Introduction to Python

Lecture 2

Agostino Merico

05 Fenbruary 2018

Python

- open source, general-purpose, programming language
- object-oriented, procedural, functional
- easy to interface with C / Java / Fortran
- easy-ish to interface with C++ (via SWIG)
- great interactive environment
- several versions exist, but main differences are between python 2.x and python 3.x

- downloads: http://www.python.org
- documentation: http://www.python.org/doc/
- free book: http://www.diveintopython.org

Technical aspects:

Installing & Running Python

Binaries

- python comes pre-installed with Mac OSX and Linux
- windows binaries from http://python.org/

- you might not have to do anything !

Dedicated distribution

But different python *distributions* exists !

People have written many python programmes (also known as *scripts*, *packages*, *module*, *libraries*, etc.) over the years and made them available to the broad community;

Thus, a python distirbution is a particular collection of programmes, which may also come with an *editor* or some other types of utilities;

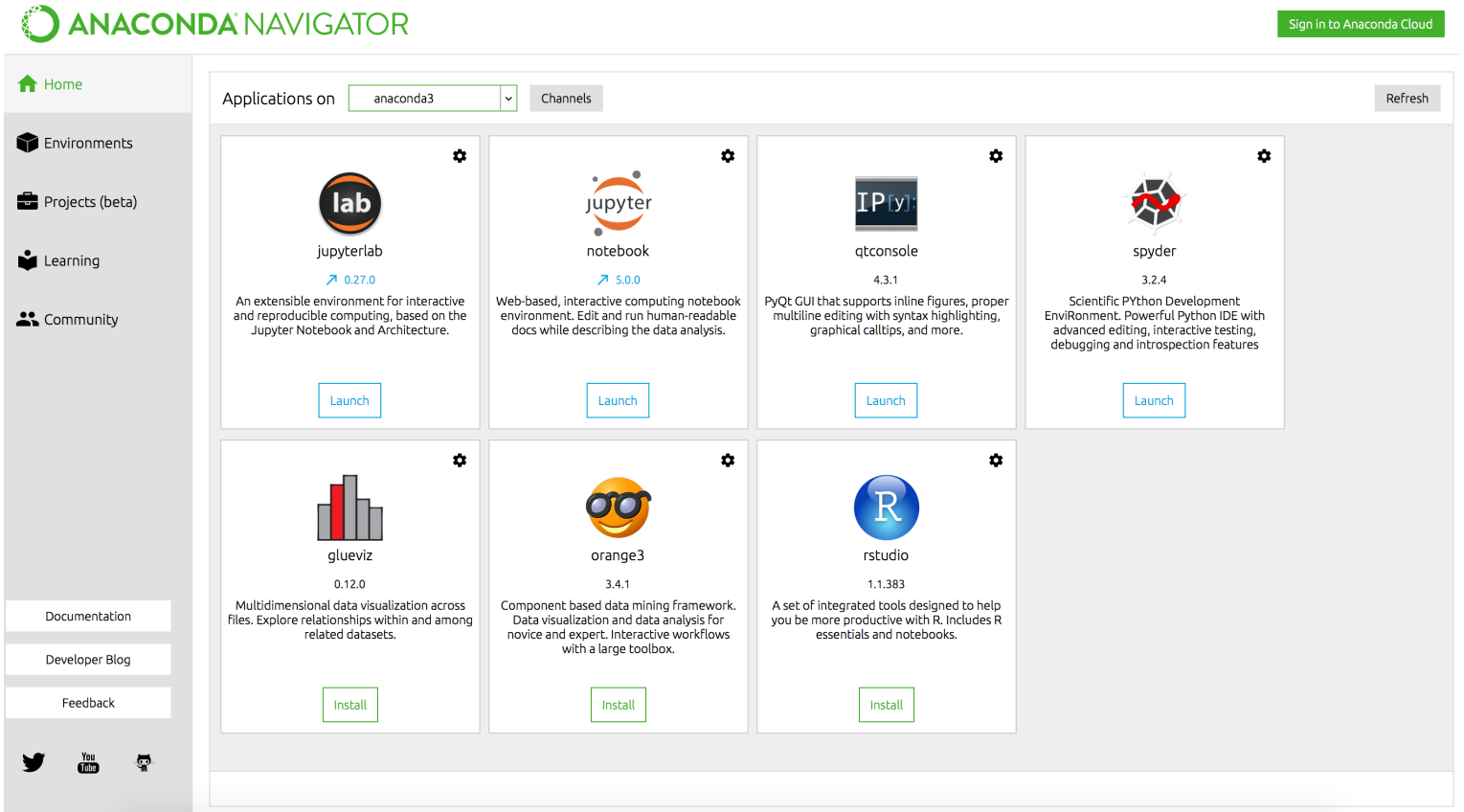
We are most interested in distributions that contains scientific packages (such as `numpy` and `scipy`) or plotting packages (such as `matplotlib`);

Two major distributions are **Enthought Canopy** and **Anaconda**;

They are both freely available (although Enthought is free only for students and academics) but in my view Anaconda offers better functionalities and does not require registration.

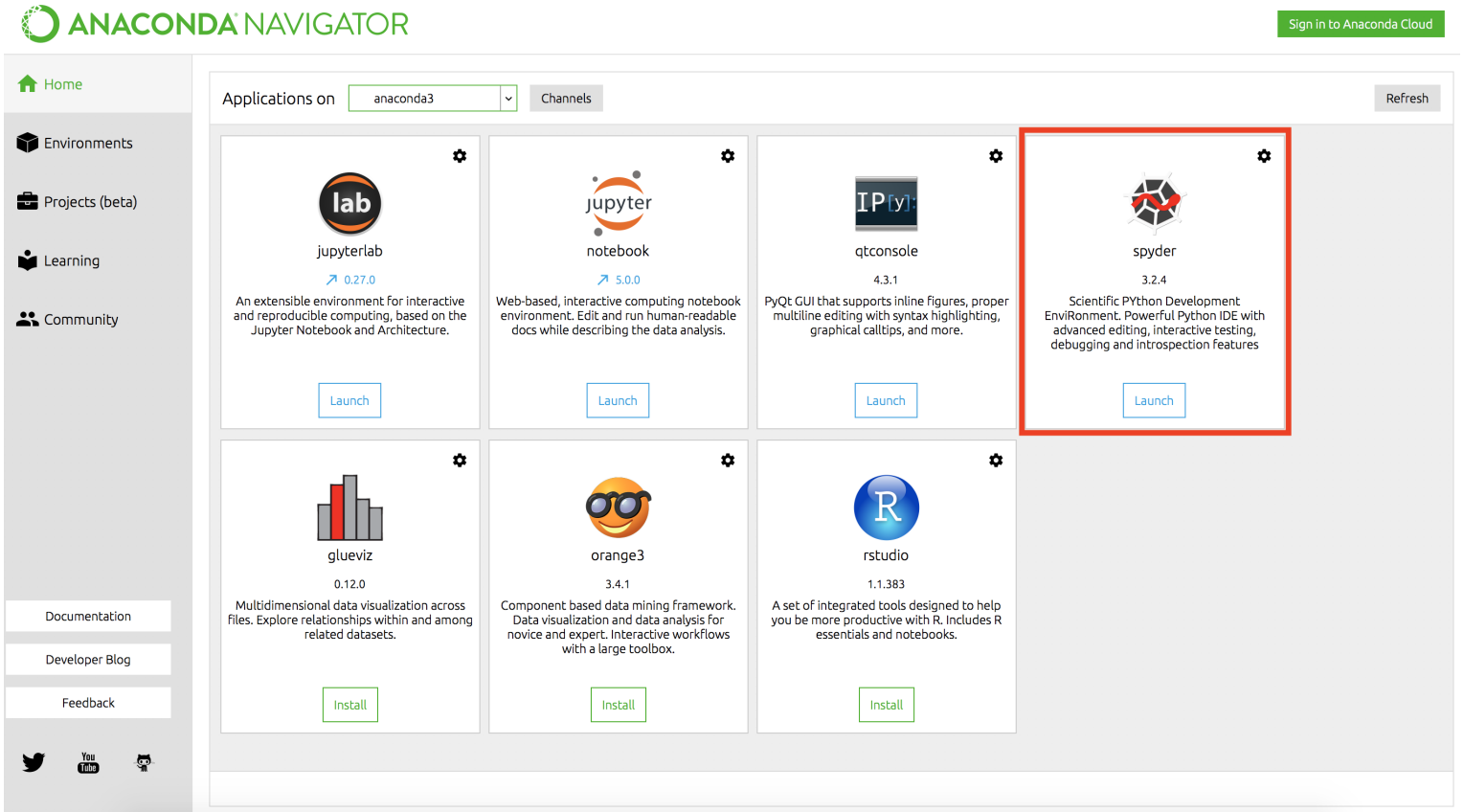
Anaconda distribution

Main anaconda navigator panel...



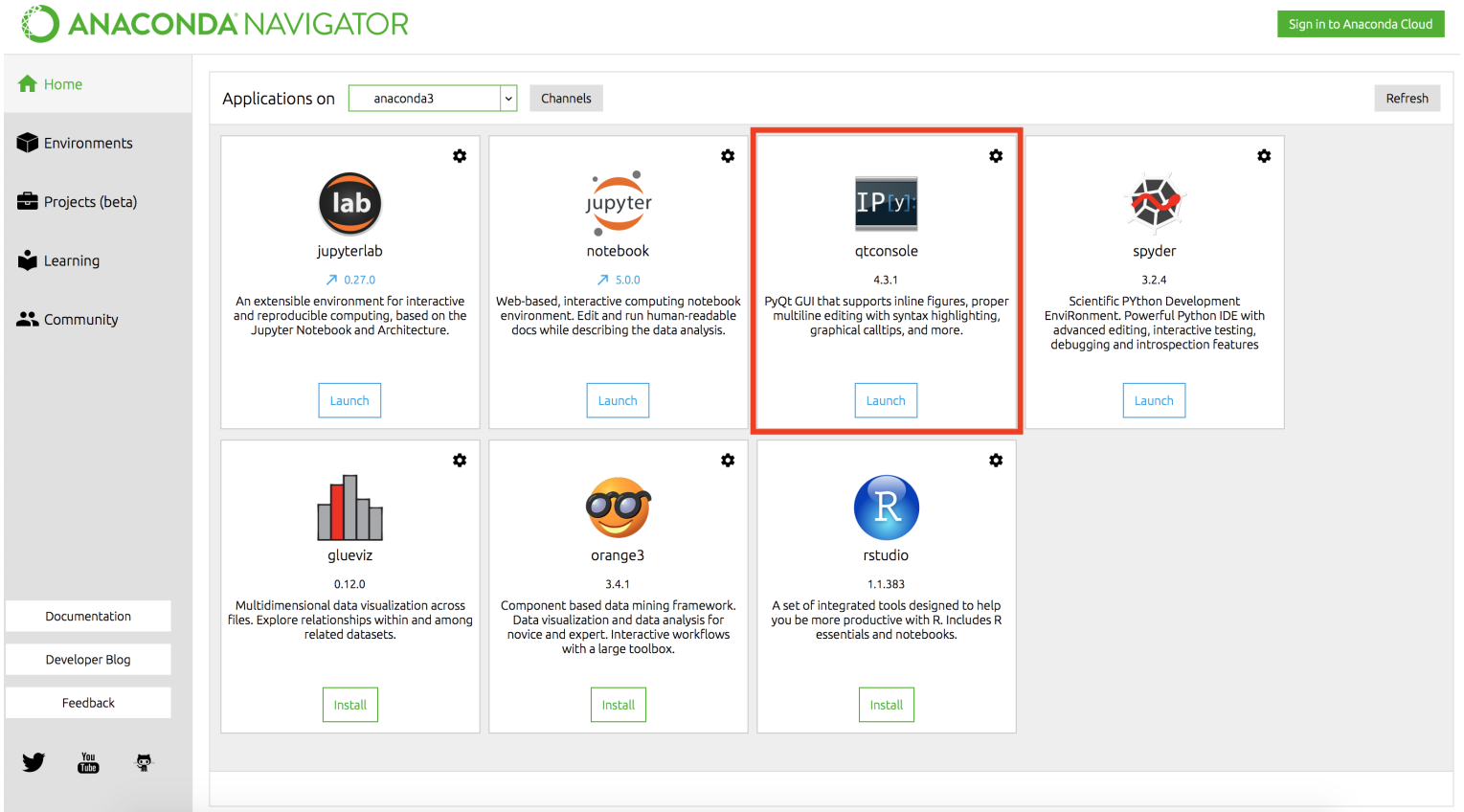
Anaconda distribution

Main anaconda navigator panel...



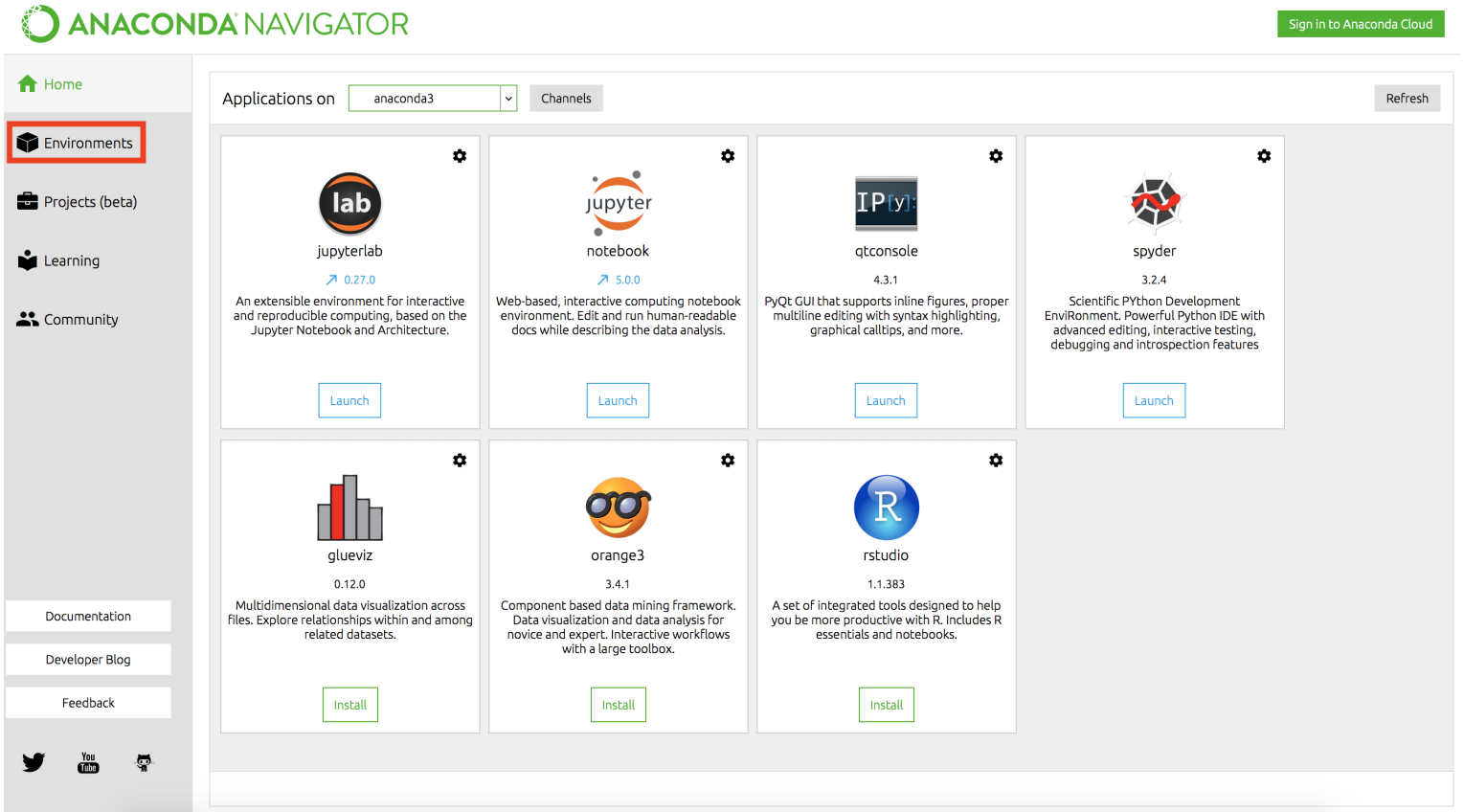
Anaconda distribution

Main anaconda navigator panel...



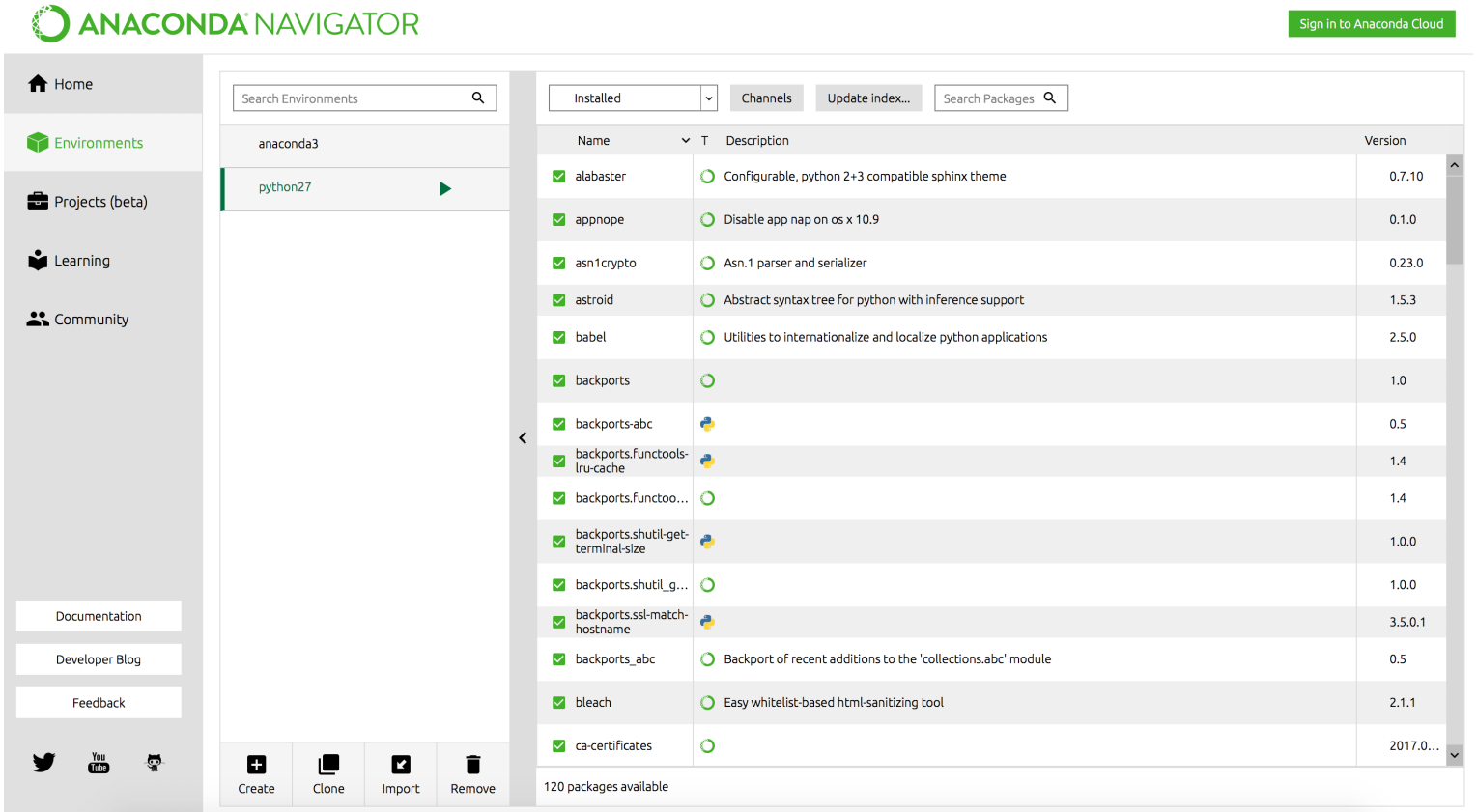
Anaconda distribution

Main anaconda navigator panel...



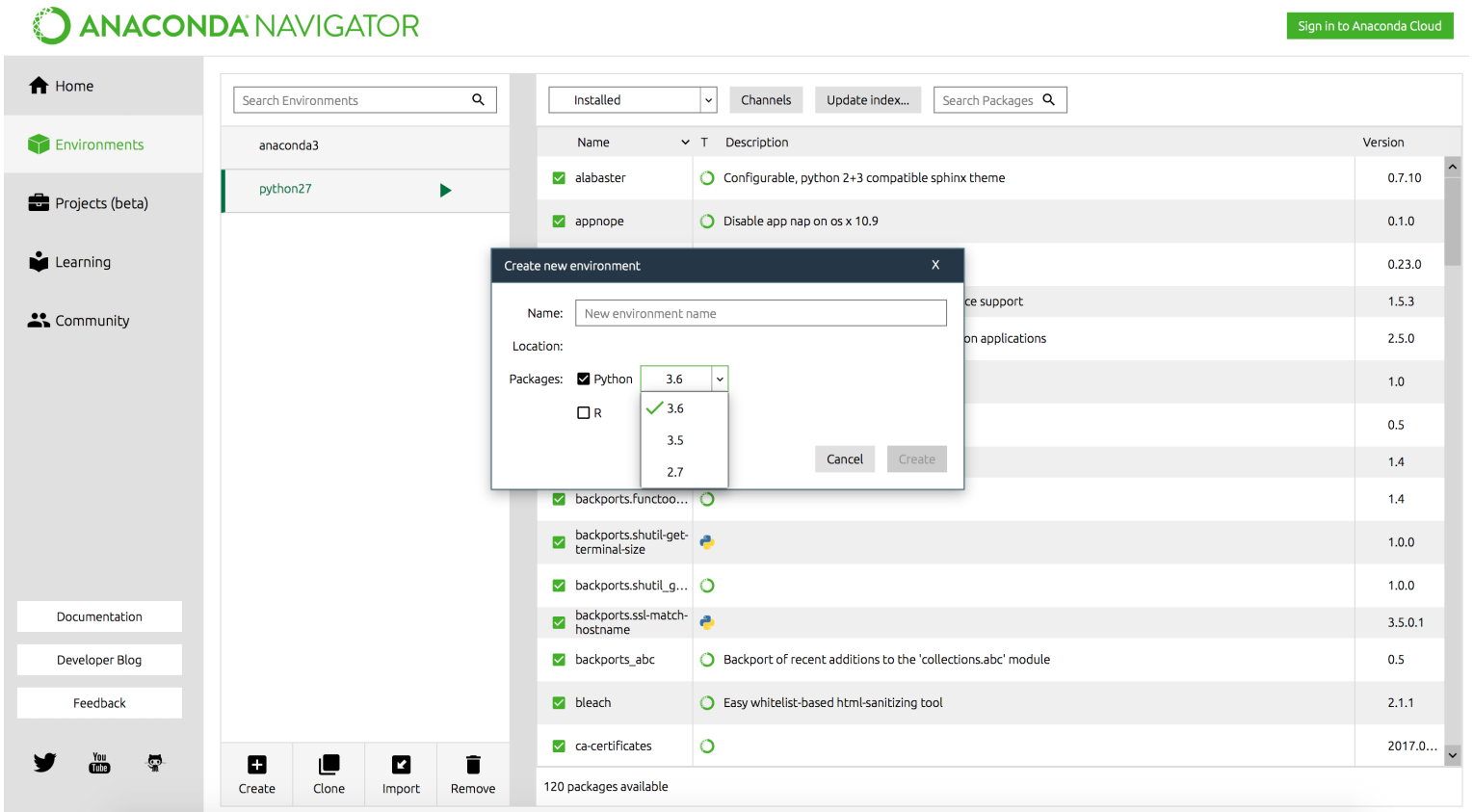
Anaconda environments

Creating new environments...



Anaconda environments

Creating new environments...



The Python Interpreter

- interactive interface to python

```
zmtnb0002:examples ago$ python
Python 2.7.10 (default, Jul 14 2015, 19:46:27)
[GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.0.39)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

- python interpreter evaluates inputs:

```
>>> 3*(7+2)
27
```

- python prompts with:

```
>>>
```

- to exit python:

```
CTRL-D
```

Running programmes

a programme called for example 'filename.py' can be run as follows:

```
zmtnb0002:examples ago$ python filename.py
```

you could make filename.py executable by adding the following line to the top of the file:

```
#!/usr/bin/env python
```

Many packages included

- large collection of proven modelus are included in differenrent distributions (e.g. `numpy`, `matplotlib`, etc.)

  http://docs.python.org/modindex.html

numpy

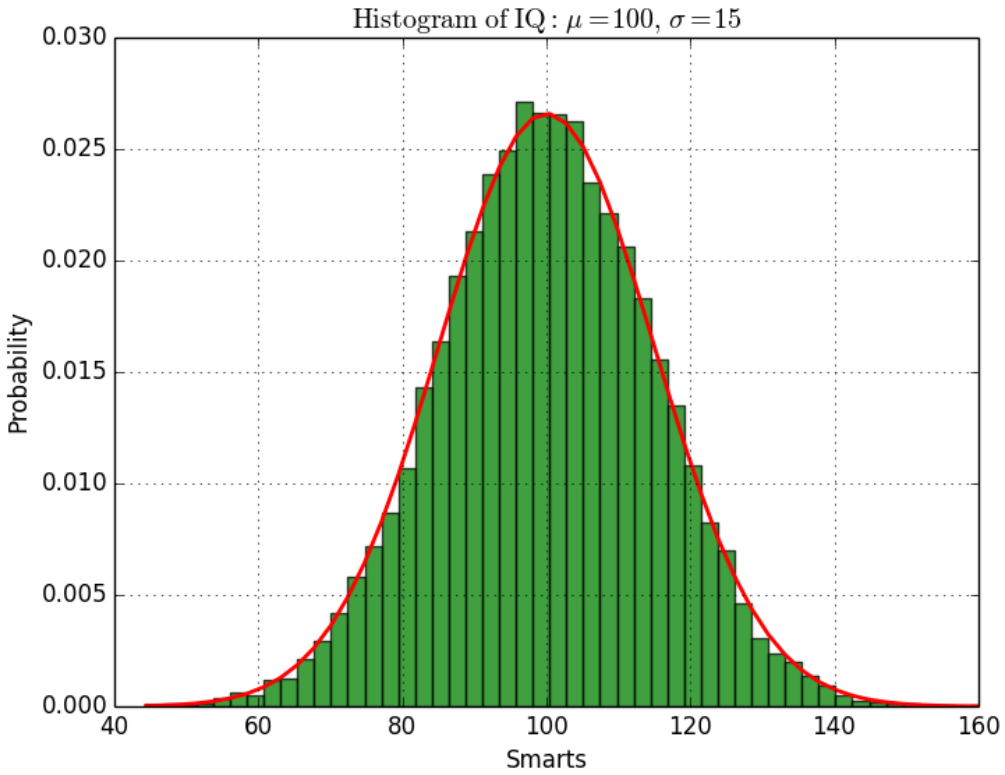
- **numpy** is the fundamental package for scientific computing with python;
- offers Matlab-ish capabilities;
- fast array operations;
- 2D arrays, multi-D arrays, linear algebra, etc.

- tutorial: http://www.scipy.org/Tentative_NumPy_Tutorial

matplotlib

- high-quality plotting library

```
#!/usr/bin/env python
import numpy as np
import matplotlib.mlab as mlab
import matplotlib.pyplot as plt

mu, sigma = 100, 15
x = mu + sigma*np.random.randn(10000)

# the histogram of the data
n, bins, patches = plt.hist(x, 50, normed
      =1, facecolor='green',alpha=0.75)

# add a 'best fit' line
y = mlab.normpdf(bins, mu, sigma)
l = plt.plot(bins, y, 'r-', linewidth=2)

plt.xlabel('Smarts')
plt.ylabel('Probability')
plt.title(r'$\mathrm{Histogram of IQ:} \
      mu=100, \sigma=15$')
plt.axis([40, 160, 0, 0.03])
plt.grid(True)

plt.show()
```



More packages

- ipython, a command shell for interactive computing:
  http://ipython.org
- scipy, a collection of scientific tools:
  http://scipy.org
- sympy, a library for symbolic mathematics
  http://www.sympy.org/de/index.html

Custom distributions

- **python(x,y)**: http://www.pythonxy.com/
  - python(x,y) is a free scientific and engineering development software for numerical computations, data analysis and visualisation

- **sage**: http://www.sagemath.org/
  - sage is a free mathematics software that combines many existing open-source packages into a common python-based interface.

The Basics

A code sample

```
#!/usr/bin/env python
x=34-23      # This is a comment
y='Hello'    # This is another comment
z=3.45
if z==3.45 or y=='Hello':
     x=x+1
     y=y+'World'    # Concatenate string
print x
print y
```

Understanding the code

- assignment uses `=` and comparison uses `==`
- for numbers `+ - * / %` are as expected
  - special use of `+` for string concatenation
  - special use of `%` for string formatting
- logical operators are words (`and`, `or`, `not`,) *not* symbols
- the basic printing command is `print`
- the first assignment to a variable creates it
  - variables don't need to be declared
  - python figures out the variable types on its own

Basic datatypes

- integers (default for numbers)
  `z = 5 / 2` the result is 2, integer division
- floats
  `x = 3.456`
- strings
  - can use " " or ' ' to specify:
    `"abc"` and `'abc'` (are the same things)
  - unmatched can occur within the string:
    `"matt's"`
  - use triple double-quotes for multi-line strings or strings that contain ' and " inside them:
    `""" "a'b"c" """`

Whitespace

Whitespace is meaningful in python: especially indentation and placement of newlines!

- use a newline to end a line of code
  - use \ when must go to next line prematurely
- no braces { } to mark blocks of code in python, use consistent indentation instead
  - the first line with *less* indentation is outside of the block
  - the first line with *more* indentation starts a nested block
- often a colon (:) appears at the start of a new block (e.g. for function and class definitions)

Whitespace

- starts comments with #, the rest of the line will be ignored
- can include a "documentation string" as the first line of any new function or class that you define
- the development environment, debugger, and other tools use it: it's good style to include one

```
def my_function(x, y):
    """This is the docstring. This function does blah blah blah."""
    # The code would go here...
```

Assignment

- binding a variable in python means setting a *name* to hold a *reference* to some *object*
  - assignment creates references, not copies
- names in python do not have an intrinsic type, objects have types
  - python determines the type of the reference automatically based on the data object assigned to it
- you create a name the first time it appears on the left side of an assignment expression: `x = 3`
- a reference is deleted after any names bound to it have passed out of scope

Accessing non-existent names

- if you try to access a name before it has been properly created (by placing it on the left side of an assignment), you'll get an error

```
>>> y
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'y' is not defined

>>> y = 3
>>> y
3
```

Multiple assignment

- you can also assign to multiple names at the same time

```
>>> x, y = 2, 3

>>> x
2

>>> y
3
```

Naming rules

- names are case sensitive and cannot start with a number, they can contain letters, numbers, and anderscores:

  ```
  bob Bob _bob _2_bob_ bob_2 BoB
  ```

- there are some reserved words:

  ```
  and, assert, break, class, continue, def, del, elif, else, except,
  exec, finally, for, from, global, if, import, in, is, lambda, not,
  or, pass, print, raise, return, try, while
  ```

Understanding
Reference Semantics in Python

Understanding reference semantics I

- assignment manipulates references

  `y = 3` → makes y reference the object 3
  `x = y` → does not make a copy of the object referenced by y (i.e. 3)

  `x = y` → makes x reference the object referenced by y

- very useful, but beware!

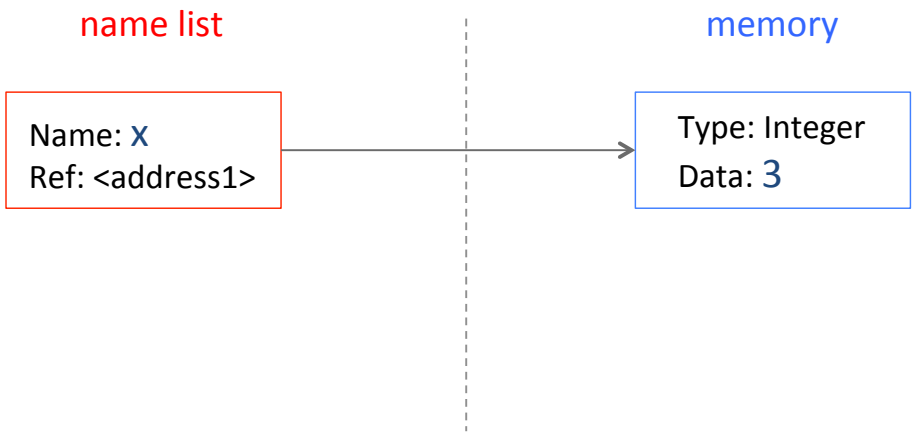
- example:

```
>>> a = [1, 2, 3]          # a references the list [1, 2, 3]
>>> b = a                  # b references the object referenced by a
>>> a.append(4)            # this changes the list referenced by a
>>> print b                # if we print the object referenced by b...
[1, 2, 3, 4]               # we notice that the object has changed!
```

WHY ?

Understanding reference semantics II

- there is a lot going on when we type: `x = 3`
- first, an integer `3` is created and stored in memory
- a name `x` is created
- a *reference* to the memory location storing the `3` is then assigned to the name `x`
- so then we say that the value of `x` is `3`, we mean that `x` now refers to the integer `3`.



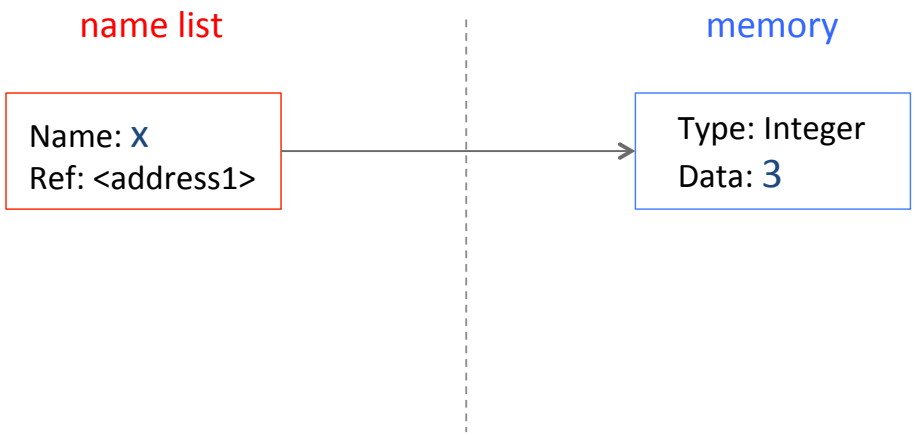
Understanding reference semantics III

- the data 3 is of type integer, in python the data types integer, float, and string (and tuple) are "**immutable**"
- this does not mean we cannot change the value of `x`, i.e. change what `x` refers to
- for example, we could increment `x`:

```
>>> x = 3
>>> x = x + 1
>>> print x
4
```

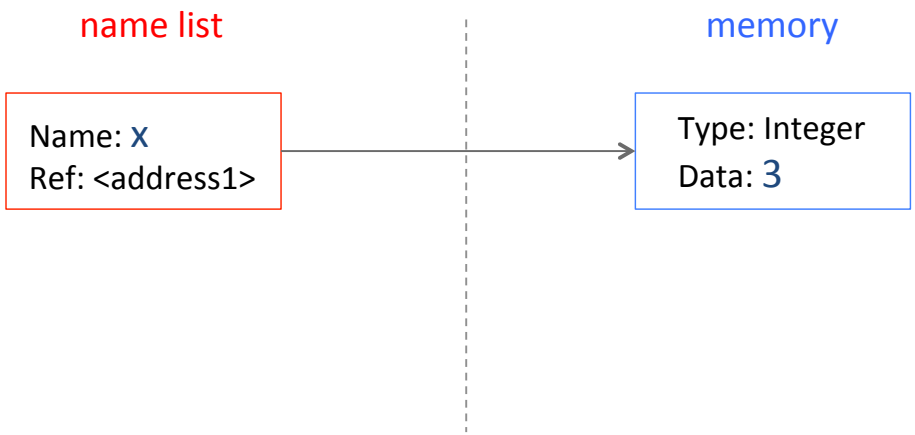
Understanding reference semantics IV

- if we increment `x`, then what really happens is:
  1. the reference of name `x` is looked up
  2. the value at that reference is retrieved
  3. the 3+1 calculation occurs, producing a new data element `4`, which is assigned to a fresh memory location with a new reference
  4. the name `x` is changed to point to this new reference
  5. the old data `3` is deleted if no other name refers to it



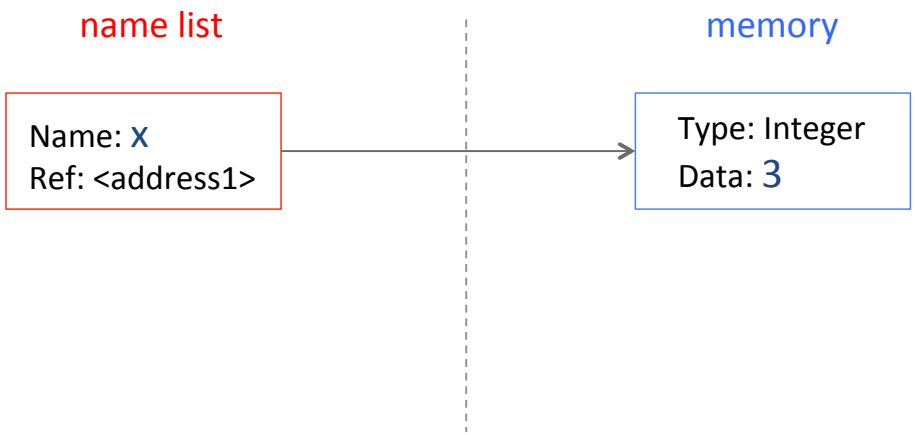
Understanding reference semantics IV

- if we increment `x`, then what really happens is:
  1. the reference of name `x` is looked up
  2. the value at that reference is retrieved
  3. the 3+1 calculation occurs, producing a new data element 4, which is assigned to a fresh memory location with a new reference
  4. the name `x` is changed to point to this new reference
  5. the old data 3 is deleted if no other name refers to it



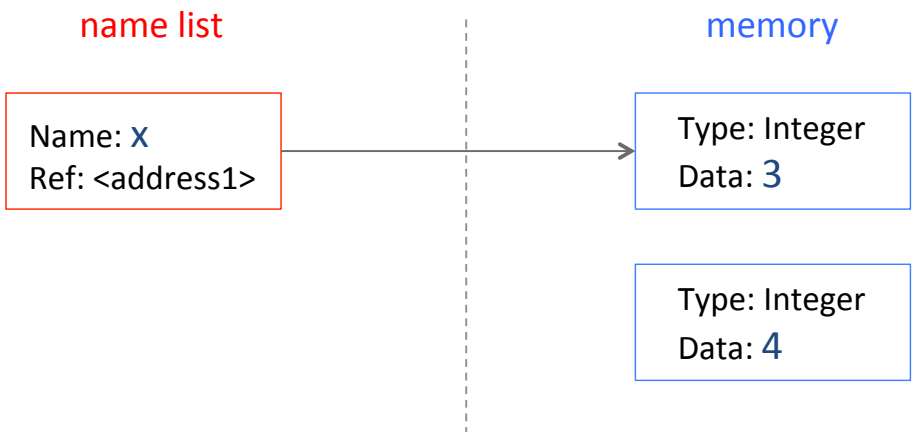
Understanding reference semantics IV

- if we increment `x`, then what really happens is:
  1. the reference of name `x` is looked up
  2. the value at that reference is retrieved
  3. the 3+1 calculation occurs, producing a new data element 4, which is assigned to a fresh memory location with a new reference
  4. the name `x` is changed to point to this new reference
  5. the old data 3 is deleted if no other name refers to it



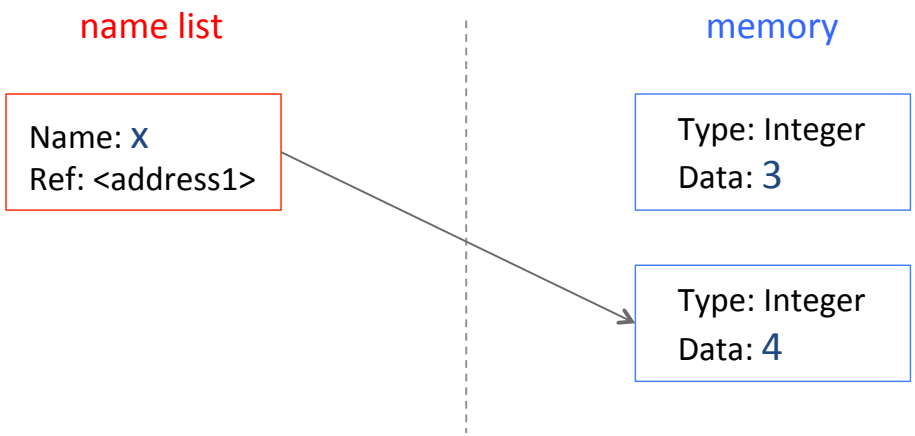
Understanding reference semantics IV

- if we increment `x`, then what really happens is:
  1. the reference of name `x` is looked up
  2. the value at that reference is retrieved
  3. the 3+1 calculation occurs, producing a new data element 4, which is assigned to a fresh memory location with a new reference
  4. the name `x` is changed to point to this new reference
  5. the old data 3 is deleted if no other name refers to it



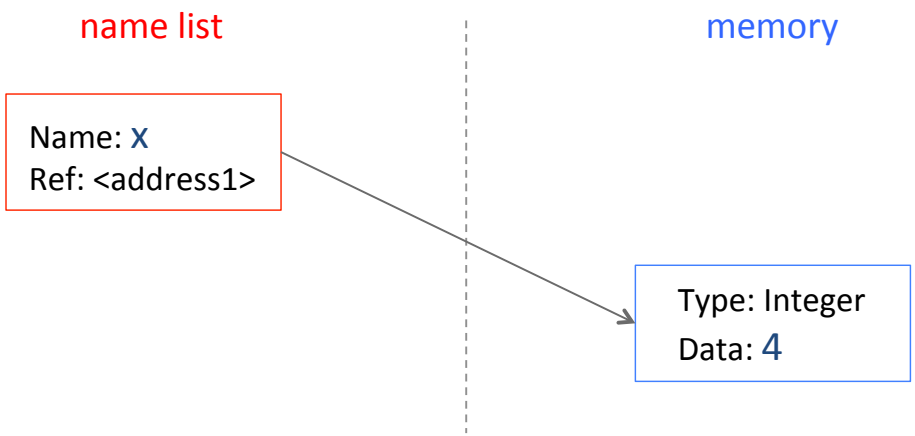
Understanding reference semantics IV

- if we increment `x`, then what really happens is:
  1. the reference of name `x` is looked up
  2. the value at that reference is retrieved
  3. the 3+1 calculation occurs, producing a new data element 4, which is assigned to a fresh memory location with a new reference
  4. the name `x` is changed to point to this new reference
  5. the old data 3 is deleted if no other name refers to it



Understanding reference semantics IV

- if we increment `x`, then what really happens is:
  1. the reference of name `x` is looked up
  2. the value at that reference is retrieved
  3. the 3+1 calculation occurs, producing a new data element 4, which is assigned to a fresh memory location with a new reference
  4. the name `x` is changed to point to this new reference
  5. the old data 3 is deleted if no other name refers to it



Assignment I

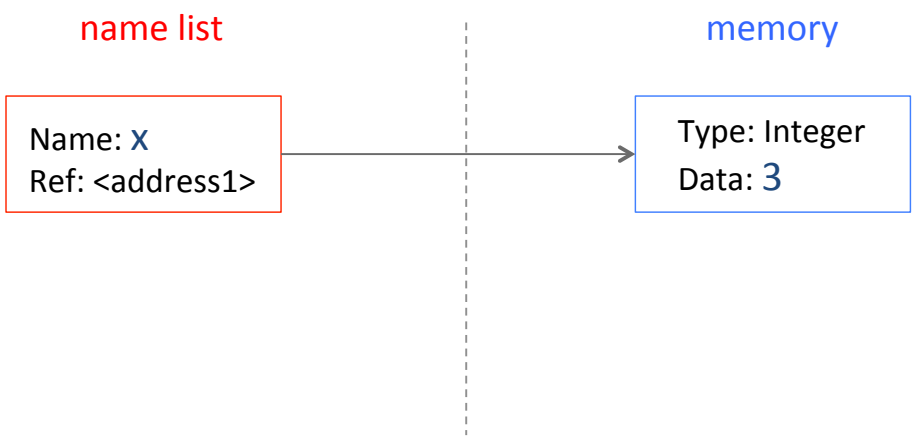
- for simple built-in datatypes (integers, floats, strings), assignment behaves as you would expect:

```
>> x = 3                    # creates name x that refers to 3
>> y = x                    # creates name y that refers to 3
>> y = 4                    # makes y reference to 4, y changes
>> print x                  # no effect on x, which still refers to 3
3
```

Assignment I

- for simple built-in datatypes (integers, floats, strings), assignment behaves as you would expect:

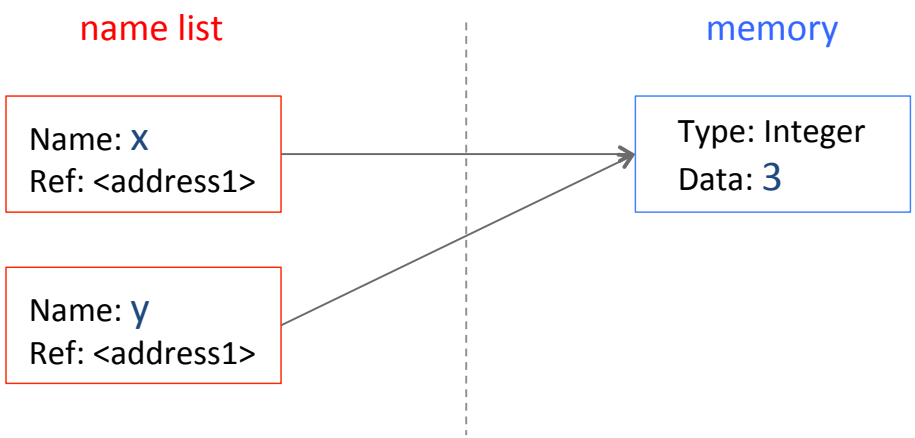
```
>> x = 3                    # creates name x that refers to 3
>> y = x                    # creates name y that refers to 3
>> y = 4                    # makes y reference to 4, y changes
>> print x                  # no effect on x, which still refers to 3
3
```



Assignment I

- for simple built-in datatypes (integers, floats, strings), assignment behaves as you would expect:

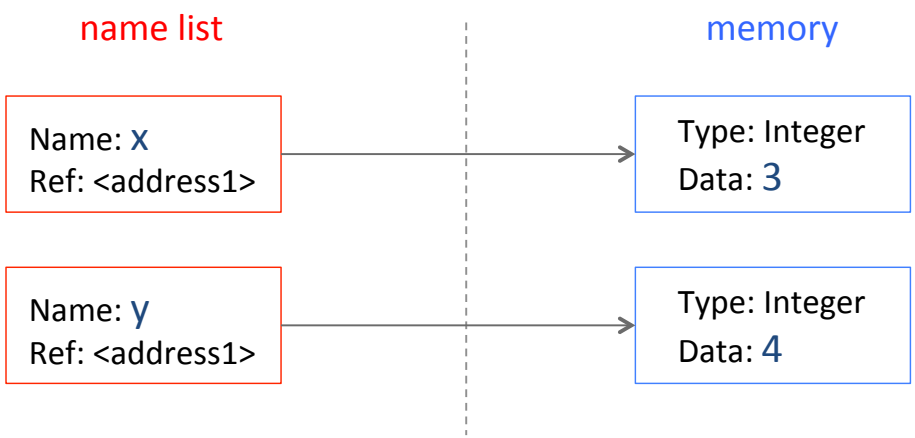
```
>> x = 3                    # creates name x that refers to 3
>> y = x                    # creates name y that refers to 3
>> y = 4                    # makes y reference to 4, y changes
>> print x                  # no effect on x, which still refers to 3
3
```



Assignment I

- for simple built-in datatypes (integers, floats, strings), assignment behaves as you would expect:

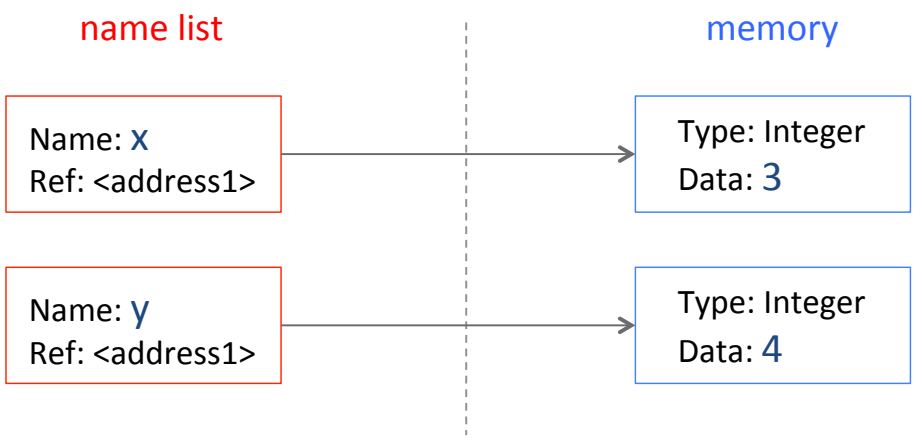
```
>> x = 3                    # creates name x that refers to 3
>> y = x                    # creates name y that refers to 3
>> y = 4                    # makes y reference to 4, y changes
>> print x                  # no effect on x, which still refers to 3
3
```



Assignment I

- for simple built-in datatypes (integers, floats, strings), assignment behaves as you would expect:

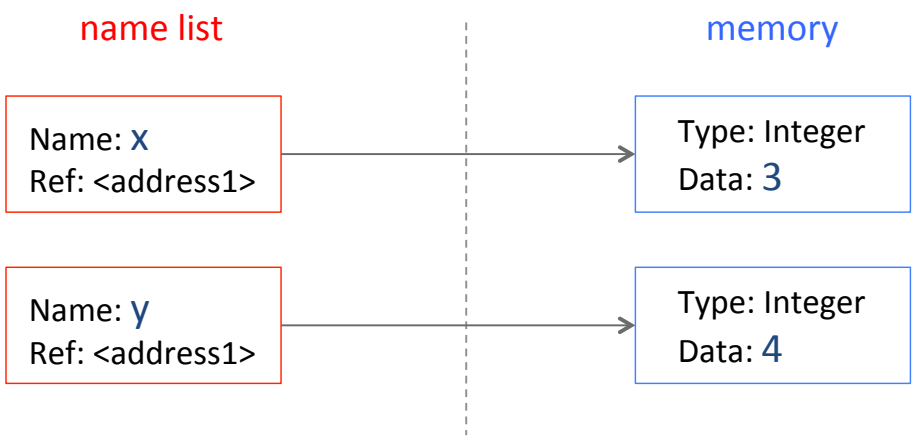
```
>> x = 3                    # creates name x that refers to 3
>> y = x                    # creates name y that refers to 3
>> y = 4                    # makes y reference to 4, y changes
>> print x                  # no effect on x, which still refers to 3
3
```



Assignment I

- for simple built-in datatypes (integers, floats, strings), assignment behaves as you would expect:

```
>> x = 3                    # creates name x that refers to 3
>> y = x                    # creates name y that refers to 3
>> y = 4                    # makes y reference to 4, y changes
>> print x                  # no effect on x, which still refers to 3
3
```



Assignment II

- for other datatypes (lists, dictionaries, user-defined types), assignment works differently
  - these datatypes are "**mutable**"
  - when we change these data, we do it *in place*
  - we do not copy them into a new memory address each time
  - if we type `y=x` and then modify `y`, both `x` and `y` are changed

immutable

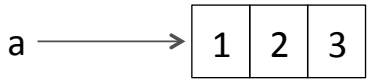
```
>> x = 3
>> y = x
>> y = 4
>> print x
3
```

mutable

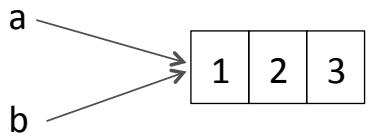
```
x = some mutable object
y = x
make a change to y
look at x
x changed as well
```

Example: changing a shared list

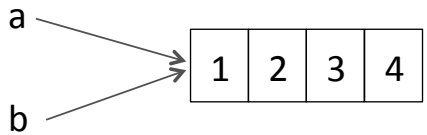
a = [1, 2, 3]



b = a



a.append(4)



Example: changing a shared list

- so again here is the source code:

```
>>> a = [1, 2, 3]          # a references the list [1, 2, 3]
>>> b = a                  # b references the object referenced by a
>>> a.append(a)            # this changes the list referenced by a
>>> print b                # if we print the object referenced by b...
[1, 2, 3, 4]               # b has changed!
```

Sequence types:

Tuples, Lists, and Strings

Sequence types

1. Tuple
   - A simple ***immutable*** ordered sequence of items
   - items can be of mixed types, including collection of types
2. Strings
   - ***immutable***
   - conceptually very much like a tuple
3. List
   - ***mutable*** ordered sequence of items of mixed types

Similar syntax

- all three sequence types (tuples, strings, and lists) share much of the same syntax and functionality
- key differences:
  - tuples and strings are ***immutable***
  - lists are ***mutable***
- the operations shown in this section can be applied to all sequence types
  - most examples will just show the operation performed on one

Sequence types I

- tuples are defined using parentheses (and commas):

```
>>> tu = (23, 'abc', 4.56, (2,3), 'def')
```

- lists are defined using square brackets (and commas):

```
>>> li = ["abc", 34, 4.34, 23]
```

- strings are defined using quotes (", ', or """):

```
>>> st = "Hello World"
>>> st = 'Hello World'
>>> st = """This is a multi-line string that uses triple quotes."""
```

Sequence types II

- individual elements of a tuple, list or string can be accessed using square barcket "array" notation (note that all are 0 based...):

```
>>> tu = (23, 'abc', 4.56, (2,3), 'def')
>>> tu[1]
'abc'


>>> li = ["abc", 34, 4.34, 23]
>>> li[1]
34


>>> st = "Hello World"
>>> st[1]
'e'
```

Positive and negative indices

```
>>> tu = (23, 'abc', 4.56, (2,3), 'def')
```

positive index: count from the left, starting with 0:

```
>>>tu[1]
'abc'
```

negative lookup: count from right, starting with -1:

```
>>> tu[-3]
4.56
```

Slicing: return copy of a subset I

```
>>> tu = (23, 'abc', 4.56, (2,3), 'def')
```

return a copy of the container with a subset of the original members; start copying at the first index and stop copying *before* the second index:

```
>>>tu[1:4]
('abc', 4.56, (2,3))
```

you can also use negative indeces when slicing:

```
>>> tu[1:-1]
('abc', 4.56, (2,3))
```

Slicing: return copy of a subset II

```
>>> tu = (23, 'abc', 4.56, (2,3), 'def')
```

omit the first index to make a copy starting from the beginning of the container:

```
>>>tu[:2]
(23, 'abc')
```

omit the second index to make a copy starting at the first index and going to the end of the container:

```
>>> tu[2:]
(4.56, (2,3), 'def')
```

Copying the whole sequence

to make a copy of an entire sequence, you can use `[:]`:

```
>>> tu[:]
(23, 'abc', 4.56, (2,3), 'def')
```

note the difference between these two lines for mutable sequences:

```
>>> list2 = list1      # 2 names refer to 1 object, changing one affects both

>>> list2 = list1[:]   # 2 independent copies, 2 objects
```

The 'in' operator

- boolean test on whether a value is inside a container:

```
>>> t = [1, 2, 4, 5]
>>> 3 in t
False
>>> 4 in t
True
>>> 4 not in t
False
```

- for strings, tests for substrings:

```
>>> a = 'abcde'
>>> 'c' in a
True
>>> 'cd' in a
True
>>> 'ac' in a
False
```

- be careful: the 'in' keyword is also used in the synthax of 'for' loops, and list comprehensions...

The '+' operator

- the + operator produces a new tuple, list, or string whose value is the concatenation of its argument:

```
>>> (1, 2, 3) + (4, 5, 6)
(1, 2, 3, 4, 5, 6)
```

```
>>> [1, 2, 3] + [4, 5, 6]
[1, 2, 3, 4, 5, 6]
```

```
>>> "Hello" + " " + "World"
'Hello World'
```

The '*' operator

- the * operator produces a new tuple, list, or string that repeats the original content:

```
>>> (1, 2, 3) * 3
(1, 2, 3, 1, 2, 3, 1, 2, 3)


>>> [1, 2, 3] * 3
[1, 2, 3, 1, 2, 3, 1, 2, 3]


>>> "Hello" * 3
'HelloHelloHello'
```

Creating lists

python has core functions for creating lists; the most useful is the `range` function, which can be used to create a uniformly spaced sequence of integers; the general form of the function is `range(start, stop, step)`, where the arguments are all integers and `start` and `step` are optional;

```
>>> range(10)      # makes a list of 10 integers from 0 to 9
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

>>> range(3,10)    # makes a list of 10 integers from 3 to 9
[3, 4, 5, 6, 7, 8, 9]

>>> range(0,10,2) # makes a list of 10 integers from 0 to 9 with increment 2
[0, 2, 4, 6, 8]
```

Multidimensional lists

we can also make multidimensional lists, or lists of lists; such constructs can be useful in making tables, matrices, and other structures; consider, for example, a list of three elements, where each element in the list is itself a list:

```
a = [[3, 9], [8, 5], [11, 1]] # create a multidimensional list
```

```
>>> a[0] # access the first element of the list
[3, 9]
```

```
>>> a[1] # access the second element of the list
[8, 5]
```

```
>>> a[1][0] # access the first element of a[1]
8
```

```
>>> a[2][1] # access the second element of a[2]
1
```

`numpy` arrays

the `numpy` array is the real workhorse of data structures for scientific applications;

the `numpy` array, formally called `ndarray` (now simply `array`), is similar to a list but where all elements are of the same type; the elements of a `numpy` array are usually numbers, but can also be strings, or other objects;

when the elements are numbers, they must all be of the same type; for example, they might be all integers or all floating point numbers.

Creating 1-D arrays – `array`

`numpy` has a number of functions for creating arrays; we focus on some of them; the first, the `array` function, can convert a list to an array:

```
>>> a = [0, 0, 1, 4, 7., 16, 31, 64, 127]   # create a list

>>> b = np.array(a)   # convert the list into a 1-d array

>>> b
array([  0.,   0.,   1.,   4.,   7., 16., 31., 64., 127.])
```

note that all elements of the array have become floating numbers because one element of the original list was a floating number.

Creating 1-D arrays – `linspace`

the second way arrays can be created is using the numpy function `linspace`, which creates an array of `N` evenly spaced points between a starting point and an ending point, both included; the form of the function is `linspace(start, stop, N)`; if the third argument `N` is omitted, then `N`=50.

```
>>> np.linspace(0, 10, 5)  # creates 5 evenly spaced points between 0 and 10
array([  0. ,  2.5,  5. ,  7.5, 10. ])
```

note that we will use `linspace` a lot for creating the time arrays for integrating the differetial equations comprising our models!

Creating 1-D arrays – arange

the third way arrays can be created is using the numpy function `arange`, which is similar to the python function `range` for creating lists; the form of the function is `arange(start, stop, step)`; if the third argument is omitted, then `step=1`; if the first and third arguments are omitted, then `start=0` and `step=1`.

```
>>> np.arange(0, 10, 2)  # create an array from 0 to 10 with step 2
array([0, 2, 4, 6, 8])

>>> np.arange(0., 10, 2)
array([ 0., 2., 4., 6., 8.])

>>> np.arange(0, 10, 1.5)
array([ 0. , 1.5, 3. , 4.5, 6. , 7.5, 9. ])
```

note that the `arange` function produces points evenly spaced and the final point is excluded.

Creating 1-D arrays – zeros and ones

a fourth way to create an array is with the zeros and ones functions; as their names imply, they create arrays where all the elements are either zeros or ones; they take one mandatory argument, the number of elements in the array, and one optional argument that specifies the data type of the array; left unspecified, the data type is float.

```
>>> np.zeros(6)
array([ 0., 0., 0., 0., 0., 0.]) # create an array of 6 zeros

>>> np.ones(8)
array([ 1., 1., 1., 1., 1., 1., 1., 1.]) # create an array of 8 ones

>>> np.ones(8, dtype=int)
array([1, 1, 1, 1, 1, 1, 1, 1]) # create an array of 8 ones, integers
```

Multi-dimensional arrays and matrices

so far we have examined only 1-D (one-dimensional) `numpy` arrays, that is, arrays that consist of one sequence of numbers;

however, `numpy` arrays can be used to represent multidimensional arrays; for example, matrices, which consist of a series of rows and columns;

matrices can be represented using 2-D or higher dimension `numpy` arrays.

Creating multidimensional arrays

there are a number of ways of creating multidimensional `numpy` arrays; the most straightforward way is to convert a list to an array using the `numpy` `array` function.

```
>>> a = [[1., 4, 5], [9, 7, 4]] # create a nested list (a list of two lists)
>>> b = np.array(b) # convert the list into an array
>>> b
array([[ 1.,  4.,  5.],
       [ 9.,  7.,  4.]])
```

Creating multidimensional arrays

also the functions `zeros` and `ones` can be used to create multidimensional arrays; for example a 3 row by 4 column array (or a 3×4 array) with all the elements filled with ones or a 5×5 array with all the elements filled with zeros can be created as follows

```
>>> a = np.ones((3,4), dtype=float) # create a 3x4 array of ones
>>> a
array([[ 1., 1., 1., 1.],
       [ 1., 1., 1., 1.],
       [ 1., 1., 1., 1.]])


>>> b = np.zeros((5,5), dtype=int) # create a 5x5 array of zeros
>>> b
array([[0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0]])
```

Accessing elements of multidimensional arrays

the individual elements of multidimensional arrays can be accessed as follows

```
>>> b = np.array([[1., 4, 5], [9, 7, 4]]) # create a 2x3 array
>>> b
array([[ 1.,  4.,  5.],
       [ 9.,  7.,  4.]])
>>> b[0][2] # access the third element [2] of the first row [0]
5.


>>> b[0,2] # access the third element [2] of the first row [0]
5.


>>> b[:,2]  # access the third element [2] of all raws [:]
array([ 5.,  4.])


>>> b[1,:]  # access the all elements [:] of the second raw [1]
array([ 5.,  4.])


>>> b[:,:]  # access the all elements [:] of all raws [:]
array([[ 1.,  4.,  5.],
       [ 9.,  7.,  4.]])


>>> b[:]  # access the all elements of all raws
array([[ 1.,  4.,  5.],
       [ 9.,  7.,  4.]])
```

Mutability:

Tuples vs. Lists

Tuples: immutable

```
>>> t = (23, 'abc', 4.56, (2,3), 'def')
>>> t[2] = 3.14

Traceback (most recent call last):
  File "<pyshell#75>", line 1, in -toplevel-
    tu[2] = 3.14
TypeError: object doesn't support item assignment
```

tuples cannot be changed; but a new tuple can be declared that is referenced by the previously used name:

```
>>> t = (23, 'abc', 3.14, (2,3), 'def')
```

Lists: mutable

```
>>> li = ['abc', 23, 4.34, 23]
>>> li[1] = 45
['abc', 45, 4.34, 23]
```

- we can change lists in place
- name `li` still points to the same memory reference
- the mutability of lists means that they aren't as fast as tuples

Operations on lists only I

```
>>> li = [1, 11, 3, 4, 5]

>>> li.append('a')     # our first exposure to method synthax
>>> li
[1, 11, 3, 4, 5, 'a']

>>>li.insert (2, 'i')
>>>li
[1, 11, 'i', 3, 4, 5, 'a']
```

The `extend` method vs. the `+` operator

- `+` creates a fresh list (with a new memory reference)
- `extend` operates on list `li` in place

```
>>> li.extend([9, 8, 7])
>>> li
[1, 11, 'i', 3, 4, 5, 'a', 9, 8, 7]
```

Confusing:

- `extend` takes a list as an argument
- `append` takes a singleton as an argument

```
>>> li.append([10, 11, 12])
>>> li
[1, 11, 'i', 3, 4, 5, 'a', 9, 8, 7, [10, 11, 12]]
```

Operations on lists only III

```
>>> li = ['a', 'b', 'c', 'b']

>>> li.index('b')     # index of first occurence
1

>>>li.count('b')      # number of occurrences
2

>>>li.remove('b')     # remove first occurrence
>>>li
['a', 'c', 'b']
```

Operations on lists only IV

```
>>> li = [5, 2, 6, 8]

>>> li.reverse()     # reverse the list in place
>>> li
[8, 6, 2, 5]

>>>li.sort()         # sort the list in place
>>>li
  [2, 5, 6, 8]

>>>li.sort(some_function) # sort in place using user-defined comparison
```

Tuples vs. lists

- Lists slower but more powerful than tuples
  - lists can be modified, and they have lots of useful operations we ca perform on them
  - tuples are immutable and have fewer features

- to convert between tuples and lists use the `list()` and `tuple()` functions:

```
li = list(tu)
tu = tuple(li)
```

Dictionaries

Dicitonaries: a mapping type

- dictionaries store a mapping between a set of keys and a set of values
  - keys can be any immutable type
  - values can be any type
  - a single dictionary can store values of different types
- you can define, modify, view, lookup, and delete the key-value pair in the dictionary

Using dictionaries

```
>>> d = {'user':'bozo', 'pswd':1234}
>>> d['user']
'bozo'
>>> d['pswd']
1234
>>> d['bozo']
Traceback (innermost last):
  File <interactive input> line 1, in ?
KeyError: bozo

>>> d = {'user':'bozo', 'pswd':1234}
>>> d['user'] = 'clown'
>>> d
{'user':'clown', 'pswd':1234}

>>> d['id'] = 45
>>> d
{'user':'clown', 'id':45, 'pswd':1234}
```

```
>>> d = {'user':'bozo', 'p':1234, 'i':34}
>>> del d['user']   # remove one
>>> d
{'p':1234, 'i':34}
>>> d.clear()       # remove all
>>> d
{}

>>> d = {'user':'bozo', 'p':1234, 'i':34}
>>> d.keys()        # list of keys
['user', 'p', 'i']
>>> d.values()      # list of values
['bozo', 1234, 34]
>>> d.items()       # list of item tuples
[('user','bozo'), ('p',1234), ('i',34)]
```

Functions

Functions

- `def` creates a function and assigns it a name
- `return` sends a result back to the caller
- arguments are passed by assignment
- arguments and return types are not declared

```
def <name>(arg1, arg2, ..., argN):
  <statements>
  return <value>
```

- example:

```
def times(x,y):
  z = x*y
  return z
```

Passing arguments to functions

- arguments are passed by assignment
- passed arguments are assigned to local names
- assignment to argument names do not affect the caller
- changing a mutable argument may affect the caller

- example:

```
def changer(x,y):
  x = 2          # changes local value of x only
  y[0] = 'hi'    # changes shared object in place


>>> X = 1
>>> L = [1, 2]
>>> changer(X, L)     # pass immutable and mutable
>>> X, L              # X unchanged, L changed
(1, ['hi', 2])
```

Optional arguments

- can define defaults for arguments that need not be passed
- example:

```
def func(a, b, c=10, d=100):
  print a, b, c, d


>>> func(1,2)
  1 2 10 100


>>> func(1,2,3,4)
  1 2 3 4
```

More properties of functions

- all functions in python have a return value
  - even if there is no return line inside the code
- functions without a return, return the special value `None`
- there is no function overloading in python
  - two different functions cannot have the same name, even if they have different arguments
- functions can be used as any other data type, they can be:
  - arguments to function
  - return values of functions
  - assigned to variables
  - part of tuples, lists, etc.

Control of flow

Control of flow

- examples:

```
if x == 3:
   print "x equals 3"
elif x == 2:
   print "x equals 2"
else:
   print "x equals somthing else"
print "this is outside the if"
```

```
assert(number_of_players < 5)
```

```
x = 3
while x < 10:
   if x > 10:
      x+=2 # x=x+2
      continue
   x+=1    # x=x+1
   print "still in the loop"
   if x == 8:
      break
print "outside of the loop"
```

```
for x in range(10):
   if x > 7:
      x+=2 # x=x+2
      continue
   x+=1    # x=x+1
   print "still in the loop"
   if x == 8:
      break
print "outside of the loop"
```

Modules

Why use modules ?

- code reuse
  - routines can be called multiple times within a programme
  - routines can be used from multiple programmes
- namespace partitioning
  - group data together with functions used for that data
- implementing shared services or data
  - can provide global data structure that is accessed by multiple subprogrammes

Creating modules

- modules are functions and variables defined in separate files
- the file name is the module name with the suffix `.py` appended
- example, create a file called `fibo.py` with the following content:

```
# 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
```

Using modules

- now import the module and use the functions it contains
- items are imported using `import` or `from`:

```
>>> import fibo
>>> fibo.fib(500)   # use the funciton fib contained in module fibo
1 1 2 3 5 8 13 21 34 55 89 144 233 377
>>> fibo.fib2(100)
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377]


>>> from fibo import *
>>> fib(500)   # use the funciton fib directly
1 1 2 3 5 8 13 21 34 55 89 144 233 377
>>> fib2(100)
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377]
```

A special function: `odeint`

the '`scipy`' package, and especially its subpackage '`integrate`' provides several integration techniques, including an ordinary differential equation integrator called `odeint`;

`odeint` is a special function that solves a system of ordinary differential equations (ODEs);

Using odeint

- let us solve the equation dy/dt = -2 y over time t 0 to 4 with initial condition y(t=0) = 1
- we define the following function:

```
def deriv(y, t):
    dydt = -2*y
    return dydt
```

- we import odeint:

```
from scipy.integrate import odeint
```

- we create the time vector and set the initial condition:

```
import numpy as np
t = np.linspace(0, 4.0, 40)
y0=1.0
```

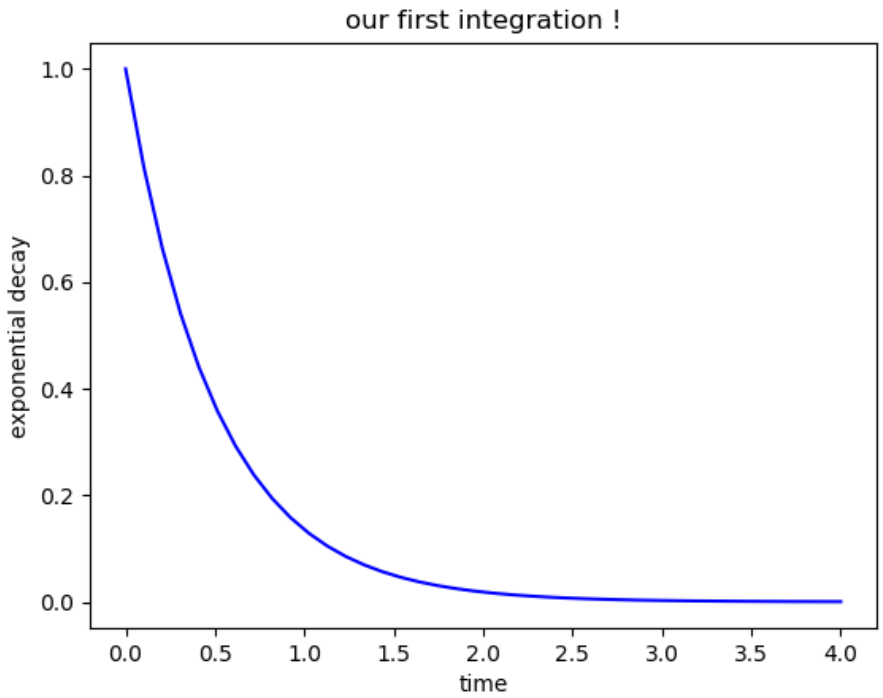
- we call odeint to integrate the differential equation:

```
y = odeint(deriv, y0, t)
```

Plotting the results

- to plot the results we use the module `pyplot` contained in `matplotlib`:

```
import matplotlib.pyplot as plt
plt.plot(t, y, 'b')
plt.title('our first integration!')
plt.ylabel('y')
plt.xlabel('time')
plt.show()
```



Exercises

Exercise 1 Familiarise with python, run the basic commands introduced during the lecture and experience their effects. Code the simple exponential decay model, produce different solutions using different time decay parameter values, plot the results...

Exercise 2 Code the 'branching' model, i.e.: $dn(t)/dt = b \cdot n(t)$, in python considering: an initial number of branches of $n(t = 0) = 10.0$, a branching rate of $b = 0.25$ branches per day; generate a solution $n(t)$ over 101 evenly spaced time steps over 10 days and plot the results.

Exercise 3 Code the 'cat and mouse' model, i.e.: $dn(t)/dt = b \cdot n(t) - d \cdot n(t) + m$, in python considering: an initial number of mice of $n(t = 0) = 10.0$, a birth rate of 2 mice every 8 days, a cat eats 1 mouse every 2 days, and 3 new mice migrate every 10 days from nearby yards; generate a solution $n(t)$ over 101 evenly spaced time steps over 10 days and plot the results.

Exercise 4 Code the 'flu' model, i.e.: $ds(t)/dt = -a \cdot c \cdot n(t) \cdot s(t)$ and $dn(t)/dt = +a \cdot c \cdot s(t) \cdot n(t)$, in python considering: 10 contacts between carrier and healthy individuals every day and a 2.5 % probability that the flu is transmitted at every contact, an initial number of healthy individuals of $s(t = 0) = 95.0$, an initial number of carriers of $n(t = 0) = 5.0$; generate a solution over 101 evenly spaced time steps over 10 days and plot the results; plot both $s(t)$ and $n(t)$ and $s(t) + n(t)$ (i.e. the total number of individuals) in the same panel; after how many days does the model reach steady-state?

Further reading

M. Lutz, 2013
Learning Python
O'Reilly Media, Inc.

S. Linge & H. P. Langtangen, 2016
Programming for Computations – Python
Springer Open