UNIL | Université de Lausanne

Global Solution Methods

Simon Scheidegger
simon.scheidegger@unil.ch
January 14th, 2019

Cowles Foundation – Yale University

Road-map – fast forward:

Lecture 1: Monday, January 14th, 9.00-10.20am

- A Brief overview on the course – why are global solution methods useful?

- A Crash Course in Python.

Lecture 2: Wednesday, January 16th, 9.00-10.20am

- Introduction to Sparse Grids and Adaptive Sparse Grids.

Road-map – fast forward (2):

Lecture 3: Friday, January 18th, 9.00-10.20am

- Dynamic Programming and Time Iteration with Sparse Grids.

Lecture 4: Wednesday, January 23rd, 9.00-10.20am

- Introduction Machine Learning (supervised and unsupervised machine learning).
- Basics on Gaussian Process Regression (supervised machine learning).

Road-map – fast forward (3):

Lecture 5: Monday, January 28th, 9.00-10.20am

- Gaussian Process Regression (part II).

- Gaussian Process Classification.

- Bayesian Gaussian Mixture Models (unsupervised machine learning).

Lecture 6: Wednesday, January 30th, 9.00-10.20am

- Dimension-reduction with the active subspace method.

- Solving dynamic models on high-dimensional, (irregularly-shaped) state spaces.

- Wrap-up of the lecture-suite.

Before we start...

!!! Lecture slides and example codes available on github !!!

https://github.com/sischei/global_solution_yale19.git

!!!Bring your Laptops to the classes!!!

Today: I need your names, NetIDs, and Yale emails to request you access on Yale's HPC infrastructure

1. Why Global Solution Methods?

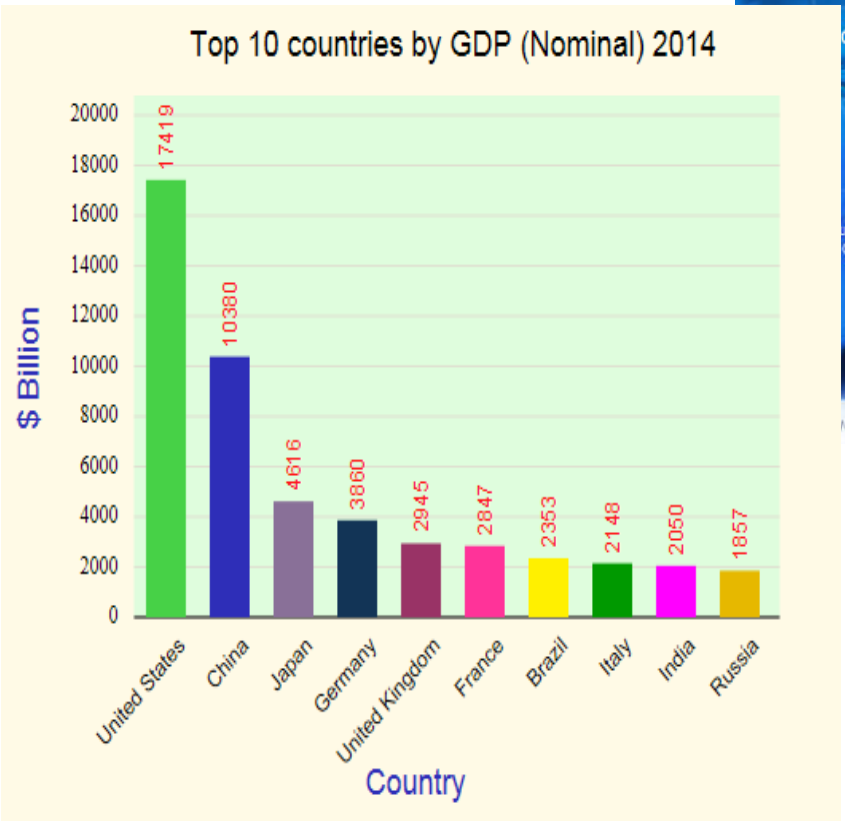
- Modern dynamic economic models such (e.g. DSGE) are extremely **rich** to capture **all the effects of interest**:
  - large stochastic shocks that lead to highly non-linear policies.
  - many agents that lead to a high-dimensional state space
  - ...
- Standard solution techniques such as log-linearization **often fail to deliver reliable results** across the entire domain of interest.
- The latter method may be useful to describe an economy that operates in normal times, but **fails** in the presence of nonlinearities such as occasionally binding constraints, among other types of salient features of the economic reality that the modelers would like to capture appropriately in their models.

Example – Heterogeneity in IRBC models

- Model trade imbalance

- FX rates

-...



- How many regions does a minimal model have?
- Are policy functions smooth? (borrowing constraints)

→ **Model heterogeneous & high-dimensional**

Example – Heterogeneity in OLG* models

*Overlapping generation models

To model e.g. social security:

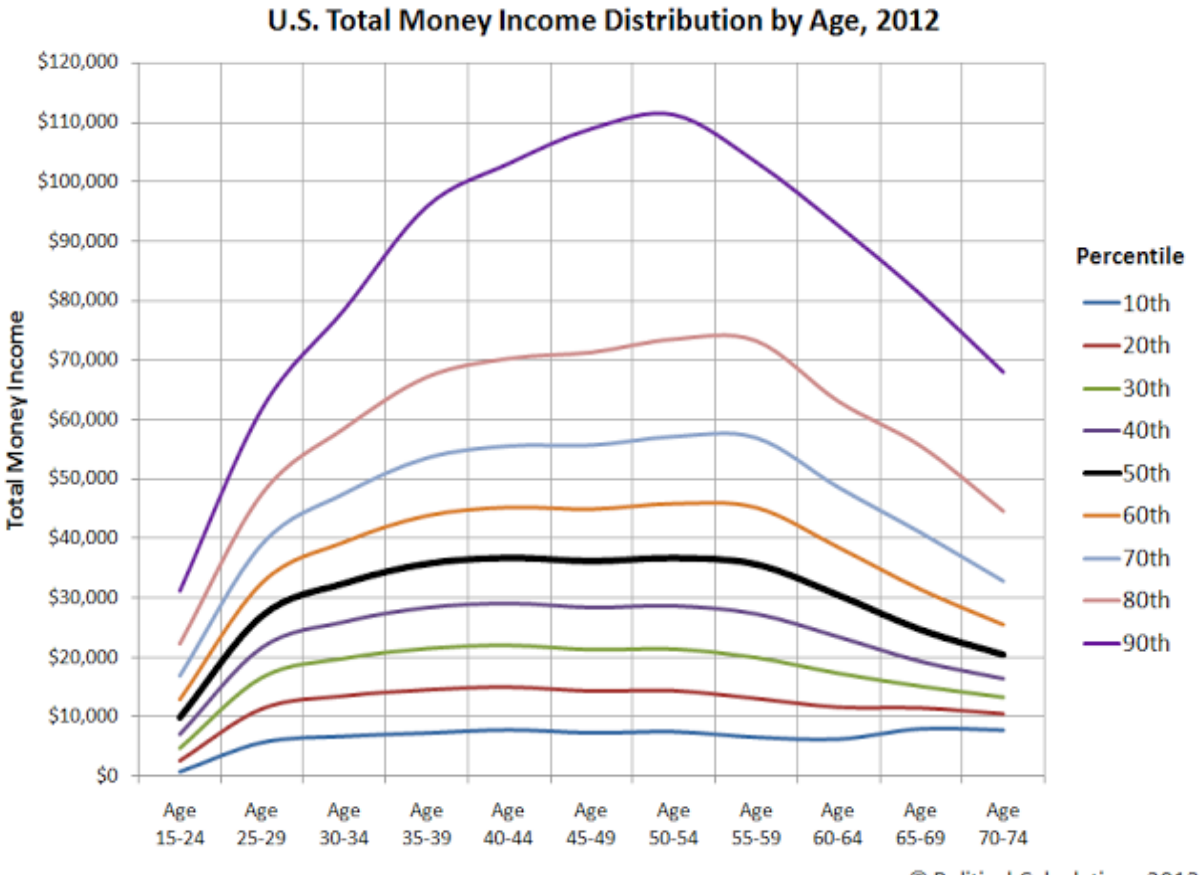
- How many age groups?

- borrowing constraints?

- aggregate shocks?

- …

→ Model: heterogeneous & high-dimensional



Financial markets: non-Gaussian returns

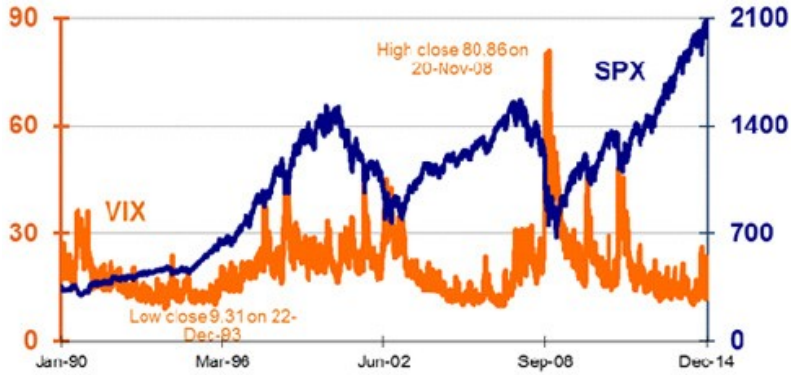
- Derivative contracts giving a right to buy or sell an underlying security.
  - *European* if exercise at expiration only.
  - ***American*** if exercise any time until expiration.

- American options are extremely challenging:
  → Dynamic optimization problem.

- Basic models do not describe dynamics accurately (e.g., Hull (2011)).

- Financial returns are often not Gaussian.

- Realistic models are hard to deal with, as they need many factors.
  → Curse of dimensionality.



Dynamic Programming/Value Function Iteration

e.g. Stokey, Lucas & Prescott (1989), Judd (1998), ...

Dynamic programming seeks a time-invariant policy function ***p*** mapping a state $\mathbf{x}_t$ into the control $\mathbf{u}_t$ such that for all $t \in \mathbb{N}\ u_t = p(x_t)$

The solution is approached in the limit as $j \longrightarrow \infty$ by iterations on:

$$V_{j+1}(x) = \max_u \{r(x,u) + \beta V_j(\tilde{x})\}$$

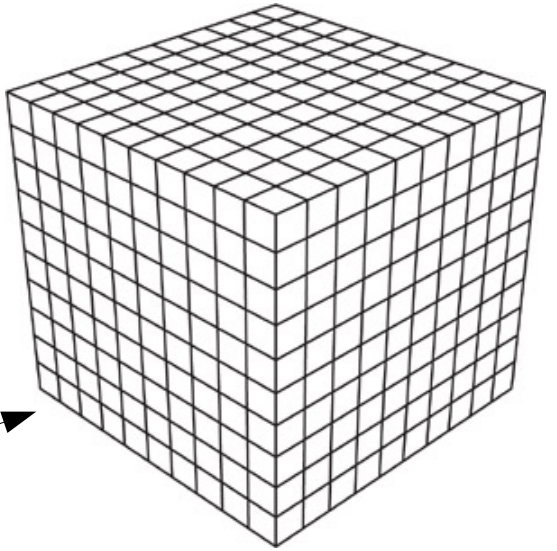
s.t.

$$\tilde{x} = g(x,u)$$

x: grid point, describes your system.
State-space potentially **high-dimensional**.

`old solution':
high-dimensional function on which **we interpolate**.

→ **N^d** points in ordinary discretization schemes.



→ Use-case for (adaptive) sparse grids.
→ Use-case for Gaussian Process regression.

How many is dimensions is high dimensions?

?

How many is dimensions is high dimensions?

Number of parameters (the dimension)	**Number of model runs** (at 10 points per dimension)	**Time for parameter study** (at 1 second per run)
1	10	10 sec
2	100	~ 1.6 min
3	1,000	~ 16 min
4	10,000	~ 2.7 hours
5	100,000	~ 1.1 days
6	1,000,000	~ 1.6 weeks
…	…	…
20	1e20	3 trillion years (240x age of the universe)

How many is dimensions is high dimensions?

Number of parameters (the dimension)	**Number of model runs** (at 10 points per dimension)	**Time for parameter study** (at 1 second per run)
1	10	10 sec
2	100	~ 1.6 min
3	1,000	~ 16 min
4	10,000	~ 2.7 hours
5	100,000	~ 1.1 days
6	1,000,000	~ 1.6 weeks
…	…	…
20	1e20	3 trillion years (240x age of the universe)

Dimension reduction
Exploit symmetries, e.g., via the active subspace method

Deal with #Points
Adaptive Sparse Grids

High-performance computing
Reduces time to solution, but not the problem size

A Crash-course in Python

Outline of this mini-course in Python

I. Motivation – why Python.

II. First steps in Python.

III. Nonlinear equations and optimization.

VI. Pointers to tutorials and literature.

Computational science in general

- **Computational science**: a rapidly growing multidisciplinary field that uses **advanced computing** capabilities to understand and solve complex problems.

- It is an area of science which spans many disciplines (comp. finance, comp. econ, comp. physics, comp. biology).

→ At its core it involves the **development of models** and **simulations** to understand complex systems.

→ computational science aims to make the complexity of those systems tractable.

Basics: von Neumann Architecture

https://computing.llnl.gov/tutorials/parallel_comp

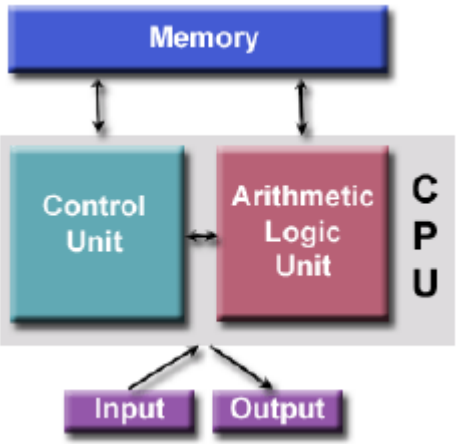
Virtually all computers have followed this basic design.
Comprised of **four main components**:
→ **Memory, Control Unit, Arithmetic Logic Unit, Input/Output.**

Read/write, **random access memory** is used to store both program instructions and data:
→ Program instructions are coded data which tell the computer to do something.
→ Data is simply information to be used by the program.

Control unit:
→ fetches instructions/data from memory, decodes the instructions and then sequentially coordinates operations to accomplish the programmed task.

Arithmetic Unit:
→ performs basic arithmetic operations.

Input/Output
→ interface to the human operator.



Hardware and how to use it

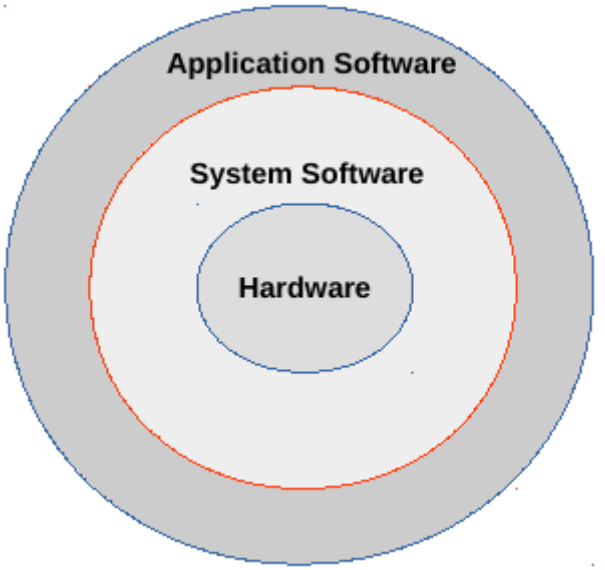
- The **hardware** can execute only simple **low-level instructions** (load,...)

- Complex applications (abstract models and their representation as high-level code) are **separated** from simple instructions by multiple layers.

- System Software

Operating System:
- I/O operation.
- Storage & memory.
- Protection among concurrent applications.

Compilers:
- Translate from high level language such as C++ to hardware instructions.



From a programming language to hardware

http://cse-lab.ethz.ch/index.php/teaching/42-teaching/classes/577-hpcsei

A computer is a “stupid” device, only understands “on” and “off”

The symbols for these states are 0 and 1 (binary)

First programmers communicated in 0 and 1.

Later programs where developed to translate from symbolic notation to binary. The first was called “**assembly**”.

> add A, B (programmer writes in assembly language)
> 1000110010100000 (assembly translates to machine language)

Advanced programming languages are better than “assembly” for

- → programmer thinks in a more natural language
- → productivity of software development
- → portability

“There are only two kinds of (programming) languages: the ones people complain about and the ones nobody uses.”

— Bjarne Stroustrup (designer of C++)

Lets complain about...

What is Python?

- Python is a **general purpose** programming language conceived in 1989 by Dutch programmer Guido van Rossum.

- Python is **free and open source**, with development coordinated through the **Python Software Foundation** https://www.python.org/psf/).

- Python has experienced rapid adoption in the last decade, and is now one of the most popular programming languages.

Common uses

Python is a general purpose language used in almost all application domains*:

- communications
- web development
- graphical user interfaces
- games, multimedia, data processing, security, etc.
- Machine Learning, Artificial Intelligence

Used extensively by high tech companies such as

- Google
- Dropbox
- YouTube
- Walt Disney Animation,...

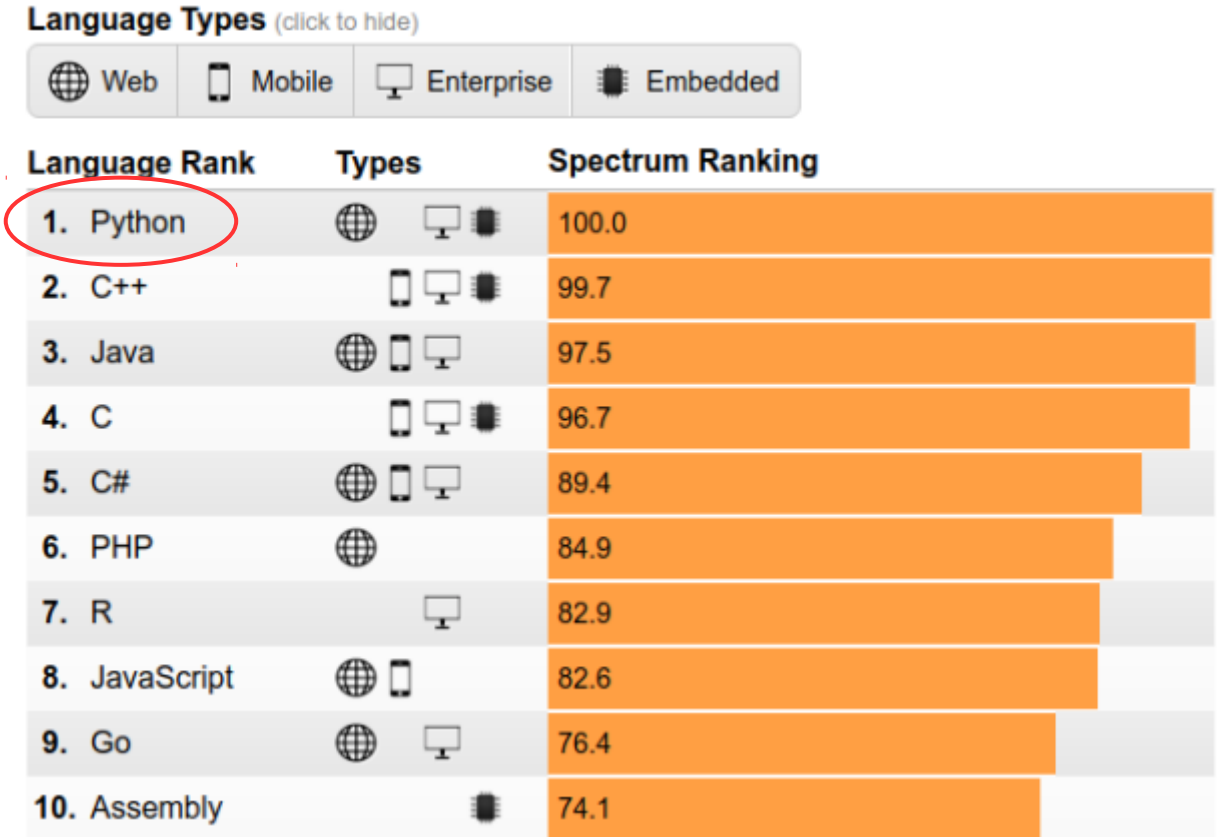
→ **Often used to teach computer science and programming**

→ **For reasons we will discuss, Python is particularly popular within the scientific community**

*See e.g. Quantecon for other good reasons: https://quantecon.org/

The Top Programming Languages 2018

https://spectrum.ieee.org/static/interactive-the-top-programming-languages-2018



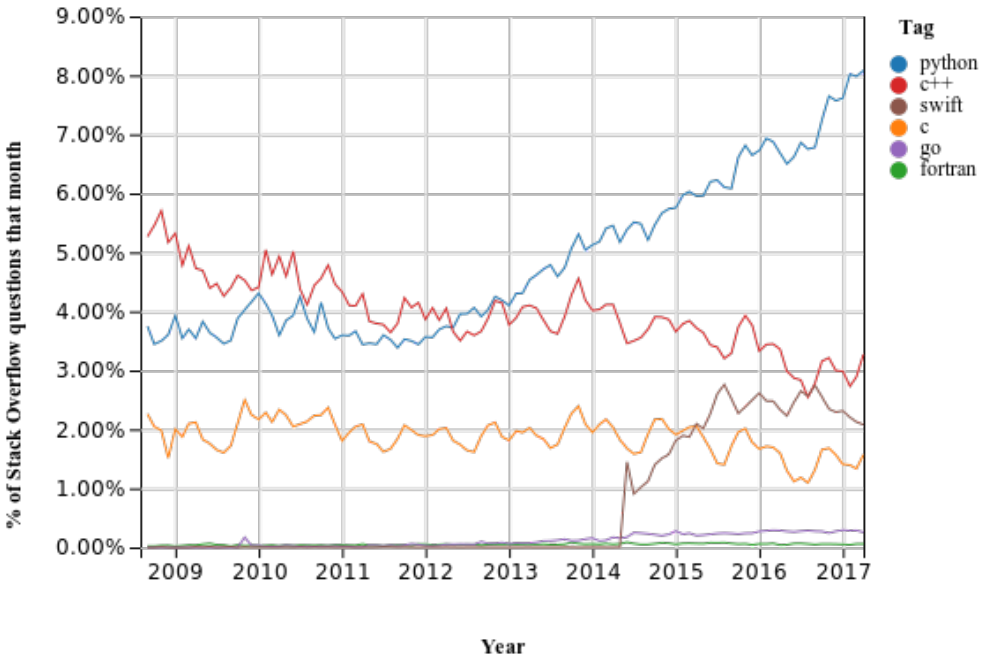
- Python is one of the most popular programming languages worldwide.

- Python is a major tool for scientific computing, accounting for a rapidly rising share of scientific work around the globe.

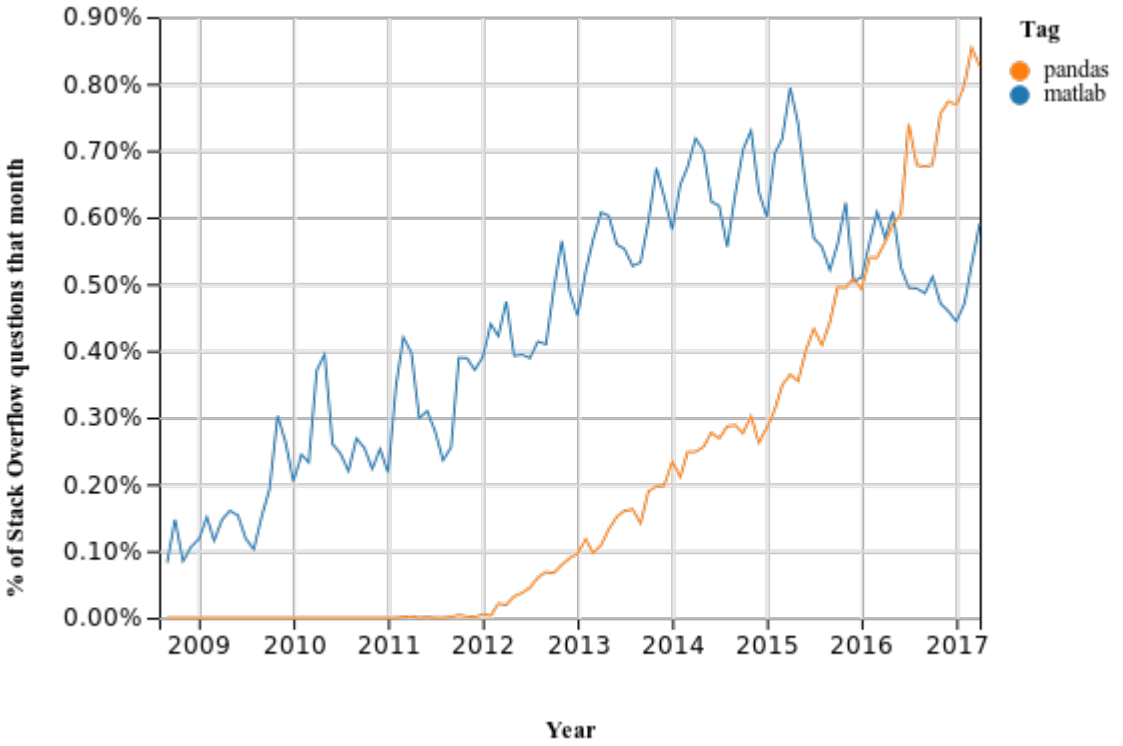
Relative popularity

https://stackoverflow.com/

- The following chart, produced using **Stack Overflow Trends**, shows one measure of the relative popularity of Python.
- The figure indicates not only that Python is widely used but also that adoption of Python has accelerated significantly since 2012.
- We suspect this is driven at least in part by uptake in the scientific domain, particularly in rapidly growing fields like **data science**.

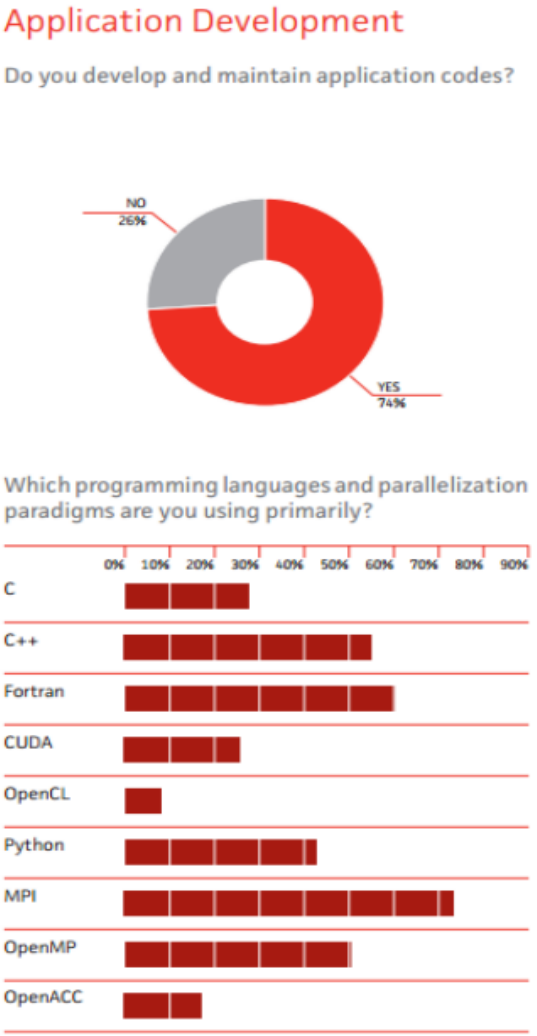


Matlab vs. Python – a replacement?



Statistics from supercomputer users

Source: CSCS annual report 2016



Even high-end users start to move there

Some features

- Python is a high level language suitable for rapid development.

- It has a relatively small core language **supported by many libraries**.

- Other features:
  A **multi-paradigm language**, in that multiple programming styles are supported (procedural, object-oriented, functional,...)

- **Interpreted rather than compiled**.

Syntax and design

One nice feature of Python is its **elegant syntax** — we'll see many examples later on.

Elegant code might sound superfluous but in fact it's highly beneficial because it makes the syntax **easy to read and easy to remember**.

Closely related to elegant syntax is elegant design.

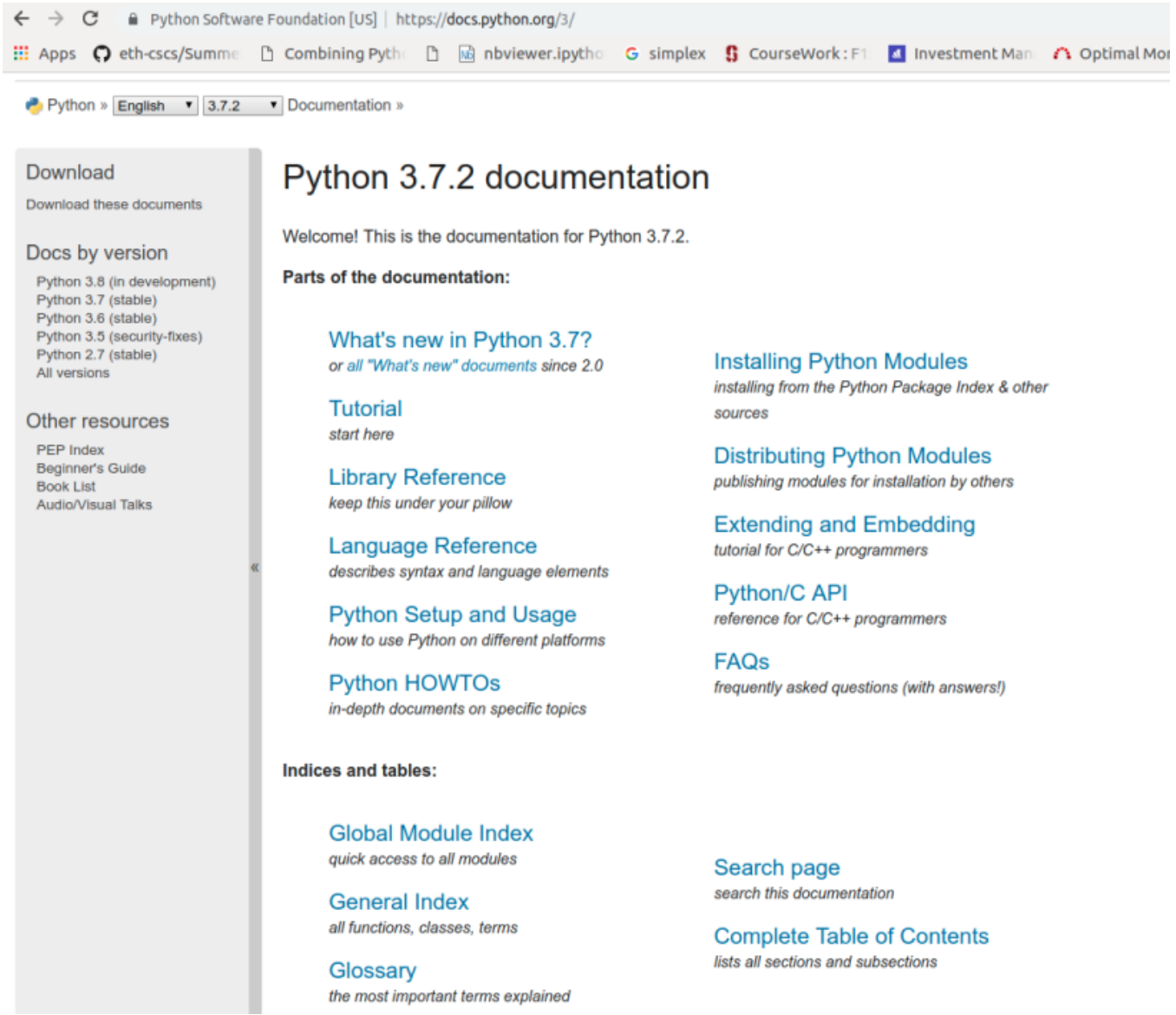
Features like iterators, generators, list comprehensions, etc. make Python highly expressive, **allowing you to get more done with less code**.

Get Python

- You can download and install Python directly from **https://www.python.org**

- Since we're going to use several libraries for numerical computation (numpy), data analysis (pandas), machine learning (scikit-learn), and visualization (matplotlib), it is easier to install Anaconda, which bundles all things required **https://www.continuum.io/downloads**

https://docs.python.org/



Install Python

https://www.python.org/downloads/

→ Find the installation you need (Linux, MacOS, Windows)

Python on Yale's **HPC compute cluster**

For this course, we use Yale's **HPC compute cluster**.

→ Its setup is very similar to any other top system

For the Manual see the documentation site at

https://research.computing.yale.edu/support/hpc/getting-started

https://research.computing.yale.edu/support/hpc/user-guide

→ Intro slides from last fall's lecture here:
global_solution_yale19/Lecture_1/Lecture_1_supplement_YaleHPC.pdf

Setting up the environment on GRACE

```
> vi ~/.bashrc      #here you can setup/store your profile
module load python #always load this lib upon login
```

For today, we may use a public Cloud

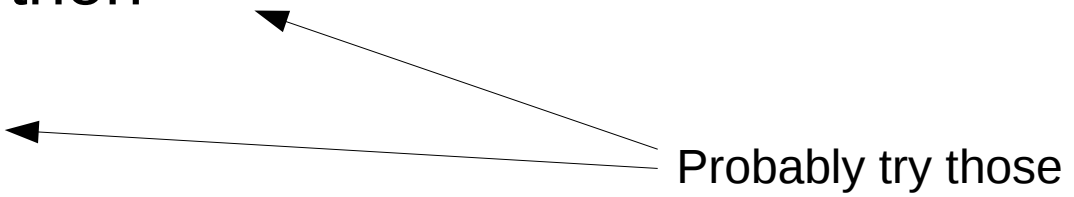
https://www.python.org/downloads/

→ **Today, use some on-line cloud version:**

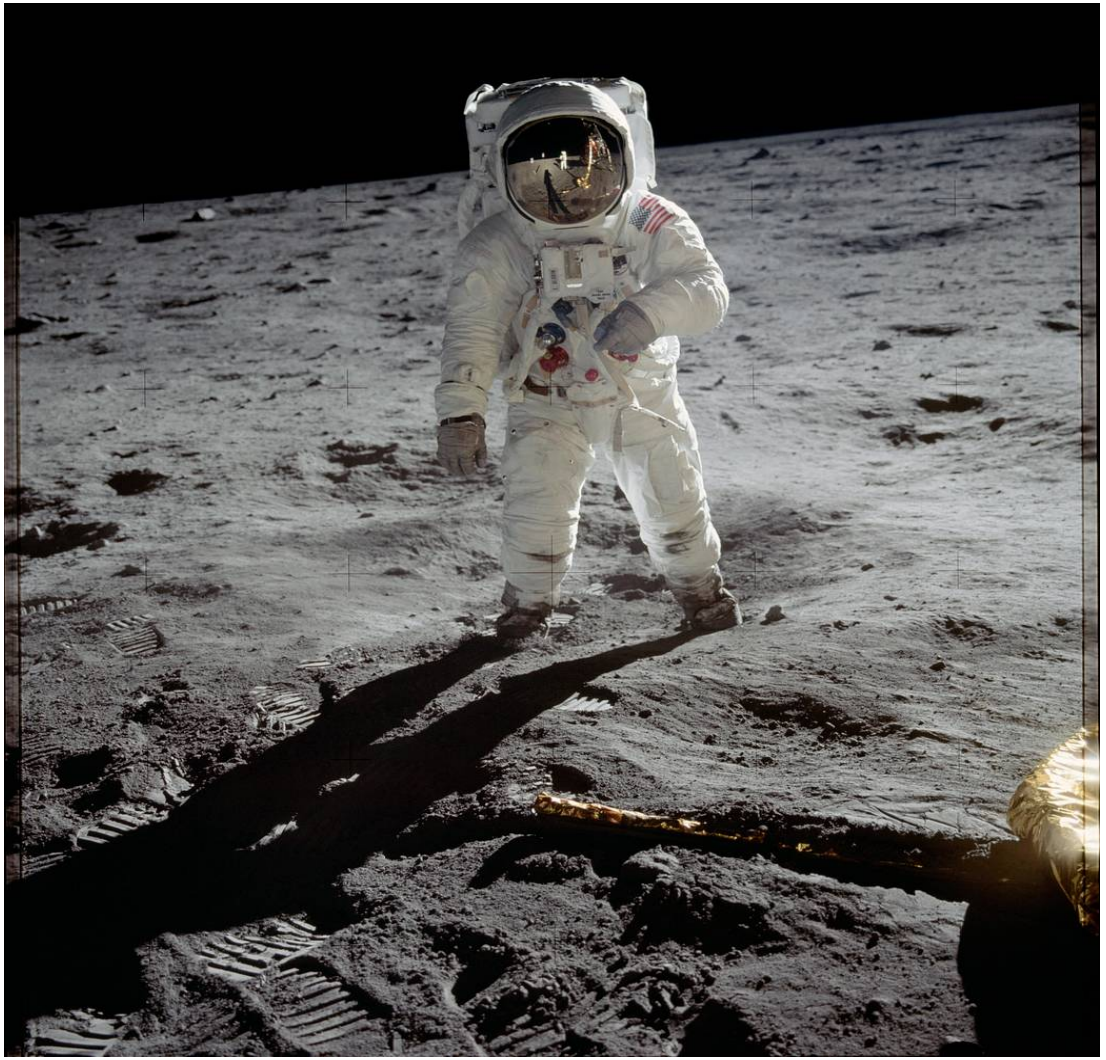
https://www.pythonanywhere.com/try-ipython/

https://repl.it/languages/python

https://trinket.io/python



II. First steps in Python
→ action required



List of examples

2.0. Setting up your environment.

2.1. Python basics.

2.2. Loops and Lists.

2.3. Functions and Branching.

2.4. Reading/writing Data.

Python setup

A bare-bones development environment consists of:

- A **text editor** (e.g., gedit, emacs, vim)
- The **Python interpreter** (it is installed by default on Ubuntu and almost any other Linux distribution)
- A **Terminal application** to run the interpreter in.

→ See https://wiki.python.org/moin/**I**ntegrated**D**evelopment**E**nvironments for a commented list of **IDE**s with Python support

Another helpful on-line tool

The **Online Python Tutor** is a free tool to visualize the
Online execution Python Tutor of - Visualize small program Python execution programs step-by-step.

Feel free to use it for the course exercises and your own code:
http://pythontutor.com/visualize.html

2.1. Python basics

Python is an *interpreted* language.

It also features an interactive "shell" for evaluating expressions and statements immediately.

The Python shell is started by invoking the command `python` in a terminal window.

```
simon@simon-ThinkPad-T450s:~$ python
Python 2.7.6 (default, Oct 26 2016, 20:30:19)
[GCC 4.8.4] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Python basics (II)

Expressions can be entered at the Python shell prompt (the '>>>' at the start of a line); they are evaluated and the result is printed:

```
>>> 2+2
4
```

A line can be continued onto the next by ending it with the character '\'; for example:

```
>>> "hello" + \
... " world!"
'hello world!'
```

The prompt changes to '. . .' on continuation lines.

Reference:

http://docs.python.org/reference/lexical_analysis.html#line-structure

Basic types

Basic object types in Python:

bool	The class of the two boolean constants `True`, `False`.
int	Integer numbers: `1`, `-2`, `...`
float	Double precision floating-point numbers, e.g.: `3.1415`, `-1e-3`.
str	Text (strings of byte-size characters).
list	Mutable list of Python objects
dict	Key/value mapping

→ **No type declaration needed – Python does that for you on the fly**

Type conversions

str(x)	Converts the argument x to a string; for numbers, the base 10 representation is used.
int(x)	Converts its argument x (a number or a string) to an integer; if x is a a floating-point literal, decimal digits are truncated.
float(x)	Converts its argument x (a number or a string) to a floating-point number.

→ **check the type of a variable by typing >>>type(a)**

Type conversions

str(x)	Converts the argument x to a string; for numbers, the base 10 representation is used.
int(x)	Converts its argument x (a number or a string) to an integer; if x is a a floating-point literal, decimal digits are truncated.
float(x)	Converts its argument x (a number or a string) to a floating-point number.

→ **check the type of a variable by typing >>>type(a)**

String literals

There are several ways to express string literals in Python.

Single and double quotes can be used interchangeably:

```
>>> "a string" == 'a string'
True
```

You can use the single quotes inside double-quoted strings, and viceversa:

```
>>> a = "Isn't it ok?"
>>> b = '"Yes", he said.'
```

String literals II

Multi-line strings are delimited by three quote characters.

```
>>> a = """This is a string,
... that extends over more
... than one line.
... """
```

In other words, you need not use the backslashes "\" at the end of the lines.

Operators

All the usual unary and binary arithmetic operators are defined in Python: `+`, `-`, `*`, `/`, `**` (exponentiation), `<<`, `>>`, etc.

Logical operators are expressed using plain English words: `and`, `or`, `not`.

Numerical and string comparison also follows the usual notation: `<`, `>`, `<=`, `==`, `!=`, ...

Reference:

- http://docs.python.org/library/stdtypes.html#boolean-operations-and-or-not
- http://docs.python.org/library/stdtypes.html#comparisons

Your first exercise to compute

Operators II

Some operators are defined for non-numeric types:

```
>>> "U" + 'ZH'
'UZH'
```

Some support operands of mixed type:

```
>>> "a" * 2
'aa'
>>> 2 * "a"
'aa'
```

Some do not:

```
>>> "aaa" / 3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for /: 'str' and 'int'
```

Operators III

The "%" operator computes the remainder of integer division.

```
>>> 9 % 2
1
```

Assignments

Assignment is done via the '=' statement:

```
>>> a = 1
>>> print(a)
1
```

There are a few shortcut notations:

- `a += b` short for `a = a + b`,
- `a -= b` short for `a = a - b`,
- `a *= b` short for `a = a * b`,

etc. — one for every legal operator.

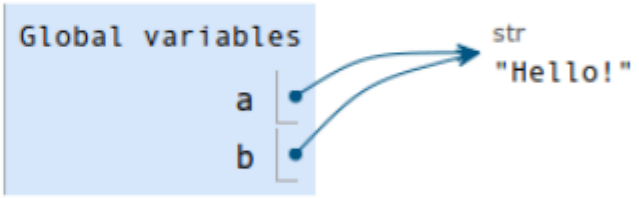
Assignments II

Try http://pythontutor.com/visualize.html

Python variables are just "names" given to values.
This allows you to *reference* the string `'Python'` by the *name* `a`:

```
1  a = "Hello!"
2  b = a
```

Edit code



The same object can be given many names!

The “is” operator

The `is` operator allows you to test whether two names refer to the same object:

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

The help function

help(`fn`) Display help on the function named `fn`

Q: *What happens if you type these at the prompt?*

- `help(abs)`
- `help(max)`

Functions

Functions are called by postfixing the function name with a parenthesized argument list.

```
>>> int("42")
42
>>> int(4.2)
4
>>> float(42)
42.0
>>> str(42)
'42'
>>> str()
''
```

Attention – rounding towards ZERO

How to execute source code

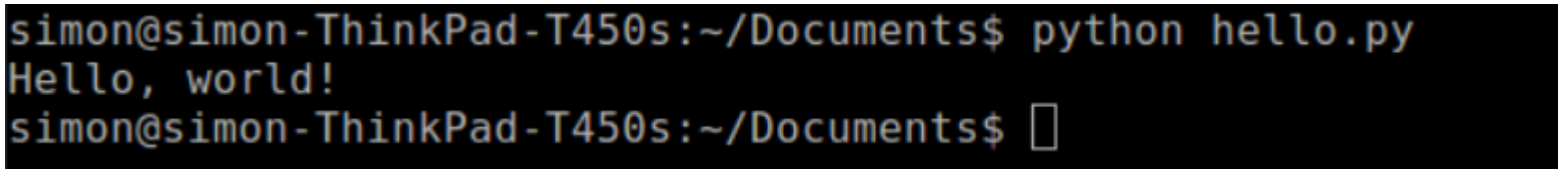
global_solution_yale19/Lecture_1/code/basic_examples/example0_hello.py

→ Store a file ***hello.py*** with this content:

```
print('Hello, world!')
```

→ Execute it with

```
$ python hello.py
```



2.2. Loops and lists

global_solution_yale19/Lecture_1/code/basic_examples/example3_while.py

The **while loop**:
→ is used to repeat a set of statements as long as a condition is true.
→ **Note: Indentation matters in Python!!!**

Example:
The task is to generate the rows of the table of C and F values. The C value starts at **−20 and is incremented by 5 as long as C ≤ 40**. For each C value we compute the corresponding F value and write out the two temperatures. In addition, we also add a line of hyphens above and below the table. We postpone to nicely format the C and F columns of numbers and perform for simplicity a plain print C, F statement inside the loop.

```
print '------------------'     # table heading
C = -20                       # start value for C
dC = 5                        # increment of C in loop
while C <= 40:                # loop heading with condition
    F = (9.0/5)*C + 32        # 1st statement inside loop
    print C, F                # 2nd statement inside loop
    C = C + dC                # 3rd statement inside loop
print '------------------'     # end of table line (after loop)
```

$python example3_while.py

```
------------------
-20 -4.0
-15 5.0
-10 14.0
-5 23.0
0 32.0
5 41.0
10 50.0
15 59.0
20 68.0
25 77.0
30 86.0
35 95.0
40 104.0
------------------
```

Loop Implementation of a Sum

global_solution_yale19/Lecture_1/code/basic_examples/example4_sum.py

$$\sin(x) \approx x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \cdots$$

```
x = 1.2  # assign some value
N = 25   # maximum power in sum
k = 1
s = x
sign = 1.0
import math

while k < N:
    sign = - sign
    k = k + 2
    term = sign*x**k/math.factorial(k)
    s = s + term

print 'sin(%g) = %g (approximation with %d terms)' % (x, s, N)
```

Lists

Up to now a variable has typically contained a single number.

Sometimes numbers are naturally grouped together.

A Python **list** can be used to represent such a group of numbers in a program.

With a variable that refers to the list, we can work with the whole group at once, but we can also access individual elements of the group.

The figure illustrates the difference between an int object and a list object.

In general, a list may contain a sequence of arbitrary objects in a given order. Python has great functionality for examining and manipulating such sequences of objects.

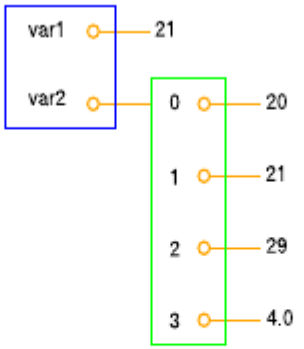


Fig. 2.1 Illustration of two variables: `var1` refers to an `int` object with value 21, created by the statement `var1 = 21`, and `var2` refers to a `list` object with value `[20, 21, 29, 4.0]`, i.e., three `int` objects and one `float` object, created by the statement `var2 = [20, 21, 29, 4.0]`.

Fig. From H.-P. Langtangen's book

Basic operations in Lists → type

To **create a list** with the numbers from the first column in our table, we just put all the **numbers inside square brackets** and separate the numbers by commas:

```
>>> C = [-10, -5, 0, 5, 10, 15, 20, 25, 30]     # create list
>>> C.append(35)                 # add new element 35 at the end
>>> C                            # view list C
[-10, -5, 0, 5, 10, 15, 20, 25, 30, 35]
```

Two lists can be added:

```
>>> C = C + [40, 45]             # extend C at the end
>>> C
[-10, -5, 0, 5, 10, 15, 20, 25, 30, 35, 40, 45]
```

New elements can be inserted anywhere in the list:

```
>>> C.insert(0, -15)           # insert new element -15 as index 0
>>> C
[-15, -10, -5, 0, 5, 10, 15, 20, 25, 30, 35, 40, 45]
```

With **del C[i]** we can remove an element with index i from the list C.

```
>>> del C[2]                    # delete 3rd element
>>> C
[-15, -10, 0, 5, 10, 15, 20, 25, 30, 35, 40, 45]
>>> del C[2]                    # delete what is now 3rd element
>>> C
[-15, -10, 5, 10, 15, 20, 25, 30, 35, 40, 45]
>>> len(C)                      # length of list
11
```

How to represent Vectors

A Python program may use a **list** or **tuple** to represent a vector:

```
v1 = [x, y]          # list of variables
v2 = (-1, 2)         # tuple of numbers
v3 = (x1, x2, x3)    # tuple of variables
from math import exp
v4 = [exp(-i*0.1) for i in range(150)]
```

Tuples ~ constant lists (not really true, but to memorize, this zero-order approach may help)

Basic operations (II) – “for loop”

- When data are collected in a list, we often want to perform the same operations on each element in the list.

- We then need to walk through all list elements. Computer languages have a special construct for doing this conveniently.

- This construct in Python and many other languages called a **“for loop”**.

example5_forloop.py

```
degrees = [0, 10, 20, 40, 100]
for C in degrees:
    print 'list element:', C
print 'The degrees list has', len(degrees), 'elements'
```

```
index = 0
while index < len(somelist):
    element = somelist[index]
    <process element>
    index += 1
```

example6_forloop.py

```
Cdegrees = []
n = 21
C_min = -10
C_max = 40
dC = (C_max - C_min)/float(n-1)  # increment in C
for i in range(0, n):
     C = -10 + i*dC
     Cdegrees.append(C)

Fdegrees = []
for C in Cdegrees:
    F = (9.0/5)*C + 32
    Fdegrees.append(F)

for i in range(len(Cdegrees)):
    C = Cdegrees[i]
    F = Fdegrees[i]
    print '%5.1f %5.1f' % (C, F)
```

Example: Sum with default tolerance

global_solution_yale19/Lecture_1/code/basic_examples/example8_sum.py

```
def L2(x, epsilon=1.0E-6):
    x = float(x)
    i = 1
    term = (1.0/i)*(x/(1+x))**i
    s = term
    while abs(term) > epsilon:   # abs(x) is |x|
        i += 1
        term = (1.0/i)*(x/(1+x))**i
        s += term
    return s, i
```

We can use the first neglected term as an estimate of the accuracy.

Here is an example involving this function to make a table of the approximation error as decreases:

```
from math import log
x = 10
for k in range(4, 14, 2):
    epsilon = 10**(-k)
    approx, n = L2(x, epsilon=epsilon)
    exact = log(1+x)
    exact_error = exact - approx
    print 'epsilon: %5.0e, exact error: %8.2e, n=%d' % \
          (epsilon, exact_error, n)
```

The output becomes

```
epsilon: 1e-04, exact error: 8.18e-04, n=55
epsilon: 1e-06, exact error: 9.02e-06, n=97
epsilon: 1e-08, exact error: 8.70e-08, n=142
epsilon: 1e-10, exact error: 9.20e-10, n=187
epsilon: 1e-12, exact error: 9.31e-12, n=233
```

2.3. Functions and Branching

The **def** statement starts a function definition.

```
def greet(name):
  """
  A friendly function.
  """
  print ("Hello, " + name + "!")

# the customary greeting
greet("world")
```

How to define new functions (II)

```
def greet(name):
    """
    A friendly function.
    """
    print ("Hello, " + name + "!")

# the customary greeting
greet("world")
```

Indentation is significant in Python: it is used to delimit blocks of code, like '{' and '}' in Java and C.

How to define new functions (III)

```
def greet(name):
    """
    A friendly function.
    """
    print ("Hello, " + name + "!")

# the customary greeting
greet("world")
```

(This is a comment. It is ignored by Python, just like blank lines.)

How to define new functions (IV)

global_solution_yale19/Lecture_1/code/basic_examples/example7_greet.py

This calls the function just defined.

```
def greet(name):
    """
    A friendly function.
    """
    print ("Hello, " + name + "!")

# the customary greeting
greet("world")
```

How to define new functions (V)

```
def greet(name):
    """
    A friendly function.
    """
    print ("Hello, " + name + "!")

# the customary greeting
greet("world")
```

What is this? The answer in the next exercise!

Try this:
`>>>help(greet)`

Modules

The `import` statement reads a `.py` file, executes it, and makes its contents available to the current program.

```
>>> import hello
Hello, world!
```

Assume your file is called hello.py

Modules are only read once, no matter how many times an `import` statement is issued.

Modules II

Modules are *namespaces:* functions and variables defined in a module must be prefixed with the module name when used in other modules:

```
>>> hello.greet("Bob")
Hello, Bob!
```

To import definitions into the current namespace, use the '`from` *x* `import` *y*' form:

```
>>> from fractions import Fraction
```

Exercises

Exercise A: Type and run the code on the previous page at the interactive prompt. (Type indentation spaces, too!)
What does `help(greet)` print? What's the result of evaluating the function `greet("world")`?

Exercise B: Type the same code in a file named `hello.py`, then type `import hello` at the interactive prompt. What happens?

Conditionals

Conditional execution uses the `if` statement:

```
if expr:
   # indented block
elif other-expr:
   # indented block
else:
   # executed if none of the above matched
```

The `elif` can be repeated, with different conditions, or left out entirely.

Also the `else` clause is optional.

Q: *Where's the 'end if'?*
There's no 'end if': indentation delimits blocks!

Branching – example “hat” function

global_solution_yale19/Lecture_1/code/basic_examples/example9_branching.py

Branching in general

```
if condition1:
    <block of statements>
elif condition2:
    <block of statements>
elif condition3:
    <block of statements>
else:
    <block of statements>
<next statement>
```

$$N(x) = \begin{cases} 0, & x < 0 \\ x, & 0 \leq x < 1 \\ 2 - x, & 1 \leq x < 2 \\ 0, & x \geq 2 \end{cases}$$

```
def N(x):
    if x < 0:
        return 0.0
    elif 0 <= x < 1:
        return x
    elif 1 <= x < 2:
        return 2 - x
    elif x >= 2:
        return 0.0
```

2.4. Input Data.

```
C = 21
F = (9/5)*C + 32
print F
```

In this program, **C is input data** in the sense that C must be known before the program can perform the calculation of F.

The results produced by the program, here F, constitute the output data. Input data can be **hard-coded** in the program as we do above.

We explicitly set variables to specific values (C = 21).

This programming style may be suitable for small programs.

In general, however, it is considered good practice to let a user of the program **provide input data when the program is running**.

→ There is then no need to modify the program itself when a new set of input data is to be explored.

Reading Keyboard Input

We may ask the user a question C=? and wait for the user to enter a number. The program can then read this number and store it in a variable C.

```
C = raw_input('C=? ')
C = float(C)
F = (9./5)*C + 32
print F
```

example10_read.py

The **raw_input** function always returns the user input as a **string object**. That is, the variable C above refers to a string object.

If we want to compute with this C, we must convert the string to a floating-point number: C = float(C).

Reading from Command line

global_solution_yale19/Lecture_1/code/basic_examples/example11_readsys.py

Inside the program we can fetch the text “number” as **sys.argv[1]**.

The **sys** module has a list **argv** containing all the command-line arguments to the program, i.e., all the “words” appearing after the program name when we run the program.

Here there is only one argument and it is stored with index 1. The first element in the sys.argv list, **sys.argv[0], is always the name of the program**.

A command-line argument is treated as a text, so **sys.argv[1] refers to a string object**. Since we interpret the command-line argument as a number and want to compute with it, it is **necessary to explicitly convert the string to a float object**.

Run as:
$python example11_readsys.py 2

```
import sys
print "This is the name of the script: ", sys.argv[0]

print "please write the degrees celcius outside:"
C = sys.argv[1]

F= 9*float(C)/5 + 32
print "it is ", F , "  degrees F"


print "Number of arguments: ", len(sys.argv)
print "The arguments are: " , str(sys.argv)
```

Reading from a file

global_solution_yale19/Lecture_1/code/basic_examples/example12_readfile.py

→ We have a text file containing numbers: say **data.txt**

→ We want to get first column in a1, second in a2 and so on.

→ Note: there are plenty of option on how to read from files → RTFM

```
1  2  3  4
5  6  7  8
9  10  11  12
13  14  15  16
```

```
a1 = []
a2 = []
a3 = []
a4 = []

with open('data.txt') as f:
    for line in f:
        data = line.split()

a1.append(int(data[0]))

a2.append(int(data[1]))

a3.append(int(data[2]))

a4.append(int(data[3]))

print a1, a2, a3, a4
f.close()
```

Other things to learn about Python

The time and scope of this course is rather limited.

Here is an (incomplete) list of Python features that you might want to look up as you become more experienced in the language:

- Generators and Iterators
- Decorators
- Class-level attributes, classmethods, staticmethods
- Properties and accessors
- Metaclasses

Do not re-invent the wheel – NumPy

NumPy is a package for linear algebra and advanced mathematics in Python.

It provides a *fast* implementation of multidimensional numerical arrays (C/FORTRAN like), vectors, matrices, tensors and operations on them.

Use it if: you long for MATLAB core features.

See also: http://www.numpy.org/

Do not re-invent the wheel – SciPy

"SciPy is open-source software for mathematics, science, and engineering. *[. . .]* The SciPy library provides many user-friendly and efficient numerical routines such as routines for numerical integration and optimization."

One of its main aim is to provide a reimplementation of the MATLAB toolboxes.

Use it if: you long for MATLAB toolbox features.

See also: http://www.scipy.org/

Pandas

Pandas is a Python data analysis library, that provides optimized routines for analyzing 2D, 3D, 4D data.

"Pandas *[. . .]* enables you to carry out your entire data analysis workflow in Python without having to switch to a more domain specific language like R."

Use it if: you need features from *R*, `plyr`, `reshape2`.

Writing to a file – e.g. with **numpy**

global_solution_yale19/Lecture_1/code/basic_examples/example12_readfile.py

```
import numpy as np
mat=np.matrix([[1, 2, 3],[4, 5, 6],[7, 8, 9]])
print mat
np.savetxt('matrix.txt',mat,fmt='%.2f')
```

Curve Plotting

- Visualizing a function f (x) is done by drawing the curve y = f (x) in an x-y coordinate system.

- When we use a computer to do this task, we say that we plot the curve.

- Technically, we plot a curve by drawing straight lines between n points on the curve.

- The more points we use, the smoother the curve appears.

- Suppose we want to plot the function f (x) for a ≤ x ≤ b.

$$x_i = a + ih, \quad h = \frac{b-a}{n-1}.$$

Matplotlib: publication quality plotting library

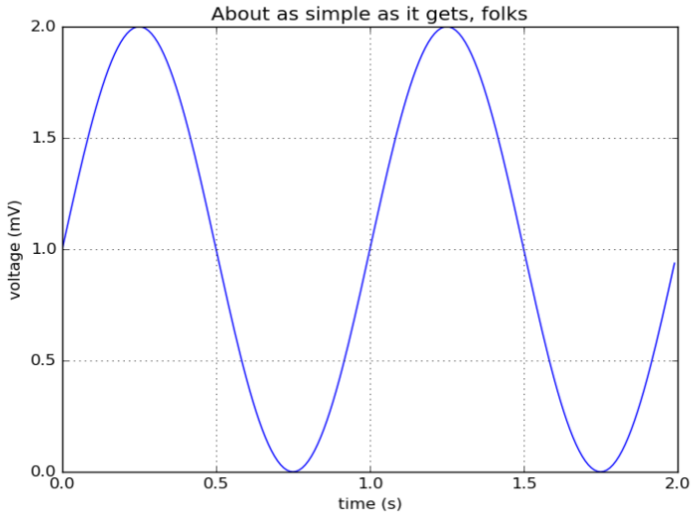
matplotlib "is a python 2D plotting library which produces publication quality figures in a variety of hardcopy formats and interactive environments across platforms. matplotlib can be used in python scripts, the python and ipython shell (ala MATLAB® or Mathematica®), web application servers, and six graphical user interface toolkits."

Matplotlib: a basic example

global_solution_yale19/Lecture_1/code/basic_examples/example13_plot.py

Let us plot the curve ***s*** for values between 0 and 2. First we generate equally spaced coordinates for t. Then we compute the corresponding ***s*** values at these points, before we call the plot(t,s) command to make the curve plot.

```
import matplotlib.pyplot as plt
import numpy as np

t = np.arange(0.0, 2.0, 0.01)
s = 1 + np.sin(2*np.pi*t)
plt.plot(t, s)

plt.xlabel('time (s)')
plt.ylabel('voltage (mV)')
plt.title('About as simple as it gets,
folks')
plt.grid(True)
plt.savefig("test.png")
plt.show()
```



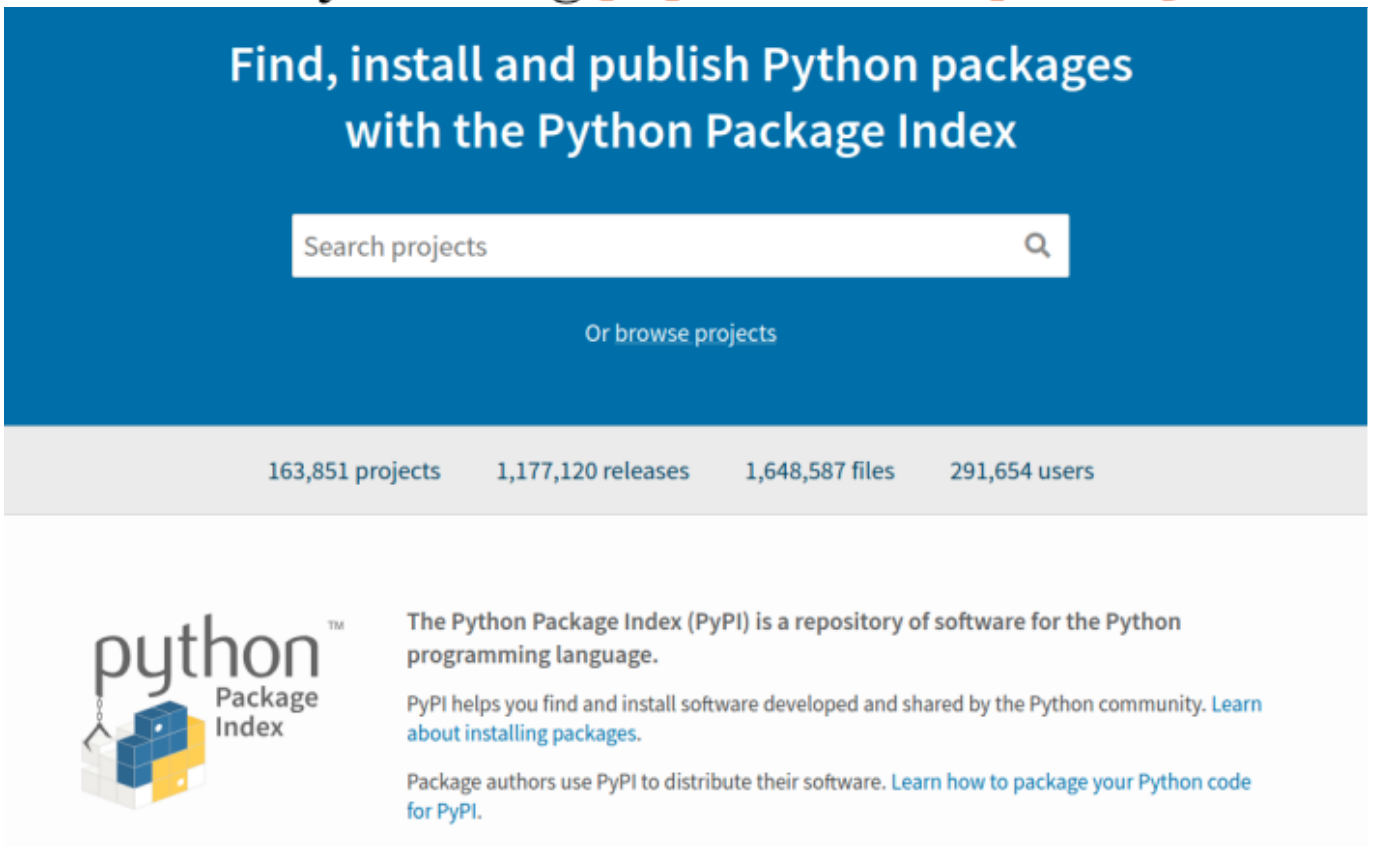
To include the plot in electronic documents, we need a hardcopy of the figure in PostScript, PNG, or another image format. The savefig function saves the plot to files in various image formats.

Want more?

PyPI is *the* index of Python software packages.

It currently indexes **163,851** packages, so the choice is really vast.

Almost all packages can be installed with a single command by running `pip install packagename`.



III. Nonlinear equations & optimization.

- Quantitative economics heavily relies on solving large systems of nonlinear equations or (un-) constraint optimization problems both in static as well as in dynamic models (see, e.g., dynamic programming, time iteration,...).

→ In Python, you have plenty of options, e.g.:

- SciPy.org

- PyOpt.org

- IPOPT (https://www.coin-or.org/Ipopt; https://github.com/xuy/pyipopt)

Constrained optimization with SciPy

The minimize function also provides an interface to several constrained minimization algorithm.

As an example, the Sequential Least SQuares Programming optimization algorithm (SLSQP) will be considered here.

This algorithm allows to deal with constrained minimization problems of the form:

$$
\begin{aligned}
&\min F(x) \\
&\text{subject to} \quad C_j(X) = 0, \quad j = 1, \ldots, \text{MEQ} \\
&\qquad\qquad\quad C_j(x) \geq 0, \quad j = \text{MEQ} + 1, \ldots, M \\
&\qquad\qquad XL \leq x \leq XU, \; I = 1, \ldots, N.
\end{aligned}
$$

Constrained optimization – example

As an example, let us consider the problem of optimizing the function:

$$f(x, y) = 2\,x\,y + 2\,x - x^2 - 2\,y^2$$

subject to an equality and an inequality constraints defined as:

$$x^3 - y = 0$$
$$y - 1 \geq 0$$

Example for Optimization

global_solution_yale19/Lecture_1/code/opt_nonlinear/example1_opt.py

```
import numpy as np
from scipy.optimize import minimize


def func(x, sign=1.0):
  """ Objective function """
  return sign*(2*x[0]*x[1] + 2*x[0] - x[0]**2 - 2*x[1]**2)


def func_deriv(x, sign=1.0):
  """ Derivative of objective function """
  dfdx0 = sign*(-2*x[0] + 2*x[1] + 2)
  dfdx1 = sign*(2*x[0] - 4*x[1])
  return np.array([ dfdx0, dfdx1 ])

"""
Note that since minimize only minimizes functions, the sign parameter is
introduced to multiply the objective function (and its derivative) by -1 in order to perform a maximization.

Then constraints are defined as a sequence of dictionaries, with keys type, fun and jac.
"""
cons = ({'type': 'eq',
         'fun' : lambda x: np.array([x[0]**3 - x[1]]),
         'jac' : lambda x: np.array([3.0*(x[0]**2.0), -1.0])},
        {'type': 'ineq',
        'fun' : lambda x: np.array([x[1] - 1]),
         'jac' : lambda x: np.array([0.0, 1.0])})
»
»
»
"""A constrained optimization as:"""

res = minimize(func, [-1.0,1.0], args=(-1.0,), jac=func_deriv,
»              constraints=cons, method='SLSQP', options={'disp': True})

print(res.x)
```

Root finding (nonlinear equations)

Finding a root of a set of non-linear equations can be achieve using the root function.

Several methods are available, amongst which ***hybr*** (the default) and ***lm*** which respectively use the hybrid method of Powell and the Levenberg-Marquardt method from MINPACK.

Consider a set of non-linear equations

$$x_0 \cos(x_1) = 4,$$
$$x_0 x_1 - x_1 = 5.$$

Example for Nonlinear Equations

global_solution_yale19/Lecture_1/code/opt_nonlinear/example2_nonlinear.py

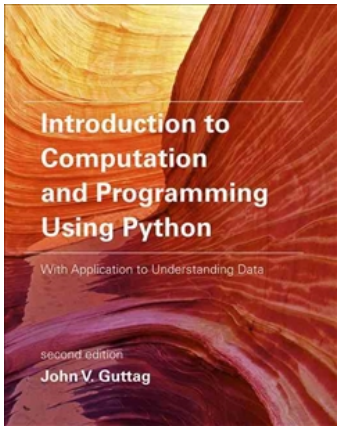
```
import numpy as np
from scipy.optimize import root

def func2(x):
  f = [x[0] * np.cos(x[1]) - 4, x[1]*x[0] - x[1] - 5]
  df = np.array([[np.cos(x[1]), -x[0] * np.sin(x[1])],[x[1], x[0] - 1]])
  return f, df


sol = root(func2, [1, 1], jac=True, method='lm')
solution = sol.x

print "the solution of this nonlinear set of equations is:", solution
```

IV. Some source materials

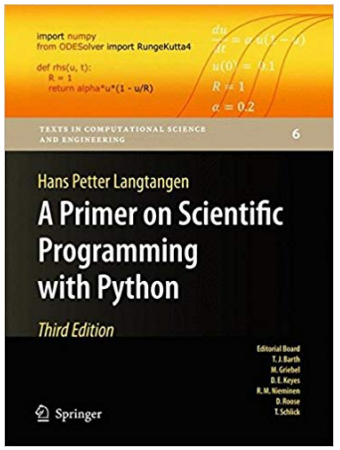
I will not follow a textbook, but will point in my slides to the relevant literature.

Some useful introductory textbooks:

Introduction to Computaiton and Programming using Python
J. Guttag. MIT Press, 2013
https://www.cs.ubc.ca/~murphyk/MLbook/index.html



A Primer on Scientific Programming with Python
H.-P. Langtangen



Moreover, for Python, there is almost an unlimited amount of very good on-line tutorials available, e.g.,
https://python-textbok.readthedocs.io/en/1.0/Errors_and_Exceptions.html
https://www.manning.com/books/get-programming

Pointers to tutorials and literature

There is an unlimited amount of tutorials and source codes on the web available. Here is an incomplete list:

The Python tutorial:

http://docs.python.org/tutorial/

The Zen of Python in 3 days:

http://pixelmonkey.org/pub/python-training/

Python for Java programmers:

http://python4java.necaiseweb.org/Main/TableOfContents

Youtube:

https://forums.kjdelectronics.com/blog/?page_id=48 (links inthere)

Python Material for Economics

https://quantecon.org/ 

https://econ-ark.org/ 

...

Questions?

1. Advice – RTFM
https://en.wikipedia.org/wiki/RTFM

2. Advice – http://lmgtfy.com/
http://lmgtfy.com/?q=introduction+to+python