

```
>>> x = np.linspace(0, 3, 20)
>>> y = np.linspace(0, 9, 20)
>>> plt.plot(x, y)       # line plot
[<matplotlib.lines.Line2D object at ...>]
>>> plt.plot(x, y, 'o')  # dot plot
[<matplotlib.lines.Line2D object at ...>]
>>> plt.show()           # <-- shows the plot (not needed with Ipython)
```

2D arrays (such as images):

```
>>> image = np.random.rand(30, 30)
>>> plt.imshow(image, cmap=plt.cm.gray)
<matplotlib.image.AxesImage object at ...>
>>> plt.colorbar()
<matplotlib.colorbar.Colorbar instance at ...>
>>> plt.show()
```


Indexing and slicing

McGill HPC

Accessing and assigning to items as for other sequences (e.g. lists)

```
>>> a = np.arange(10)
>>> a
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> a[0], a[2], a[-1]
(0, 2, 9)
```

For multidimensional arrays, indices are tuples of integers:

```
>>> a = np.diag(np.arange(3))
>>> a
array([[0, 0, 0],
       [0, 1, 0],
       [0, 0, 2]])
>>> a[1, 1]
1
>>> a[2, 1] = 10 # third line, second column
>>> a
array([[ 0,  0,  0],
       [ 0,  1,  0],
       [ 0, 10,  2]])
>>> a[1]
array([0, 1, 0])
```

Indexing and slicing

McGill HPC

In 2D, the first dimension corresponds to rows, the second to columns.
For multidimensional `a`, `a[0]` gets all elements in unspecified dimensions.
Slicing Arrays, like other Python sequences can also be sliced:

```
>>> a = np.arange(10)
>>> a
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> a[2:9:3] # [start:end:step]
array([2, 5, 8])
```

Note that the last index is not included!

```
>>> a[:4]
array([0, 1, 2, 3])
```

All three slice components are not required: by default, `start` is 0, `end` is the last and `step` is 1:

```
>>> a[1:3]
array([1, 2])
>>> a[::2]
array([0, 2, 4, 6, 8])
>>> a[3:]
array([3, 4, 5, 6, 7, 8, 9])
```


Copies and views

McGill HPC

A slicing operation creates a **view** on the original array, which is just a way of accessing array data. Thus the original array is **not** copied in memory.
When modifying the view, the original array is modified as well:

```
>>> a = np.arange(10)
>>> a
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> b = a[::2]; b
array([0, 2, 4, 6, 8])
>>> b[0] = 12
>>> b
array([12,  2,  4,  6,  8])
>>> a   # (!)
array([12,  1,  2,  3,  4,  5,  6,  7,  8,  9])

>>> a = np.arange(10)
>>> b = a[::2].copy()  # force a copy
>>> b[0] = 12
>>> a
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```


Copies and views

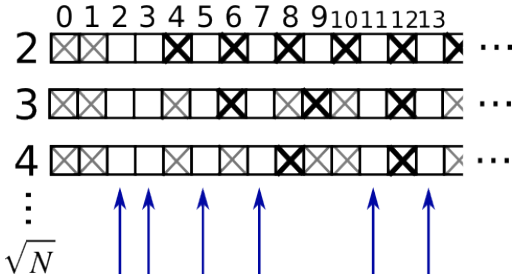
McGill HPC

This behavior can be surprising at first sight, but it allows to save both memory and time.
Warning: the transpose is a view.
As a result, a matrix cannot be made symmetric in-place:

```
>>> a = np.ones((100, 100))
>>> a += a.T
>>> a
array([[ 2.,  2.,  2., ...,  2.,  2.,  2.],
       [ 2.,  2.,  2., ...,  2.,  2.,  2.],
       [ 2.,  2.,  2., ...,  2.,  2.,  2.],
       ...,
       [ 3.,  3.,  3., ...,  2.,  2.,  2.],
       [ 3.,  3.,  3., ...,  2.,  2.,  2.],
       [ 3.,  3.,  3., ...,  2.,  2.,  2.]])
```


Worked example: Prime number sieve

McGill HPC



Compute prime numbers in 0–99, with a sieve
Construct a shape (100,) boolean array `is_prime`, filled with `True` in the beginning:

```
>>> is_prime = np.ones((100,), dtype=bool)
```

Cross out 0 and 1 which are not primes:

```
>>> is_prime[:2] = 0
```

Worked example: Prime number sieve

McGill HPC

For each integer `j` starting from 2, cross out its higher multiples:

```
>>> N_max = int(np.sqrt(len(is_prime)))
>>> for j in range(2, N_max):
...     is_prime[2*j::j] = False
```

Skim through `help(np.nonzero)`, and print the prime numbers
Follow-up:

- Move the above code into a script file named prime_sieve.py
- Run it to check it works
- Convert the simple sieve to the sieve of Eratosthenes:
  1. Skip `j` which are already known to not be primes
  2. The first number to cross out is j^2

Fancy indexing

McGill HPC

Numpy arrays can be indexed with slices, but also with boolean or integer arrays (**masks**): *fancy indexing*. This creates **copies not views**.
Using boolean masks:

```
>>> np.random.seed(3)
>>> a = np.random.random_integers(0, 20, 15)
>>> a
array([10,  3,  8,  0, 19, 10, 11,  9, 10,  6,  0, 20, 12,  7, 14])
>>> (a % 3 == 0)
array([False,  True, False,  True, False, False, False,  True, False,
        True,  True, False,  True, False, False], dtype=bool)
>>> mask = (a % 3 == 0)
>>> extract_from_a = a[mask] # or,  a[a%3==0]
>>> extract_from_a           # extract a sub-array with the mask
array([ 3,  0,  9,  6,  0, 12])
```

Indexing with a mask is useful to assign a new value to a sub-array:

```
>>> a[a % 3 == 0] = -1
>>> a
array([10, -1,  8, -1, 19, 10, 11, -1, 10, -1, -1, 20, -1,  7, 14])
```


Indexing with an array of integers

McGill HPC

```
>>> a = np.arange(10)
>>> a
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

Index using an array of integers, where indices are repeated several times:

```
>>> a[[2, 3, 2, 4, 2]]  # note: [2, 3, 2, 4, 2] is a Python list
array([2, 3, 2, 4, 2])
```

New values can be assigned with this kind of indexing:

```
>>> a[[9, 7]] = -10
>>> a
array([  0,   1,   2,   3,   4,   5,   6, -10,   8, -10])
```

When a new array is created by indexing with an array of integers, the new array has the same shape as the array of integers:

```
>>> a = np.arange(10)
>>> idx = np.array([[3, 4], [9, 7]])
>>> a[idx]
array([[3, 4],
       [9, 7]])
>>> b = np.arange(10)
```


Elementwise array operations

McGill HPC

With scalars:

```
>>> a = np.array([1, 2, 3, 4])
>>> a + 1
array([2, 3, 4, 5])
>>> 2**a
array([ 2,  4,  8, 16])
```

All arithmetic operates elementwise:

```
>>> b = np.ones(4) + 1
>>> a - b
array([-1.,  0.,  1.,  2.])
>>> a * b
array([ 2.,  4.,  6.,  8.])

>>> j = np.arange(5)
>>> 2**(j + 1) - j
array([ 2,  3,  6, 13, 28])
```

Warning
Array multiplication is not matrix multiplication:

```
>>> c = np.ones((3, 3))
>>> c * c # NOT matrix multiplication!
array([[ 1.,  1.,  1.],
       [ 1.,  1.,  1.],
       [ 1.,  1.,  1.]])
```

Matrix multiplication

```
>>> c.dot(c)
array([[ 3.,  3.,  3.],
       [ 3.,  3.,  3.],
       [ 3.,  3.,  3.]])
>>> np.matrix(c) ** 2
matrix([[ 3.,  3.,  3.],
        [ 3.,  3.,  3.],
        [ 3.,  3.,  3.]])
```

Array operations

McGill HPC

Comparisons:

```
>>> a = np.array([1, 2, 3, 4])
>>> b = np.array([4, 2, 2, 4])
>>> a == b
array([False,  True, False,  True], dtype=bool)
>>> a > b
array([False, False,  True, False], dtype=bool)
```

Logical operations:

```
>>> a = np.array([1, 1, 0, 0], dtype=bool)
>>> b = np.array([1, 0, 1, 0], dtype=bool)
>>> np.logical_or(a, b)
array([ True,  True,  True, False], dtype=bool)
>>> np.logical_and(a, b)
array([ True, False, False, False], dtype=bool)
```

Shape mismatches

```
>>> a = np.arange(4)
>>> a + np.array([1, 2])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: operands could not be broadcast together with shapes (4) (2)
```

Broadcasting? We'll return to that later.

Basic reductions

McGill HPC

Computing sums:

```
>>> x = np.array([1, 2, 3, 4])
>>> np.sum(x)
10
>>> x.sum()
10
```

Sum by rows and by columns:

```
>>> x = np.array([[1, 1], [2, 2]])
>>> x
array([[1, 1],
       [2, 2]])
>>> x.sum(axis=0)   # columns (first dimension)
array([3, 3])
>>> x[:, 0].sum(), x[:, 1].sum()
(3, 3)
>>> x.sum(axis=1)   # rows (second dimension)
array([2, 4])
>>> x[0, :].sum(), x[1, :].sum()
(2, 4)
```


Calling (legacy) Fortran code

McGill HPC

Shape-preserving functions with elementwise non-Python routines.

```
! 2_a_fortran_module.f90
subroutine some_function(n, a, b)
  integer :: n
  double precision, dimension(n), intent(in) :: a
  double precision, dimension(n), intent(out) :: b
  b = a + 1
end subroutine some_function
```

We can use f2py to wrap this fortran code in Python:

```
f2py --fcompiler=gfortran -c -m fortran_module 2_a_fortran_module.f90
```

```
import numpy as np
import fortran_module
def some_function(input):
    """
    Call a Fortran routine, and preserve input shape
    """
    input = np.asarray(input)
    # fortran_module.some_function() takes 1-D arrays!
    output = fortran_module.some_function(input.ravel())
    return output.reshape(input.shape)
print some_function(np.array([1, 2, 3])) # [ 2.  3.  4.]
print some_function(np.array([[1, 2], [3, 4]])) # [[ 2.  3.]  [ 4.  5.]]
```

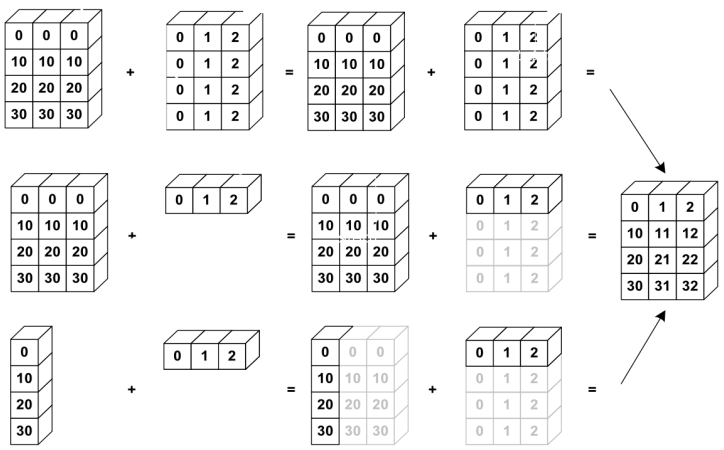

Broadcasting

McGill HPC

Basic operations on numpy arrays (addition, etc.) are elementwise. This works on arrays of the same size.
Nevertheless, it is also possible to do operations on arrays of different sizes if Numpy can transform these arrays so that they all have the same size: this conversion is called broadcasting. Let's verify:

```
>>> a = np.tile(np.arange(0, 40, 10), (3, 1)).T
>>> a
array([[ 0,  0,  0],
       [10, 10, 10],
       [20, 20, 20],
       [30, 30, 30]])
>>> b = np.array([0, 1, 2])
>>> a + b
array([[ 0,  1,  2],
       [10, 11, 12],
       [20, 21, 22],
       [30, 31, 32]])
```

Broadcasting



Broadcasting

A useful trick:

```
>>> a = np.arange(0, 40, 10)
>>> a.shape
(4,)
>>> a = a[:, np.newaxis]  # adds a new axis -> 2D array
>>> a.shape
(4, 1)
>>> a
array([[ 0],
       [10],
       [20],
       [30]])
>>> a + b
array([[ 0,  1,  2],
       [10, 11, 12],
       [20, 21, 22],
       [30, 31, 32]])
```


Example

Let's construct an array of distances (in miles) between cities of Route 66: Chicago, Springfield, Saint-Louis, Tulsa, Oklahoma City, Amarillo, Santa Fe, Albuquerque, Flagstaff and Los Angeles.

```
>>> mileposts = np.array([0, 198, 303, 736, 871, 1175, 1475, 1544,
...        1913, 2448])
>>> distance_array = np.abs(mileposts - mileposts[:, np.newaxis])
>>> distance_array
array([[   0,  198,  303,  736,  871, 1175, 1475, 1544, 1913, 2448],
       [ 198,    0,  105,  538,  673,  977, 1277, 1346, 1715, 2250],
       [ 303,  105,    0,  433,  568,  872, 1172, 1241, 1610, 2145],
       [ 736,  538,  433,    0,  135,  439,  739,  808, 1177, 1712],
       [ 871,  673,  568,  135,    0,  304,  604,  673, 1042, 1577],
       [1175,  977,  872,  439,  304,    0,  300,  369,  738, 1273],
       [1475, 1277, 1172,  739,  604,  300,    0,   69,  438,  973],
       [1544, 1346, 1241,  808,  673,  369,   69,    0,  369,  904],
       [1913, 1715, 1610, 1177, 1042,  738,  438,  369,    0,  535],
       [2448, 2250, 2145, 1712, 1577, 1273,  973,  904,  535,    0]])
```


Summary

What is needed to get started?

- How to create arrays : `array`, `arange`, `ones`, `zeros`.
- Array shapes using `array.shape`
- Use slicing to obtain different views of the array: `array[::2]`, etc.
- Adjust array shape using `reshape` or flatten with `ravel`.
- Obtain a subset of the elements of an array and/or modify their values with masks:

```
>>> a[a < 0] = 0
```

- Know miscellaneous operations on arrays, such as finding the mean or max (`array.max()`, `array.mean()`).
- No need to remember everything, but know how to search documentation (online documentation, `help()`, `lookfor()`).
- Advanced use: indexing with arrays of integers and broadcasting. More Numpy functions to handle various array operations.

Matplotlib

McGill HPC

- Probably the single most used Python package for 2D-graphics.
- provides a quick way to visualize data from Python and publication-quality figures in many formats.

IPython and the pylab mode

- IPython helps the scientific-computing workflow in Python in combination with Matplotlib.
- Start IPython with the command line argument `--pylab`, for interactive matplotlib sessions with Matlab/Mathematica-like functionality.

pylab

- Provides a procedural interface to the matplotlib object-oriented plotting library.
- Closely modeled after Matlab™, so most plotting commands in `pylab` have Matlab™ analogs with similar arguments.

Simple plot

McGill HPC

Draw the cosine and sine functions on the same plot:
First get data for sine and cosine functions:

- `X` is a numpy array with 256 values ranging from $-\pi$ to $+\pi$ (included).
- `C` is the cosine (256 values) and `S` is the sine (256 values).

```
import pylab as pl
import numpy as np

X = np.linspace(-np.pi, np.pi, 256, endpoint=True)
C, S = np.cos(X), np.sin(X)

pl.plot(X, C)
pl.plot(X, S)
pl.show()
```

Matplotlib allows customizing all kinds of properties:
figure size and dpi, line width, color and style, axes, axis and grid properties, text and font properties, and so on.

Scipy

McGill HPC

- Toolbox collection, handling common issues in scientific computing.
- Comparable to the GSL (GNU Scientific Library for C and C++), or Matlab's toolboxes.
- Before implementing a routine, it is worth checking if something similar is already implemented in Scipy.
- The `scipy` module is composed of task-specific sub-modules, such as `stats` for statistics.
- They all depend on `numpy`, but are mostly independent of each other. The standard way of importing Numpy and these Scipy modules is:

```
>>> import numpy as np
>>> from scipy import stats  # same for other sub-modules
```

- The main `scipy` namespace mostly contains functions that are `numpy` functions (try `scipy.cos is np.cos`), exposed for historical reasons only. Usually there is no reason to use plain `import scipy`.

List of SciPy modules

McGill HPC

`scipy.cluster`	Vector quantization / Kmeans
`scipy.constants`	Physical and mathematical constants
`scipy.fftpack`	Fourier transform
`scipy.integrate`	Integration routines
`scipy.interpolate`	Interpolation
`scipy.io`	Data input and output
`scipy.linalg`	Linear algebra routines
`scipy.ndimage`	n-dimensional image package
`scipy.odr`	Orthogonal distance regression
`scipy.optimize`	Optimization
`scipy.signal`	Signal processing
`scipy.sparse`	Sparse matrices
`scipy.spatial`	Spatial data structures and algorithms
`scipy.special`	Any special mathematical functions
`scipy.stats`	Statistics

Further reading:

McGill HPC

- Python, NumPy, SciPy, Matplotlib and Ipython can all be found at the .org website, for instance:
  `http://www.python.org`
- Official non-scientific Python tutorial:
  `http://docs.python.org/2/tutorial`
- Two more scientific tutorials:
  `http://github.com/jrjohansson/scientific-python-lectures`
  `http://python4astronomers.github.io/`
- Followup questions? Contact `guillimin@calculquebec.ca`.
- Slides 66–99: supplementary material that was not directly covered during the workshop.

More on Python: Parameters

McGill HPC

Mandatory parameters (positional arguments)

```
>>> def double_it(x):
...     return x * 2
...

>>> double_it(3)
6
>>> double_it()
------------------------------------------------------------------------
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: double_it() takes exactly 1 argument (0 given)
```

Optional parameters (keyword or named arguments)

```
>>> def double_it(x=2):
...     return x * 2
...

>>> double_it()
4
>>> double_it(3)
6
```


Parameters

McGill HPC

Keyword arguments allow you to specify **default values**.
Warning: Default values are evaluated when the function is defined, not when it is called. This can be problematic when using mutable types (e.g. dictionary or list): modifications will be persistent across invocations of the function; typically `None` is used instead.

```
>>> bigx = 10

>>> def double_it(x=bigx):
...     return x * 2
...

>>> bigx = 1e9  # Now really big

>>> double_it()
20
```

Keyword arguments are a very convenient feature for defining functions with a variable number of arguments, especially when default values are to be used in most calls to the function.

Slicing parameters

McGill HPC

More involved example implementing python's slicing:

```
>>> def slicer(seq, start=None, stop=None, step=None):
...     """Implement basic python slicing."""
...     return seq[start:stop:step]
...

>>> rhyme = 'one fish, two fish, red fish, blue fish'.split()
>>> rhyme
['one', 'fish,', 'two', 'fish,', 'red', 'fish,', 'blue', 'fish']
>>> slicer(rhyme)
['one', 'fish,', 'two', 'fish,', 'red', 'fish,', 'blue', 'fish']
>>> slicer(rhyme, step=2)
['one', 'two', 'red', 'blue']
>>> slicer(rhyme, 1, step=2)
['fish,', 'fish,', 'fish,', 'fish']
>>> slicer(rhyme, start=1, stop=4, step=2)
['fish,', 'fish,']
```

The order of the keyword arguments does not matter:

```
>>> slicer(rhyme, step=2, start=1, stop=4)
['fish,', 'fish,']
```

good practice is to use the same ordering as the function's definition.

Value passing

McGill HPC

Parameters to functions are references to objects, which are passed by value.
Functions:

- **cannot** modify immutable values that are passed in.
- **can** modify mutable values that are passed in.
- have a local variable table called a **local namespace**.

The variable `x` only exists within the function `try_to_modify`.

```
>>> def try_to_modify(x, y, z):
...     x = 23
...     y.append(42)
...     z = [99] # new reference
...     print(x)
...     print(y)
...     print(z)
...
>>> a = 77    # immutable variable
>>> b = [99]  # mutable variable
>>> c = [28]
>>> try_to_modify(a, b, c)
23
[99, 42]
[99]
>>> print(a)
77
>>> print(b)
[99, 42]
>>> print(c)
[28]
```


Global variables

McGill HPC

Functions can reference outside variables declared outside them:

```
>>> x = 5
>>> def addx(y):
...     return x + y
...

>>> addx(10)
15
```

Functions can only modify such variables if declared `global` in function.

This doesn't work:

```
>>> def setx(y):
...     x = y
...     print('x is %d' % x)
...
...

>>> setx(10)
x is 10

>>> x
5
```

This works:

```
>>> def setx(y):
...     global x
...     x = y
...     print('x is %d' % x)
...
...

>>> setx(10)
x is 10
>>> x
10
```


Variable number of parameters

McGill HPC

Special forms of parameters:
`*args`: any number of positional arguments packed into a tuple
`**kwargs`: any number of keyword arguments packed into a dictionary

```
>>> def variable_args(*args, **kwargs):
...     print 'args is', args
...     print 'kwargs is', kwargs
...

>>> variable_args('one', 'two', x=1, y=2, z=3)
args is ('one', 'two')
kwargs is {'y': 2, 'x': 1, 'z': 3}
```


More on functions and methods

McGill HPC

Functions are first-class objects, which means they can be:

- assigned to a variable
- an item in a list (or any collection)
- passed as an argument to another function.

```
>>> va = variable_args

>>> va('three', x=1, y=2)
args is ('three',)
kwargs is {'y': 2, 'x': 1}
```

Methods are functions attached to objects, as seen in examples on lists, dictionaries, strings, etc...

Importing objects from modules

McGill HPC

```
>>> import os

>>> os
<module 'os' from '/software/.../lib/python2.7/os.pyc'>

>>> os.listdir('.')
['hello.py', '2_a_fortran_module.f90', '2_a_call_fortran.py',
 'populations.txt', 'demo.py', 'demo2.py', 'solutions']
```

To only import the `listdir` function without `os.` in front:

```
>>> from os import listdir
```

Import `numpy` but call it `np` in this context:

```
>>> import numpy as np
```


Star imports

McGill HPC

```
>>> from os import *
```

Warning: use with caution ("namespace pollution")

- Code is harder to read and understand: symbols come from where?
- Impossible to guess functionality by context and name (hint: `os.name` is the name of the OS), and to profit from tab completion.
- Restricts the variable names you can use: `os.name` might override `name`, or vice-versa.
- Creates possible name clashes between modules.
- Removes possibility to check for undefined symbols.

Modules are thus a good way to organize code in a hierarchical way.
All used scientific computing tools are modules:

```
>>> import numpy as np # data arrays
>>> np.linspace(0, 10, 6)
array([  0.,   2.,   4.,   6.,   8.,  10.])
>>> import scipy # scientific computing
```


Creating modules

McGill HPC

For larger and better organized programs, where some objects are defined, (variables, functions, classes) that are reused, create our own modules.
Create a module demo contained in the file `demo.py`

```
"A demo module."

def print_b():
    "Prints b."
    print 'b'
```

```
def print_a():
    "Prints a."
    print 'a'

c=2
d=2
```

This file contains the two functions `print_a` and `print_b`. Suppose we want to call the print_a function from the interpreter. We could execute the file as a script, but since we just want to have access to the function `print_a`, instead it is **imported as a module**, using

```
>>> import demo

>>> demo.print_a()
a

>>> demo.print_b()
b
```


Importing objects from modules

McGill HPC

```
In [6]: from demo import print_a, print_b

In [7]: whos
Variable   Type        Data/Info
-------------------------------
demo       module      <module 'demo' from 'demo.py'>
print_a    function    <function print_a at 0xb7421534>
print_b    function    <function print_b at 0xb74214c4>

In [8]: print_a()
a
```

Warning: Modules are cached: if you modify demo.py and re-import it in the old session, you will get the old one.
Solution:

```
In [9]: reload(demo)
```

'__main__' and module loading

McGill HPC

File demo2.py:

```
import sys
def print_a():
    "Prints a."
    print 'a'

# print command-line arguments
print sys.argv

if __name__ == '__main__':
    print_a()
```

Importing it:

```
In [10]: import demo2
['/software/CentOS-5/.../ipython']

In [11]: import demo2
```

Running it:

```
#$ python demo2.py test arguments
In [12]: %run demo2 test arguments
['demo2.py', 'test', 'arguments']
a
```

Scripts or modules? How to organize your code:
Sets of instructions called several times should be written inside **functions** for better code reusability.
Functions (or other bits of code) called from several scripts should be written inside **modules**, so that only the module is imported in the different scripts (do not copy-and-paste your functions!).
Note about sys.argv: Don't implement option parsing yourself. Use modules such as optparse or argparse.

How modules are found and imported

McGill HPC

When the `import mymodule` statement is executed, the module `mymodule` is searched in a given list of directories.
This list includes a list of installation-dependent default path (e.g., `/usr/lib/python`) as well as the list of directories specified by the environment variable `PYTHONPATH`.
The `sys.path` variable gives the list of directories searched by Python.

```
>>> import sys

>>> sys.path
['',
 '/home/classxx/.local/lib/python2.7/site-packages',
 '/software/CentOS-5/toools/python-2.7.3/lib/python2.7/dist-packages',
 ...]
```


Packages

McGill HPC

A package is a directory that contains many modules. A package is a module with submodules (which can have submodules themselves, etc.).
A special file called __init__.py (which may be empty) tells Python that the directory is a Python package, from which modules can be imported.

```
$ ls /software/CentOS-5/tools/python-2.7.3/lib/python2.7/site-packages/scipy
... __init__.py@    maxentropy/  signal/         version.py@
    __init__.pyc    misc/        sparse/         version.pyc ...
$ cd ndimage
$ ls
... filters.pyc  __init__.py@  measurements.pyc   _ni_support.pyc  tests/
    fourier.py@  __init__.pyc  morphology.py@     setup.py@ ...
```

Packages from Ipython

```
In [1]: import scipy
In [2]: scipy.__file__
Out[2]:'/software/.../__init__.pyc'
In [3]: import scipy.version
In [4]: scipy.version.version
Out[4]: '0.10.1'
In [5]: from scipy.ndimage import morphology
In [6]: morphology.binary_dilation?
Type:           function ...
```


Input and Output

McGill HPC

Can write or read strings to/from files (other types must be converted to strings). To write to a file:

```
>>> f = open('workfile', 'w') # opens the workfile file
>>> type(f)
<type 'file'>
>>> f.write('This is a test \nand another test')
>>> f.close()
```

To read from a file

```
>>> f = open('workfile', 'r')

>>> s = f.read()

>>> print(s)
This is a test
and another test

>>> f.close()
```

There also exist Numpy methods to read and write files.

Iterating over a file

```
>>> f = open('workfile', 'r')

>>> for line in f:
...     print line
...
...
This is a test

and another test

>>> f.close()
```

File modes

- Read-only: `r`
- Write-only: `w` — Creates a new file or overwrite existing file.
- Append a file: `a`
- Read and Write: `r+`
- Binary mode: `b` — Use for binary files, especially on Windows.

Environment variables

```
>>> import os
>>> os.environ.keys()
['SSH_ASKPASS', 'HISTTIMEFORMAT', 'MODULE_VERSION', 'LESSOPEN',
'SSH_CLIENT', 'CPATH', ...
>>> os.environ['PYTHONPATH']
'.:/home/classxx/local/lib/python2.7/site-packages/:
/software/CentOS-5/python2.7/site-packages/'
>>> os.getenv('PYTHONPATH')
'.:/home/classxx/local/lib/python2.7/site-packages/:
/software/CentOS-5/python2.7/site-packages/'
```

sys module: system-specific information

System-specific information related to the Python interpreter.

Which version of python are you running and where is it installed:

```
>>> sys.platform
'linux2'
>>> sys.version
'2.7.3 (default, Sep 17 2012, 10:35:59) \n
[GCC 4.1.2 20080704 (Red Hat 4.1.2-50)]'
>>> sys.prefix
'/software/CentOS-5/tools/python-2.7.3'
```


Exception handling in Python

Exceptions are raised by different kinds of errors. Such errors can be caught, or custom error types can be defined.

```
>>> 1/0
ZeroDivisionError: integer division or modulo by zero

>>> 1 + 'e'
TypeError: unsupported operand type(s) for +: 'int' and 'str'

>>> d = {1:1, 2:2}
>>> d[3]
KeyError: 3

>>> l = [1, 2, 3]
>>> l[4]
IndexError: list index out of range

>>> l.foobar
AttributeError: 'list' object has no attribute 'foobar'
```

There exist **different types** of exceptions for different errors.

Catching exceptions

try/except

```
>>> while True:
...     try:
...         x = int(raw_input('Please enter a number: '))
...         break
...     except ValueError:
...         print('That was no valid number.  Try again...')
...
...
Please enter a number: a
That was no valid number.  Try again...
Please enter a number: 1

>>> x
1
```

Raising exceptions

McGill HPC

Capturing and reraising an exception:

```
>>> def filter_name(name):
...     try:
...         name = name.encode('ascii')
...     except UnicodeError as e:
...         if name == 'Gaël':
...             print('OK, Gaël')
...         else:
...             raise e
...     return name
...

>>> filter_name('Gaël')
OK, Gaël
'Ga\xc3\xabl'

>>> filter_name('Stéfan')
---------------------------------------------------------------------------
UnicodeDecodeError: 'ascii' codec can't decode byte 0xc3 in
position 2: ordinal not in range(128)
```


Object-oriented programming (OOP)

McGill HPC

Python supports object-oriented programming (OOP). Goals of OOP:

- to organize the code, and
- to re-use code in similar contexts.

Example: a Student **class**: an object gathering several custom functions (**methods**) and variables (**attributes**)

```
>>> class Student(object):
...     def __init__(self, name):
...         self.name = name
...     def set_age(self, age):
...         self.age = age
...     def set_major(self, major):
...         self.major = major
...
>>> anna = Student('anna')
>>> anna.set_age(21)
>>> anna.set_major('physics')
```

The Student class has `__init__`, `set_age` and `set_major` methods. Its attributes are `name`, `age` and `major`.

OOP

McGill HPC

Call methods and attributes using `classinstance.method` or `classinstance.attribute`.
`__init__` constructor: special method called from `MyClass(init parameters if any)`.
Consider new class MasterStudent: same as Student class, but with additional `internship` attribute. Do not copy and add, but inherit:

```
>>> class MasterStudent(Student):
...     internship = 'mandatory, from March to June'
...
>>> james = MasterStudent('james')
>>> james.internship
'mandatory, from March to June'
>>> james.set_age(23)
>>> james.age
23
```

Code can be organized with classes corresponding to different objects (Experiment class, Flow class, etc.). Then use inheritance to re-use code. E.g., from a Flow base class, derive StokesFlow, TurbulentFlow, etc.

More NumPy: Transpose and Linear Algebra

McGill HPC

```
>>> a = np.triu(np.ones((3, 3)), 1)   # see help(np.triu)
>>> a
array([[ 0.,  1.,  1.],
       [ 0.,  0.,  1.],
       [ 0.,  0.,  0.]])
>>> a.T
array([[ 0.,  0.,  0.],
       [ 1.,  0.,  0.],
       [ 1.,  1.,  0.]])
```

Basic **Linear algebra** is provided by the sub-module numpy.linalg.

```
>>> a = np.diag((2.,3.,4.))
>>> np.linalg.det(a) # determinant
23.999999999999993
>>> np.linalg.eig(a) # eigenvalues
(array([ 2.,  3.,  4.]),
array([[ 1.,  0.,  0.],
       [ 0.,  1.,  0.],
       [ 0.,  0.,  1.]]))
```

Other uses: solving linear systems, singular value decomposition, etc.
Alternatively, use the more extensive `scipy.libalg` module.

Basic reductions in higher dimensions

McGill HPC

```
>>> x = np.random.rand(2, 2, 2)
>>> x.sum(axis=2)[0, 1]
1.14764...
>>> x[0, 1, :].sum()
1.14764...
```

Other reductions — works the same way (and take axis=)

Statistics:

```
>>> x = np.array([1, 2, 3, 1])
>>> y = np.array([[1, 2, 3], [5, 6, 1]])
>>> x.mean()
1.75
>>> np.median(x)
1.5
>>> np.median(y, axis=-1) # last axis
array([ 2.,  5.])

>>> x.std()          # full population standard dev.
0.82915619758884995
```


More basic reductions

McGill HPC

Extrema:

```
>>> x = np.array([1, 3, 2])
>>> x.min()
1
>>> x.max()
3

>>> x.argmin()  # index of minimum
0
>>> x.argmax()  # index of maximum
1
```

Logical operations:

```
>>> np.all([True, True, False])
False
>>> np.any([True, True, False])
True
```

Comparing arrays:

```
>>> a = np.zeros((100, 100))
>>> np.any(a != 0)
False
>>> np.all(a == a)
True

>>> a = np.array([1, 2, 3, 2])
>>> b = np.array([2, 2, 3, 2])
>>> c = np.array([6, 4, 4, 5])
>>> ((a <= b) & (b <= c)).all()
True
```

... and many more (best to learn as you go).

Example: data statistics

McGill HPC

Data in populations.txt describes the populations of hares, lynxes, and carrots in northern Canada during 20 years. First plot the data:

```
>>> data = np.loadtxt('populations.txt')
>>> year, hares, lynxes, carrots = data.T  # trick: columns to variables
>>> from matplotlib import pyplot as plt
>>> plt.axes([0.2, 0.1, 0.5, 0.8])
<matplotlib.axes.Axes object at ...>
>>> plt.plot(year, hares, year, lynxes, year, carrots)
[<matplotlib.lines.Line2D object at ...>, ...]
>>> plt.legend(('Hare', 'Lynx', 'Carrot'), loc=(1.05, 0.5))
<matplotlib.legend.Legend object at ...>
```

The mean populations over time:

```
>>> populations = data[:, 1:]
>>> populations.mean(axis=0)
array([ 34080.95238095,  20166.66666667,  42400.        ])
```

The sample standard deviations:

```
>>> populations.std(axis=0)
array([ 20897.90645809,  16254.59153691,   3322.50622558])
>>> # Which species has the highest population each year?:
>>> np.argmax(populations, axis=1)
array([2, 2, 0, 0, 1, 1, 2, 2, 2, 2, 2, 2, 0, 0, 0, 1, 2, 2, 2, 2, 2])
```


Array shape manipulation

McGill HPC

Flattening

```
>>> a = np.array([[1, 2, 3], [4, 5, 6]])
>>> a.ravel()
array([1, 2, 3, 4, 5, 6])
>>> a.T
array([[1, 4],
       [2, 5],
       [3, 6]])
>>> a.T.ravel()
array([1, 4, 2, 5, 3, 6])
```

Higher dimensions: last dimensions ravel out "first".

Reshaping: the inverse operation to flattening:

```
>>> a.shape
(2, 3)
>>> b = a.ravel()
>>> b.reshape((2, 3))
array([[1, 2, 3],
       [4, 5, 6]])
```

Or,

```
>>> c = b.reshape((2, -1))    # unspecified (-1) value is inferred
```

Reshape returns a view or a copy

McGill HPC

Warning `ndarray.reshape` may return view (see `help(np.reshape)`)

```
>>> c[0, 0] = 99
>>> b # returns a view
array([99,  1,  2,  3,  4,  5,  6])
```

But reshape may also return a copy!

```
>>> a = np.zeros((3, 2))
>>> b = a.T.reshape(3*2)
>>> b[0] = 9
>>> a
array([[ 0.,  0.],
       [ 0.,  0.],
       [ 0.,  0.]])
```

Dimension shuffling

```
>>> a = np.arange(4*3*2).reshape(4, 3, 2)
>>> a.shape
(4, 3, 2)
>>> a[0, 2, 1]
5
```

```
>>> b = a.transpose(1, 2, 0)
>>> b.shape
(3, 2, 4)
>>> b[2, 1, 0]
5
```

Also creates a view:

```
>>> b[2, 1, 0] = -1
>>> a[0, 2, 1]
-1
```


Resizing

McGill HPC

Size of an array can be changed with ndarray.resize:

```
>>> a = np.arange(4)
>>> a.resize((8,))
>>> a
array([0, 1, 2, 3, 0, 0, 0, 0])
```

However, it must not be referred to somewhere else:

```
>>> b = a
>>> a.resize((4,))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: cannot resize an array that has been referenced or is
referencing another array in this way.  Use the resize function
```


Sorting data

McGill HPC

Sorting along an axis:

```
>>> a = np.array([[4, 3, 5], [1, 2, 1]])
>>> b = np.sort(a, axis=1)
>>> b
array([[3, 4, 5],
       [1, 1, 2]])
```

Sorts each row separately!

In-place sort:

```
>>> a.sort(axis=1)
>>> a
array([[3, 4, 5],
       [1, 1, 2]])
```

Sorting with fancy indexing:

```
>>> a = np.array([4, 3, 1, 2])
>>> j = np.argsort(a)
>>> j
array([2, 3, 1, 0])
>>> a[j]
array([1, 2, 3, 4])
```


Exercise 7: Mandelbrot set

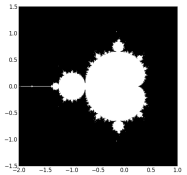
McGill HPC

Write a script that computes the Mandelbrot fractal, using this iteration:

```
N_max = 50
some_threshold = 50

c = x + 1j*y

for j in xrange(N_max):
    z = z**2 + c
```



Point (x, y) belongs to the Mandelbrot set if `|c| < some_threshold`.

Do this computation as follows:

1. Construct a grid of c = x + 1j*y values in range [-2, 1] x [-1.5, 1.5]
2. Do the iteration
3. Form the 2-d boolean mask indicating which points are in the set
4. Save the result to an image with:

```
>>> import matplotlib.pyplot as plt
>>> plt.imshow(mask.T, extent=[-2, 1, -1.5, 1.5])
<matplotlib.image.AxesImage object at ...>
>>> plt.gray()
>>> plt.savefig('mandelbrot.png')
```

Loading data files

McGill HPC

Text files, example: populations.txt

```
>>> data = np.loadtxt('populations.txt')
>>> data
array([[  1900.,  30000.,   4000.,  48300.],
       [  1901.,  47200.,   6100.,  48200.],
       [  1902.,  70200.,   9800.,  41500.],
...
>>> np.savetxt('pop2.txt', data)
>>> data2 = np.loadtxt('pop2.txt')
```

If you have a complicated text file, you can try:

- np.genfromtxt
- Using Python's I/O functions and e.g. regexps for parsing (Python is quite well suited for this)

Navigating the filesystem with IPython

```
In [1]: pwd      # show current directory
'/home/classxx
In [2]: cd ex
'/home/classxx/ex
In [3]: ls
populations.txt  species.txt
```


Loading data files: Images

McGill HPC

Using Matplotlib:

```
>>> img = plt.imread('data/elephant.png')
>>> img.shape, img.dtype
((200, 300, 3), dtype('float32'))
>>> plt.imshow(img)
<matplotlib.image.AxesImage object at ...>
>>> plt.savefig('plot.png')
>>> plt.imsave('red_elephant', img[:,:,0], cmap=plt.cm.gray)
```

This saved only one channel (of RGB):

```
>>> plt.imshow(plt.imread('red_elephant.png'))
<matplotlib.image.AxesImage object at ...>
```

Other libraries:

```
>>> from scipy.misc import imsave
>>> imsave('tiny_elephant.png', img[::6,::6])
>>> plt.imshow(plt.imread('tiny_elephant.png'), interpolation='nearest')
<matplotlib.image.AxesImage object at ...>
```

Numpy has its own binary format, not portable but with efficient I/O:

```
>>> data = np.ones((3, 3))
>>> np.save('pop.npy', data)
>>> data3 = np.load('pop.npy')
```


Well-known (& more obscure) file formats

McGill HPC

- HDF5: h5py, PyTables
- NetCDF: `scipy.io.netcdf_file`, netcdf4-python, ...
- Matlab: `scipy.io.loadmat`, `scipy.io.savemat`
- MatrixMarket: `scipy.io.mmread`, `scipy.io.mmread`

... if somebody uses it, there's probably also a Python library for it.

Exercise 8: Text data files

Write a Python script that loads data from `populations.txt` and drops the last column and the first 5 rows. Save the smaller dataset to `pop2.txt`.