Python for Scientific Computing

http://bender.astro.sunysb.edu/classes/python-science

Course Goals

- Simply: to learn how to use python to do
  - Numerical analysis
  - Data analysis
  - Plotting and visualizations
  - Symbol mathematics
  - Write applications
  - …

Why Python?

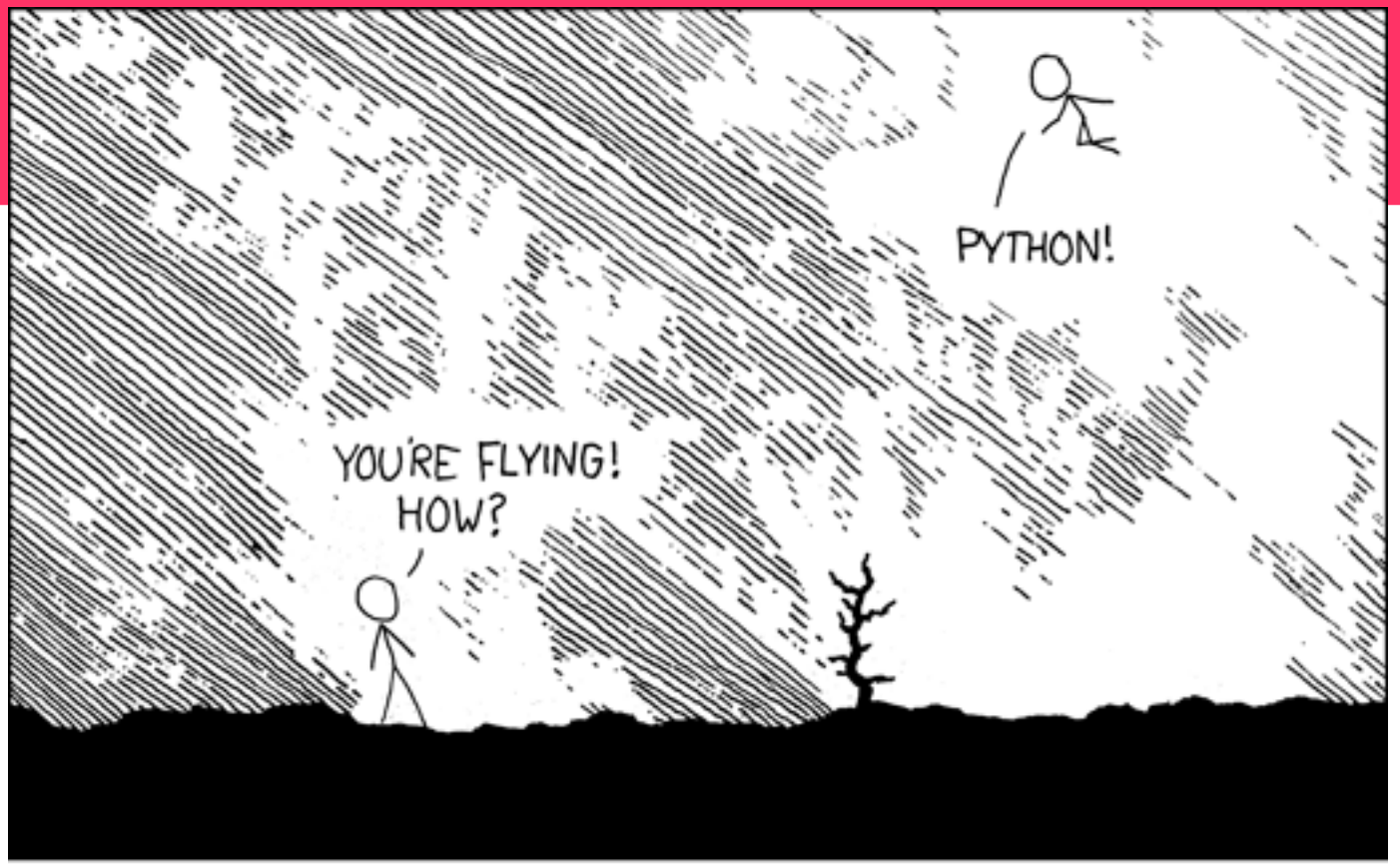
- Very high-level language
  - Provides many complex data-structures (lists, dictionaries, ...)
  - Your code is shorter than a comparable algorithm in a compiled language
- Many powerful libraries to perform complex tasks
  - Parse structured inputs files
  - send e-mail
  - interact with the operating system
  - make plots
  - make GUIs
  - do scientific computations
  - ...
- Easy to prototype new tools
- Cross-platform and Free

Why Python?

- Dynamically-typed
- Object-oriented foundation
- Extensible (easy to call Fortran, C/C++, ...)
- Automatic memory management (garbage collection)
- Ease of readability (whitespace matters)

... and for this course ...

- Very widely adopted in the scientific community
  - Mostly due to the very powerful array library NumPy


PYTHON!
YOU'RE FLYING! HOW?


I LEARNED IT LAST NIGHT! EVERYTHING IS SO SIMPLE!
HELLO WORLD IS JUST print "Hello, world!"
I DUNNO... DYNAMIC TYPING? WHITESPACE?
COME JOIN US! PROGRAMMING IS FUN AGAIN! IT'S A WHOLE NEW WORLD UP HERE!
BUT HOW ARE YOU FLYING?
I JUST TYPED import antigravity
THAT'S IT?
... I ALSO SAMPLED EVERYTHING IN THE MEDICINE CABINET FOR COMPARISON.
BUT I THINK THIS IS THE PYTHON.

(xkcd)

Hello, World!

- If you have python installed properly, you can start python simply by typing `python` at your command line prompt
  - The python shell will come up
  - Our hello world program is simply:

    ```
    print “Hello, World!”
    ```
  - Or, in python 3.x (more on this later…):

    ```
    print(“Hello World”)
    ```

Communities

- Many scientific disciplines have their own python communities that
  - Provide tutorials
  - Cookbooks
  - Libraries to do standard analysis in the field
- For the most part, these uild on the Open nature of python
- I've put links on our course page to some information on the python communities for:
  - Astronomy
  - Atmospheric Science
  - Biology
  - Cognitive Science
  - Psychology
  - Let me know of any others!

Scientific Python Stack

- Most scientific libraries in python build on the Scientific Python stack:

NumPy
Base N-dimensional array package

SciPy library
Fundamental library for scientific computing

Matplotlib
Comprehensive 2D Plotting

IPython
Enhanced Interactive Console

Sympy
Symbolic mathematics

pandas
Data structures & analysis

(scipy.org)

Starters...

- Before we get into python, we'll review some of the core ideas about numerical computing

Basics of Computation

- Computers store information and allow us to operate on it.
  - That's basically it.
  - Computers have finite memory, so it is not possible to store the infinite range of numbers that exist in the real world, so approximations are made.
- Great floating point reference
  - *What Every Computer Scientist Should Know About Floating-Point Arithmetic* by D. Goldberg

Integers

- Basic unit of information in a computer is a bit: 0 or 1
- 8 bits = 1 byte
- Different types of information require more bits/bytes to store.
  - A logical (T or F) can be stored in a single bit.
    - C/C++: `bool` datatype
    - Fortran: `logical` datatype
  - Integers:
    - Standard in many languages is 4-bytes. This allows for 2^{32}-1 distinct values.
      - This can store: -2,147,483,648 to 2,147,483,647 (signed)
        - C/C++: `int` (usually) or `int32_t`
        - Fortran: `integer` or `integer*4`
      - Or it can store: 0 to 4,294,967,295 (unsigned)
        - C/C++: `uint` or `uint32_t`
        - Fortran (as of 95): `unsigned`

Integers

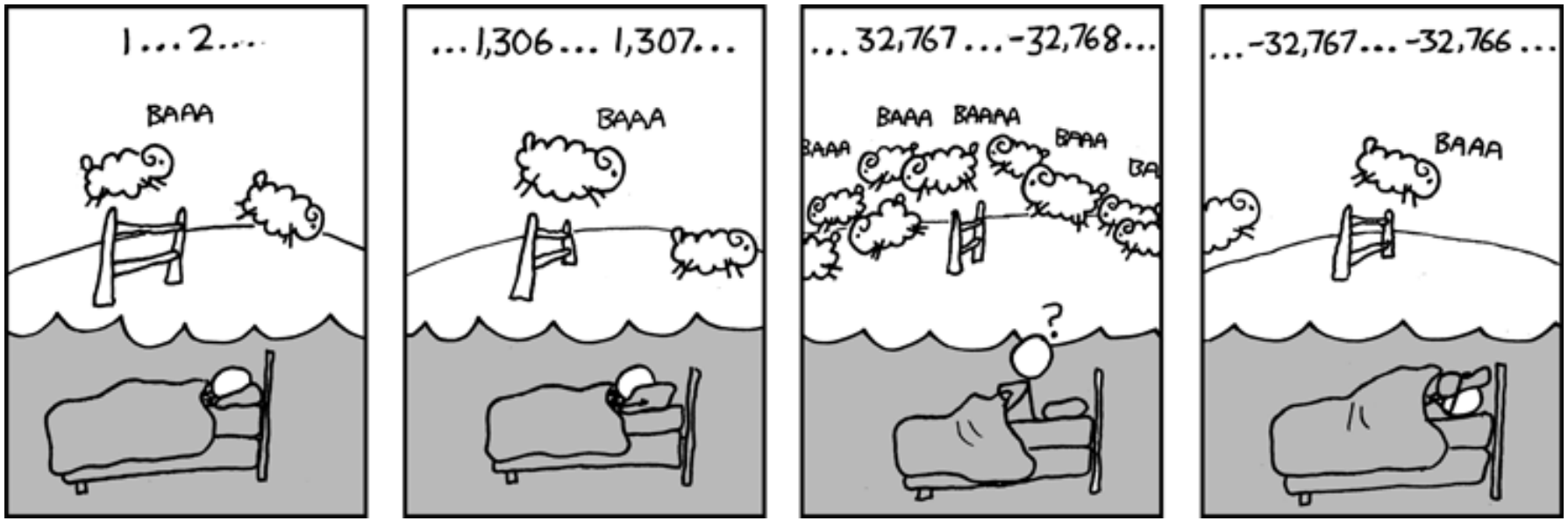
- Integers (continued):
  - Sometimes 2-bytes. This allows for 2^{16}-1 distinct values.
    - This can store: -32,768 to 32,767 (signed)
      - C/C++: `short` (usually) or `int16_t`
      - Fortran: `integer*2`
    - Or it can store: 0 to 65,535 (unsigned)
      - C/C++: `uint16_t`
      - Fortran (as of 95): `unsigned*2`
  - Note for IDL users: the standard integer in IDL is a 2-byte integer. If you do

    ```
    i = 2
    ```

    that's 2-bytes. To get a 4-byte integer, do:

    ```
    i = 2l
    ```

Overflow



(xkcd)

Overflow example...

Overflow

```
iold = -1
i = 1

type_init = type(i)
print "type currently: ", type_init

while (i > iold):
    print i
    iold = i
    i *= 2

    if (not type(i) == type_init):
        print "type changed, now: ", type(i)
        break

print i
```

Integers

- Python allows for the date size of the integer to scale with the size of the number: https://www.python.org/dev/peps/pep-0237/
  - Initially, it is the largest value supported in hardware on a machine (64-bits on a 64-bit machine—see `sys.maxint`

```
def fac(x):
    if (x == 0):
        return 1
    else:
        return x*fac(x-1)


a = fac(52)
print a
print a.bit_length()
```

- This prints:

```
80658175170943878571660636856403766975289505440883277824000000000000
226
```

Note: python 3.x does away with the distinction between int and long altogether.

Real Numbers

- Floating point format
  - Infinite real numbers on the number line need to be represented by a finite number of bits
  - Approximations made
    - Finite number of decimal places stored, maximum range of exponents.
  - Not every number can be represented.
    - In base-2 (what a computer uses), `0.1` does not have an exact representation (see S 2.1 of Goldberg)

Exact Representations

- `0.1` as represented by a computer:

```
>>> b = 0.1
>>> print type(b)
<type 'float'>
>>> print "{:30.20}".format(b)
         0.10000000000000000555

>>> import sys
>>> print sys.float_info
```

Floating Point

- In the floating point format, a number is represented by a significand (or mantissa), and exponent, and a sign.
  - Base 2 is used (since we are using bits)
  - $0.1 \sim (1 + 1 \cdot 2^{-1} + 0 \cdot 2^{-2} + 0 \cdot 2^{-3} + 1 \cdot 2^{-4} + 1 \cdot 2^{-5} + ...) \cdot 2^{-4}$

  - All significand's will start with 1, so it is not stored
  - Single precision:
    - Sign: 1 bit; exponent: 8 bits; significand: 24 bits (23 stored) = 32 bits
    - Range: 2^7-1 in exponent (because of sign) = 2^{127} multiplier ~ 10^{38}
    - Decimal precision: ~6 significant digits
  - Double precision:
    - Sign: 1 bit; exponent: 11 bits; significand: 53 bits (52 stored) = 64 bits
    - Range: 2^{10}-1 in exponent = 2^{1023} multiplier ~ 10^{308}
    - Decimal precision: ~15 significant digits

Floating Point

- Overflows and underflows can still occur when you go outside the representable range.
  - The floating-point standard will signal these (and compilers can catch them)
- Some special numbers:
  - NaN = `0/0` or $\sqrt{-1}$
  - Inf is for overflows, like `1/0`
  - Both of these allow the program to continue, and both can be trapped (and dealt with)
  - `-0` is a valid number, and `-0 == 0` in comparison
- Floating point is governed by an IEEE standard
  - Ensures all machines do the same thing
  - Aggressive compiler optimizations can break the standard

Numerical Precision

- Finite number of bits used to represent #s means there is a finite precision, the machine epsilon, ε
  - One way to find this is to ask when 1 + ε = 1

```
x = 1.0

eps = 1.0

while (not x + eps == x):
    eps = eps/2.0

# machine precision is 2*eps, since that was the last
# value for which 1 + eps was not 1
print 2*eps
```

  - This gives `2.22044604925e-16`

Numerical Precision

- This means that most real numbers do not have an exact representation on a computer.
  - Spacing between numbers varies with the size of numbers
  - Relative spacing is constant

$$\text{relative roundoff error} = \frac{|\text{true number} - \text{computer number}|}{|\text{true number}|} \leq \epsilon$$

Round-off Error

- Round-off error is the error arising from the fact that no every number is exactly representable in the finite precision floating point format.
  - Can accumulate in a program over many arithmetic operations

Round-off Example 1

(Yakowitz & Szidarovszky)

- Imagine that we can only keep track of 4 significant digits
- Compute $\sqrt{x+1} - \sqrt{x}$
  - Take $x = 1984$
  - Keeping only 4 digits each step of the way,

$$\sqrt{x+1} - \sqrt{x} = 44.55 - 44.54 = 0.01$$

  - We've lost a lot of precision
- Instead, consider:

$$\sqrt{x+1} - \sqrt{x} = (\sqrt{x+1} - \sqrt{x}) \left(\frac{\sqrt{x+1} + \sqrt{x}}{\sqrt{x+1} + \sqrt{x}} \right) = \frac{1}{\sqrt{x+1} + \sqrt{x}}$$

- Then

$$\sqrt{1985} - \sqrt{1984} = \frac{1}{\sqrt{1985} + \sqrt{1984}} = \frac{1}{44.55 + 44.54} = 0.01122$$

Comparing Floating Point #s

- When comparing two floating point #s, we need to be careful
- If you want to check whether 2 computed values are equal, instead of asking if

$$x = y$$

- it is safer to ask whether they agree to within some tolerance

$$|x - y| < \epsilon$$

Associative Property

- You learned in grade school that: (a + b) + c = a + (b + c)
  - Not true with floating point
  - You can use parenthesizes to force order of operations
  - If you want to enforce a particular association, use parenthesis

Computer Languages

- You can write any algorithm in any programming language—they all provide the necessary logical constructs
  - However, some languages make things much easier than others
  - C
    - Excellent low-level machine access (operating systems are written in C)
    - Multidimensional arrays are “kludgy”
  - Fortran (Formula Translate)
    - One of the earliest complied languages
    - Large code base of legacy code
    - Modern Fortran offers many more conveniences than old Fortran
    - Great support for arrays
  - Python
    - Offers many high-level data-structures (lists, dictionaries, arrays)
    - Great for quick prototyping, analysis, experimentation
    - Increasingly popular in scientific computing

Computer Languages

- IDL
  - Proprietary
  - Great array language
  - Modern (like object-oriented programming) features break the “clean-ness” of the language
- C++
- Others
  - Ruby, Perl, shell scripts, …

Computer Languages

- Compiled languages (Fortran, C, C++, …)
  - Compiled into machine code—machine specific
  - Produce faster code (no interpretation is needed)
  - Can offer lower level system access (especially C)
- Interpreted languages (python, IDL, perl, Java (kind-of) …)
  - Not converted into machine code, but instead interpreted by the interpreter
  - Great for prototyping
  - Can modify itself while running
  - Platform independent
  - Often has dynamic typing and scoping
  - Many offer garbage collection

http://en.wikipedia.org/wiki/Interpreted_language

Computer Languages

- Vector languages
  - Some languages are designed to operate on entire arrays at once (python + NumPy, many Fortran routines, IDL, ...)
    - For interpreted languages, getting reasonable performance requires operating on arrays instead of explicitly writing loops
      - Low level routines are written in compiled language and do the loop behind the scenes
    - We'll see this in some detail when we discuss python
  - Next-generation computing = GPUs ?
    - Hardware is designed to do the same operations on many pieces of data at the same time

Object-Oriented Languages

- Python is an object-oriented language
- Think of an object as a container that holds both data and functions (methods) that know how to operate on that date.
  - Objects are build from a datatype called a class—you can create as many instances (objects) from a class as memory allows
    - Each object will have its own memory allocated
- Objects provide a convenient way to package up data
  - In Fortran, think of a derived type that also has it's own functions
  - In C, think of a struct that, again, also has it's own functions
- Everything, even integers, etc. is an object

Programming Paradigms

Paradigm	Description	Main characteristics	Related paradigm(s)	Critics	Examples
Imperative	Computation as statements that *directly* change a program state (datafields)	Direct assignments, common data structures, global variables		Edsger W. Dijkstra, Michael A. Jackson	C, C++, Java, PHP, Python
Structured	A style of imperative programming with more logical program structure	Structograms, indentation, either no, or limited use of, goto statements	Imperative		C, C++, Java
Procedural	Derived from structured programming, based on the concept of modular programming or the *procedure call*	Local variables, sequence, selection, iteration, and modularization	Structured, imperative		C, C++, Lisp, PHP, Python
Functional	Treats computation as the evaluation of mathematical functions avoiding state and mutable data	Lambda calculus, compositionality, formula, recursion, referential transparency, no side effects			Erlang, Haskell, Lisp, Clojure, Scala, F#
Event-driven including time driven	Program flow is determined mainly by events, such as mouse clicks or interrupts including timer	Main loop, event handlers, asynchronous processes	Procedural, dataflow		ActionScript
Object-oriented	Treats datafields as *objects* manipulated through pre-defined methods only	Objects, methods, message passing, information hiding, data abstraction, encapsulation, polymorphism, inheritance, serialization-marshalling		See here and[1][2]	C++, C#, Java, PHP, Python, Ruby, Scala
Declarative	Defines computation logic without defining its detailed control flow	4GLs, spreadsheets, report program generators			SQL, regular expressions, CSS
Automata-based programming	Treats programs as a model of a finite state machine or any other formal automata	State enumeration, control variable, state changes, isomorphism, state transition table	Imperative, event-driven		
Paradigm	**Description**	**Main characteristics**	**Related paradigm(s)**	**Critics?**	**Examples**

(Wikipedia)

Roundoff vs. Truncation Error

- Roundoff error is just one of the errors we deal with
- Translating continuous mathematical expressions into discrete forms introduces truncation error
- Consider the Taylor series expansion for sine:

$$\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} + \ldots$$

- If x is small, then we can neglect the higher-order terms (we truncate the series). This introduces an error.

Potential Topics

- General python
- Advanced python / special modules
  - Regular expressions
  - Command line/input file parsing
  - Filesystem operations
- NumPy
- SciPy and numerical methods
  - Integration
  - ODEs
  - Curve fitting/optimization
  - Interpolation
  - Signal processing (FFT)
  - Linear algebra
- Plotting / visualization
  - matplotlib
  - MayaVi
- Workflow management with IPython
- Extending python w/ Fortran and C/C++
- Symbolic math with SymPy
- Writing python applications
- GUI programming
- Unit testing
- Python and the Raspberry Pi?
- Others?

Python 2.x vs. Python 3

- See https://wiki.python.org/moin/Python2orPython3
- Mostly about cleaning up the language and making things consistent
  - e.g. `print` is a statement in python 2.x but a function in 3.x
- Some trivial differences
- `.pyc` are now stored in a `__pycache__` directory
- Some gotyas:
  - `1/2` will give different results between python 2 and 3
- It's possible to write code that works with both python 2 and 3—often we will do so by `importing` from `__future__`

Python 2.x vs. Python 3

- Write for both
  - In python 2.6+, do

    ```
    from __future__ import print_function
    ```

    and then use the new `print()` style
  - `exec cmd` becomes `exec(cmd)`
  - Some small changes to how `__init__.py` are done (more on this later)
- One some systems, you can have python 2.x and 3.x installed side by side
  - May need to install packages twice, e.g. `python-numpy` and `python3-numpy`

Introduction to Python

Class Participation

- We'll learn by interactively trying out some ideas and looking at code
  - I want to learn from all of you—share you experiences and expertise
- We'll use the Discussion Forum on Blackboard to interact with one-another outside of class.
- Try out ideas and report to the class what you've learned
  - I can set up an account for you on one of my Linux servers if you want a place to try things out (you would log in remotely)

Who Are We?

- Physics
- Ecology & Evolution
- Sociology
- Biology / Applied Ecology / Evolution
- Cognitive Science
- Integrative Neuroscience
- Applied Math & Statistics
- Mechanical Engineering
- Psychology
- Marine and Atmospheric Science
- Earth & Space Science—Geoscience

What Do You Want To Learn?

- From your posts (in some order by popularity):
  - Replace R / matlab (most popular)
  - General (scientific) programming
  - Data analysis
  - Plotting & Visualization (including 3-d visualization)
  - Interacting with experiments (a la PsychoPy)
  - High performance computing
  - Curve / parameter fitting
  - General scripting / glue for C + Fortran programs

Python Shell

- This is the standard (and most basic) interactive way to use python
  - Runs in a terminal window
  - Provides basic interactivity with the python language
- Simply type `python` at your command prompt
  - You can scroll back through the history using the arrows

IPython Shell

- Type `ipython` at your prompt
- Like the standard shell, this runs in a terminal, but provides many conviences (type `%quickref` to see an overview)
  - Scrollback history preserved between sessions
  - Build-in help with ?
    - `function-name?`
    - `object?`
  - Magics (`%lsmagic` lists all the magic functions)
    - `%run script`: runs a script
    - `%timeit`: times a command
    - Lots of magics to deal with command history (including `%history`)
  - Tab completion
  - Run system commands (prefix with a `!`)
  - Last 3 *output* objects are referred to via `_`, `__`, `___`

IPython Notebooks

- A web-based environment that combines code and output, plots, plain text/stylized headings, LaTeX (later versions), ...
  - Notebooks can be saved and shared
  - Viewable on the web via: `http://nbviewer.ipython.org/`
  - Provides a complete view of your entire workflow
- Start with `ipython notebook`
- I'll provide notebooks for a lot of the lectures to follow so you can play along on your own

Python Scripts

- Scripts are a non-interactive means of writing python code
  - `filename.py`
  - Can be executable by adding:

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

    as the first line and setting the executable bit for the file
- This is also the way that you write python modules that can be `import`-ed into other python code (more on that later...)

Intro to Python: Data Types

- Python has the same basic data types as most languages: integer, floating point, complex, strings
  - data type IPython notebook
- And it has more complex data structures, including lists and dictionaries built-in
  - advanced data type IPython notebook

Lists vs. Arrays

- Lists (note that python lists are not implemented as a simple linked-list: see http://docs.python.org/2/faq/design.html#how-are-lists-implemented)
  - Implemented as a variable-length array
  - Can store multiple different datatypes in a single list
  - Easy to add items to the end (increasing list length)
  - Accessing `a[i]` is independent of list length (unlike a traditional linked list)
- Arrays:
  - Generally used for fixed-length, homogeneous data
  - Less overhead than a list
  - We'll use a special array library (NumPy) for performance later

Intro to Python: Control Flow

- Python uses indentation to denote blocks of code (no explicit endif, enddo, ...)
- Control flow via. while, if, for , ...
  - Can iterate over items in a list—very useful
  - control flow IPython notebook

Intro to Python: Functions

- No distinction between functions and procedures/subroutines
  - Always return something. If not explicitly set, then None
- Can have default values for arguments, optional arguments, variable number of arguments
- Can return multiple values, objects
- Function examples…

Intro to Python: Classes

- Classes are a fundamental concept of object-oriented programming
- An object is an instance of a class
  - Carries data (state) and has methods that can act on that data
  - Objects are easy to pass around
- Simplest use is just as a container to carry associated data together (similar to a struct in C)
- Inheritance, operator overloading, … all supported. See the tutorial.
- Classes example…

Intro to Python: Modules

- Modules are a collection of python statements, functions, variables, grouped together in a file ending with .py
  - Can be imported to be used by any routine
  - Python includes a host of useful modules
  - You can define your own. Python will look in the current directory and then in the PYTHONPATH
- Modules have their own namespace
  - Variables (even global variables) don't clash with those with the same name in the importing program
  - You can change the name of a module on import or import it into the current namespace (*).
- General practice: put all the imports at the top of your code

Intro to Python: Modules

- Example: `profile.py`—a simple timing module for measuring code performance
- In the python shell:
  - `import profile`
  - `help(profile)`
- Note that this module can also be run on its own
  - `if __name__ == "__main__":` clause at the end
- Let's look at an example of using this module in a notebook

Intro to Python: Exceptions

- Exceptions are raised if an action you tried (e.g. opening a file) fails.
  - Left alone, the code will abort, printing the exception
  - You can catch the exception, and take appropriate action to fix the situation
- Built-in exceptions have particular names that let you check for specific failure modes
  - Allows you to write robust code for other users
- Exceptions example…

Intro to Python: I/O

- Basic I/O
- File I/O
- CSV reading
- INI file reading

Caveats

- Slicing (we'll look at this with arrays soon)
- Function defaults: default values are only evaluated once.
  - From python tutorial:

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

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

  - Prints:

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

  - Correct way:

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