


Deeper Python

Tools you would want to use



8+ t in f View Your Cart

HOME PRODUCTS CONSULTING TRAINING COMPANY CONTACT US


Anaconda

Completely free enterprise-ready Python distribution for large-scale data processing, predictive analytics, and scientific computing

- 195+ of the most popular Python packages for science, math, engineering, data analysis
- Completely free - including for commercial use and even redistribution
- Cross platform on Linux, Windows, Mac
- Installs into a single directory and doesn't affect other Python installations on your system. Doesn't require root or local administrator privileges
- Stay up-to-date by easily updating packages from our free, online repository
- Easily switch between Python 2.6, 2.7, 3.3, 3.4, and experiment with multiple versions of libraries, using our conda package manager and its great support for virtual environments
- Comes with tools to connect and integrate with Excel

Download Anaconda

Full Version is Completely Free



Anaconda

Anaconda Add-Ons

Accelerate	\$129.00	Free Trial
IOPro	\$79.00	Free Trial
MKL Optimizations	\$29.00	Free Trial

All Products are Free for Academic Use

Anaconda Server

Manage the deployment of Python, R, and internal packages behind firewalls and proxies

Learn More

How Can We Help?

Why Are We Just Giving This Away?

- We want to ensure that Python, NumPy, SciPy, Pandas, IPython, Matplotlib, Numba, Blaze, Bokeh, and other great Python data analysis tools can be used everywhere.
- We want to make it easier for Python evangelists and teachers to promote the use of Python.
- We want to give back to the Python community that we love being a part of.

But all of this takes hard work and resources! Help us out -- Check out our products, sign up for our virtual and on-site courses, and contact us about doing data science or SciPy/NumPy consulting project!

Using Anaconda in a professional environment? Check out **Anaconda Server** to take control of the deployment and management of Python, R, and internal packages behind your firewall and proxy. Integration tools and install support included.

Download Anaconda

Please note: Anaconda comes with installers for Python 2.7 and 3.4. Python 2.6 and 3.3 are available through the conda command.

Get Started with the Anaconda Quick Start Guide [pdf]

Features

<https://store.continuum.io/cshop/anaconda/>



PyCharm

Overview What's New Features & Screenshots Docs & Demos Quickstart Guide Download Buy & Renew

The Most Intelligent Python IDE

Enjoy productive Python, Django, and Web development with PyCharm, an intelligent Python IDE offering unique coding experience.

Get PyCharm Now

Full-fledged Professional or Free Community



Intelligent Coding Assistance

Smart Code Navigation

Fast and Safe Refactoring

Debugging and Testing

VCS and Deployment

<https://www.jetbrains.com/pycharm/download/>

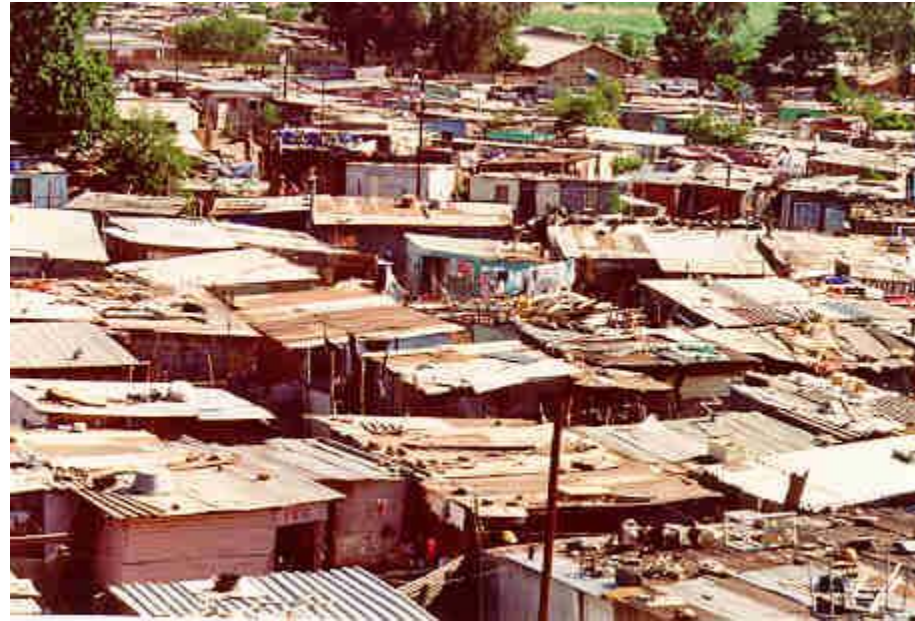
Common programming
background

Code to solve a problem

- Decide what you want to do next, google, paste code, repeat
 - 1) “Get the results as quickly as possible”
 - 2) “Since you’ve already implemented it, why can’t you just modify a bit and re-use it?”
- Requires least investment to get the initial results
- Most exciting way of coding
- Also the most frustrating in the long-term

Spaghetti obfuscated code

<http://www.laputan.org/mud/>



Things will get complicated

Size and complexity of the codebase will increase

So will the number of contributors

Performance improvements will be required

Code will need to be maintained months & years later

The code will be re-used for other projects

“Complexity increases rapidly until the it reaches a level of complexity just beyond that with which programmers can comfortably cope. At this point, complexity and our abilities to contain it reach an uneasy equilibrium. The blitzkrieg bogs down into a siege. We built the most complicated system that can possible work”

- Brian Foote and Joseph Yoder

Programming = juggling complexity

Spaghetti code is post-process complexity:

- Impossible to re-use

- Impossible to analyse

Waterfall development* is pre-process complexity

- Contradicting, constantly changing specifications

- Obsolete before they are implemented

*Planning everything ahead

Code is Part of Research Work

No matter what “real” scientists think about it

- “Nobody reads crappy research papers. That's why they are peer-reviewed, proof-read, refined, rewritten, and approved time and time again until deemed ready for publication. The same applies to a thesis **and a codebase!**”

<http://programmers.stackexchange.com/questions/155488/ive-inherited-200k-lines-of-spaghetti-code-what-now>

- “We are not getting the recognition for our programmers that they deserve. When you publish a paper you will thank that programmer or you aren't getting any more help from me. That programmer is going to be thanked by name; she's worked hard.”

<http://www.cs.virginia.edu/~robins/YouAndYourResearch.html>

http://videlectures.net/cancerbioinformatics2010_baggerly_irrh/

<http://www.pyvideo.org/video/2649/software-carpentry-lessons-learned>

Simple is hard

- The easier is the code to understand, the harder it was to write
- You want you code to be as simple as possible while still doing what it is supposed to do.

Common sense & general rules

- You cannot explain code or algorithm without explaining what core abstractions are:
 - *“Show me your flowchart and conceal your tables, and I shall continue to be mystified. Show me your tables, and I won't usually need your flowchart; it'll be obvious.”*
- Fred Brooks, “Mythical man-month”
- The more simple and well-documented the intermediate data structures are, the more maintainable and re-usable your code is
 - No custom classes
 - Dicts (JSON), lists (CSV tables), etc....
 - Numpy matrices, scipy sparse matrices, etc...

Common sense & general rules

- It is easier to explain how a single-transformation function works than one that has 20 secondary effects
- If you can't work by chaining functions and objects that do what you want, you are missing tools. Write them and re-use between projects.
 - Create the language adapted to your domain by creating abstract functions adapted to your domain (LISP => Scheme => S => R)
- “Never repeat yourself” = **NEVER** copy-paste code (even if you are going to edit a couple of lines and it is only a temporary fix)

Simple is hard

- The easier is the code to understand, the harder it is to write
- You want you code to be as simple as possible while still doing what it is supposed to do.
- You need abstractions to reason and communicate about your code.

Abstractions

- As in any other domain, a way to control the complexity is to use common abstractions:
 - Gene, promoter, protein, ...
 - Force, mass, acceleration,
 - Function, p_value, false discovery rate,
 - Field, Matrix, Group, ...
- Factory, Iterator, wrapper, ...

Core abstractions of the programming languages

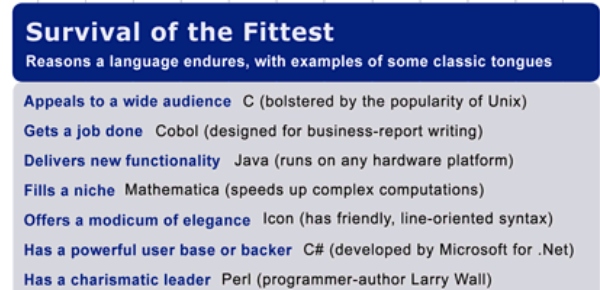
- **Assembler** = “typed commands; no more 0^s&1^s!”
- **Fortran** = “Let’s think in functions and procedures!”
- **LISP** = Lists & Lists transformation functions are all you’ll ever need
- **COBOL** = Let’s write programs as we write reports in the military!
- **Smalltalk** = Let’s do Objects, with Classes & Metaprogramming
- **C** = Pointers and Procedures
- **Shell** = Piping C together
- **Haskell** = Everything is a Function. Except for Monads. They are just monoids in the category of endofunctors
- **Python** = Everything is a dict & there is only one correct way of using it

Tracing the roots of computer languages through the ages

An ad hoc collection of engineers—electronic lexicographers, if you will—aim to save, or at least document the lingo of classic software. They're combing the globe's 9 million developers in search of coders still fluent in these nearly forgotten lingua frangas. Among the most endangered are Ada, APL, B (the predecessor of C), Lsp, Oberon, Smalltalk, and Simula.

Key

- 1954** Year Introduced
- Active:** thousands of users
- Protected:** taught at universities; compilers available
- Endangered:** usage dropping off
- Extinct:** no known active users or up-to-date compilers
- Lineage continues**



Sources: Paul Boutin; Brent Hailpern, associate director of computer science at IBM Research; The Retrocomputing Museum; Todd Proebsting, senior researcher at Microsoft; Gio Wiederhold, computer scientist, Stanford University

Abstractions on which Python is based

- All must fit into modules. Modules are there only to separate namespace and define contexts.
- An object's fitness for an application is determined from its signature, not declared class:
Error is raised when a method is not found/
- Readability for a human comes first, machine-readability is far behind
(Numpy/scipy+Cython shift the balance more towards executability)
- Everything is a dict, even if it looks like a class or a function

Python is a pragmatic language

- No commitment to any particular philosophy
 - Object Oriented Programming => Ok!
 - Function Oriented Programming => Ok!
 - Procedures => Ok!
- Easy to start with, but there is a depth in it.
 - Layer abstractions. Learn only for what you need right now
 - Use sensible defaults.
 - User must be able to get up and running your module in minutes, understand it in depth after a couple of hours and be able to directly access deeper internal functions.

Diving into Python internals

Time for Python itself!

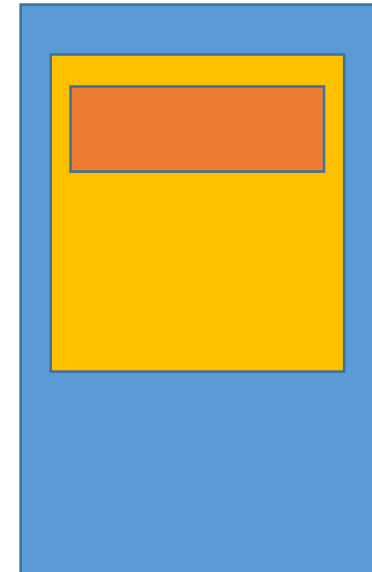
There is more to Python than just command execution

- In addition to executing your commands, Python stores a lot of information about the objects and functions it creates while executing them.
- Most of the content available this way can be found via the following commands:
 - `dir(object)`
 - `getattr(object, attrname)`
 - `Object.__doc__`

Exercise 0:

In Python, everything is a dict

- Python workspace is a giant dict.
 - When you define a variable, you add a key:value pair, where the key is the name of your variable and value is a pointer towards its location in memory
 - Depending on where you define the variable, this pair can be added to a different namespace.
- Two major dictionaries:
 - Global namespace (``global()``)
 - Local namespace (``local()``)
- Local dicts are nested one within other.
 - Function within a function has it's own scope
 - But also the scope of the outer function



Exercise 1:

If `__name__ == "__main__"`

- `__name__` is the first interesting Python internal
- when Python internals load a module to execute, they set `__name__` to `"__main__"`
- All the modules imported from `"__main__"` get `__name__` set to their respective name
- For immediate testing of the written code, `"if __name__ == '__main__':"` would execute the code in a module only when this module is executed as main, but not when it is imported.

Sidenote:

“from package.module import function”

- Attention:
 - Python will carry over with `__name__` the absolute path to the file you are using
 - If you use imports from outside `PYTHONPATH` even ``from package import module``, they will not work properly if your `“__main__”` moved around.

```
src
|-package_1
|  |-__init__.py
|  |-module_1_1
|  |-module_1_2
|-package_2
|  |-__init__.py
|  |module_2_1
|  |module_2_2
|-main_module
|-__init__.py
```

```
( main_module )
```

```
from package_2.module_2_1 import foo
```

```
( module_2_1 )
```

```
from package_1.module_1_1 import bar
```

```
Python main_module
```

```
Python module_2_1,
```



MAKE GIFS AT GIFSOU.COM

```
python -m package_2.module_2_1
```

Dict parsing (AST)

- The true backbone of Python is the AST (abstract syntax tree)
- A front-end of a python AST can be changed to a different language while keeping all the other advantages of Python
(<https://hy.readthedocs.org/>: a python-backed LISP environment)
- A back-end which managed Python-to-executable translation can be modified. (Cython, Jython, PyPy, IronPython)
- AST however is hard to comprehend for a human programmer and lot's of magic (metaprogramming) occurs before the code you type can be transformed into AST and executed.

Most magic relies on 'hidden' variables, like `__name__`

- `__builtins__`
 - gives access to all Python built-in functions
 - Automatically injected into most of the classes of your knowledge
 - `Type(object)` v.s. `object.__class__`
 - Support of more complicated things (such as injection of functions from C/C++)
- In general, `__xyz__`:
 - Gives access to python internals
 - Makes debugging hard and makes most static checks fail miserably
 - Not to be used without knowing what you are doing
 - Not to be confused with `_hidden_variables` (prone to change without notice)
- Before using `__xyz__`, check if you want to do can be done otherwise:

Exercise 2:

Problem #1 with Python: No Static typing

- You are always at risk of taking a “NameError”. After 10h after executing your code this is really painful.
 - Test your code before running
 - Break up long programs into shorter, testable functions, classes and modules



No static types, but duck types

- Some functions, even in build-in classes require specific types matched
- In this case a `TypeError` is raised from **within the function**
- Type error is not automatic, you have to raise it
- Otherwise ``attribute not found`` error will be raised

Exercise 3:

Catch/throw exceptions

```
>>> try:
...     raise Exception('spam', 'eggs')
... except Exception as inst:
...     print type(inst)      # the exception instance
...     print inst.args      # arguments stored in .args
```

```
>>> try:
...     raise KeyboardInterrupt
... finally:
...     print 'Goodbye, world!'
```

```
>>> try:
...     raise NameError('HiThere')
... except NameError:
...     print 'An exception flew by!'
...     raise
```

Exercise 4:

Mutability v.s immutability:

dict keys

- Dicts are hash tables:
 - Very efficient mapping from the key space to the space of results
 - In order to do this, keys are passed through a hash function
 - For the hash to always be the same, the key hash need to remain the same all the time
 - The function needs to be hashable, and possess equity for cash collision. For custum classes this is equivalent to implementing the following methods in your classes
 - ``__eq__(self, other)``
 - ``__hash__(self)``
 - It is better however to use named tuples from collections module or to set `__slots__` to [] or (). This prevents access to it and makes the class effectively immutable

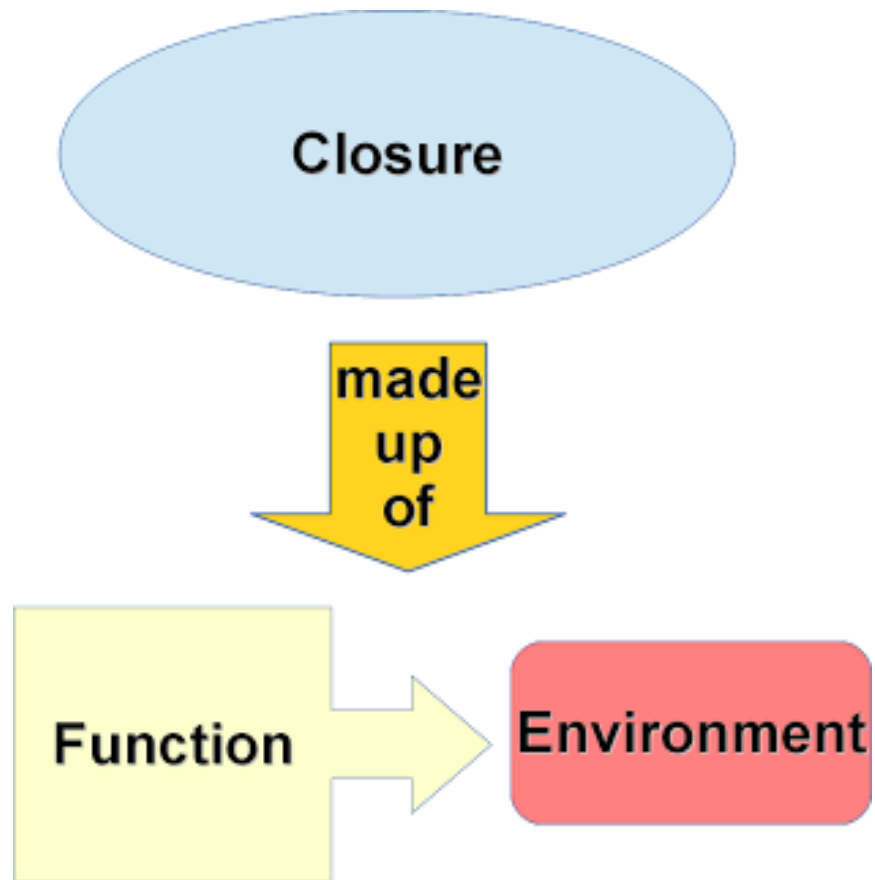
Exercise 5:

Mutability v.s immutability: scope transitions

- In Python, a lot of things are managed through dicts:
 - Local scope is a dict
 - Global scope is another dict
 - Variable calling is equivalent for a lookup inside those dicts in iteratively outlying scopes

Exercise 6:

Pythonic Closures



```
def first_closure(value = [0]):  
    print value[0]  
    value[0] += 1  
    return value[0]
```

Functions that poses default arguments store it within the dict that stores all their hidden attributes. If a default argument is mutable, it will always be the same between the calls to function. If it is modified, it will remain modified.

Exercise 7:

Wait, whaaaaaat?

```
def closure_generator(initial_memory):  
    def inner_closure(variable, memory=[initial_memory]):  
        """  
        Adds variavble to the memory. If types mismatch, cast everything to string  
        """  
        print memory  
        memory[0] += variable  
    return inner_closure
```

Functions are first-class variables

- In Python, functions are first-class variables and can be modified and manipulated on the fly:

```
def traveling_function():  
    print "Here I am!"  
  
function_dict = {  
    "func": traveling_function  
}  
  
trav_func = function_dict['func']  
trav_func()  
# >> Here I am!
```

Pythonic Wrappers & Decorators

- Wrappers in Python are functions that take functions as arguments and return other functions as arguments
 - They are very useful for logging, debugging, timing and caching
- A useful thing to know while writing wrappers
 - `def function(*args, **kwargs)`
 - `*args` unpacks a list of positional arguments and passes it to a function
 - `**kwargs` unpacks a list of keyword arguments and passes it to a function

`function(1, 2, 3, x=a, y=b)`

`=`

`function(*[1,2,3],**{'x':a, 'y':b})`

Python function do use closures

- Python functions have a reserved field
 - `f.__closure__`
 - This field contains bindings of the function's free names to the values of functions that are not in the global or local scope, but intermediate
 - They are however not easy to access on the fly
 - You can pre-generate the functions with bound variables in a way similar to wrappers to use directly this method

Exercise 8:

Pythonic Wrappers & Decorators



```
def a_function(arguments):
```

```
...
```

```
a_function = modifier(a_function)
```

=

```
@ modifier
```

```
def a_function(arguments):
```

```
....
```

Class modification on the fly

- Python classes are nothing but very special dicts
 - You shouldn't mess with them unless you need them
 - Using dicts instead of classes leaves out a lot of deep optimization, but is possible
- Dicts that support inner names mappings
 - `self` = reference to the object
 - `cls` = reference to the class of the object
 - `@staticmethod`
 - `@property`
 - `Getattr`, `setattr`, `hasattr`

Inheritance and multiple inheritance

- Inheritance is mainly injection of methods from one class to another
 - Pointers towards `__init__()` and `__new__()` methods are provided via the ``super`` interface
 - Multiple inheritance is injection of methods with non-conflicting names into the class
 - According to the order in which the base classes were provided
 - And how they inherit one from another
- Except when there is no specific reasonable way of resolving it.
- In which case it will raise an error

Exercise 9:

Meta Programming

- In Python, objects are not pre-existent. They are created by **Metaclasses**, usually by the **type** metaclass, that provides a mapping from your code into the actual AST and underlying C code implementing it.
- It is possible to manipulate metaclass behavior directly:

```
class MyMeta(type):  
    def __new__(cls, name, parents, dct):  
        print "A new class named " + name + " is going to be created"  
        return super(MyMeta, cls).__new__(cls, name, parents, dct)
```

```
class MyClass(object):  
    __metaclass__ = MyMeta
```

Metaprogramming



“Metaclasses are deeper magic than 99% of users should ever worry about. If you wonder whether you need them, you don’t (the people who actually need them know with certainty that they need them, and don’t need an explanation about why).”

– Tim Peters

This is very different in other languages



Exercise 10:

Advanced flow controls

Simplest flow controls

Repetition

- For
- While
- break
- pass

Selection

- If
- elif
- else

Iterators, list comprehensions and itertools

- Most frequent reasons to iterate something:
 - Go through the contents of something
 - Modify each element of the content (increment by 1)
 - Do something with all of the contents of a list (get a sum of it all)
 - Go through a combination of contents something
 - Get items with position -> Enumerate
 - Get items with corresponding position from other list -> zip

Exercise 11:

Iterators, list comprehensions and itertools

- Most frequent reasons to iterate something:
 - Go through the contents of something
 - Modify each element of the content (increment by 1)
 - Do something with all of the contents of a list (get a sum of it all)
 - Go through a combination of contents something
 - Get items with position -> Enumerate
 - Get items with corresponding position from other list -> zip
- Naïve loops are **slow, heavy** and **hard to maintain** in Python
 - Iterators – yield
 - List comprehensions – [], (), {}
 - Itertools module – combinations, product and cross-product
 - Generators as a flow structure on heavy data

Exercise 12:

Lambda functions

- Sometimes there is a need for simple, one-line functions, that need to be used in one location inside the code
- You might not want to declare them as the other functions

- Python has a special argument for it: “lambda”

```
g = lambda: x, y : x+y
```

```
def g(x, y):  
    return x+y
```

Exercise 13:

Functional Python

- Despite Guido van Rossum being explicitly against functional programming (too dogmatic and not practical enough), Python allows basic functional programming and higher-order functions
- Standard library: ``reduce((lambda x, y: y(x)), [layer_1, layer_2, layer_3], tst_lst)`` is a higher order function
- ``functools`` standard library module provides tools for writing high-order functions more easily

Exercise 14:

Collections module

- Recent high-performance data containers for specific applications:
 - Tuple with names (`namedtuple`) => immutable dict
 - Fast queues and heaps (`deque`)
 - Quick counter of object occurrence in list (`Counter`)
 - Dict that remembers order in which key-value pairs were added (`OrderedDict`)
 - Dict that has pre-set values to prevent undefined key errors (`defaultdict`)
- Whenever possible, they should be used instead of manual naïve implementations.

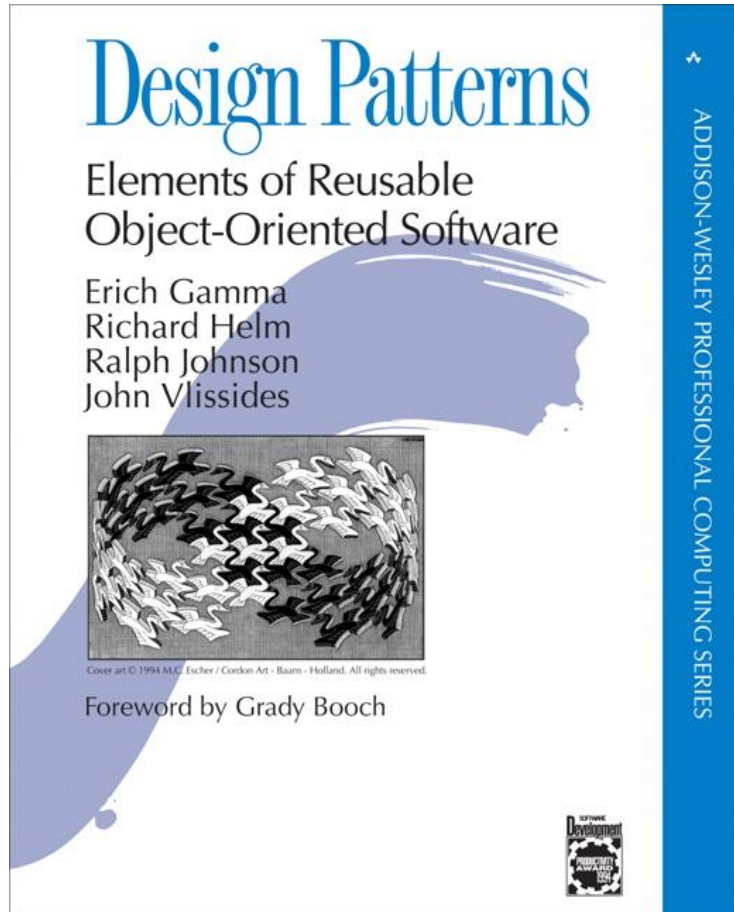
Exercise 15:

Basic parallelism in Python

- A vast majority of problems are embarrassingly parallel
 - For each element, do an operation
- Unfortunately, Python is not naively parallel and allows for easy routine-oriented programming, leading to frequent secondary efforts
- Rendering it parallel requires to re-implement execution pipelines as functional chain, operating within
 - map-reduce scheme
 - actor model scheme (Erlang tutorials tell a lot about it)

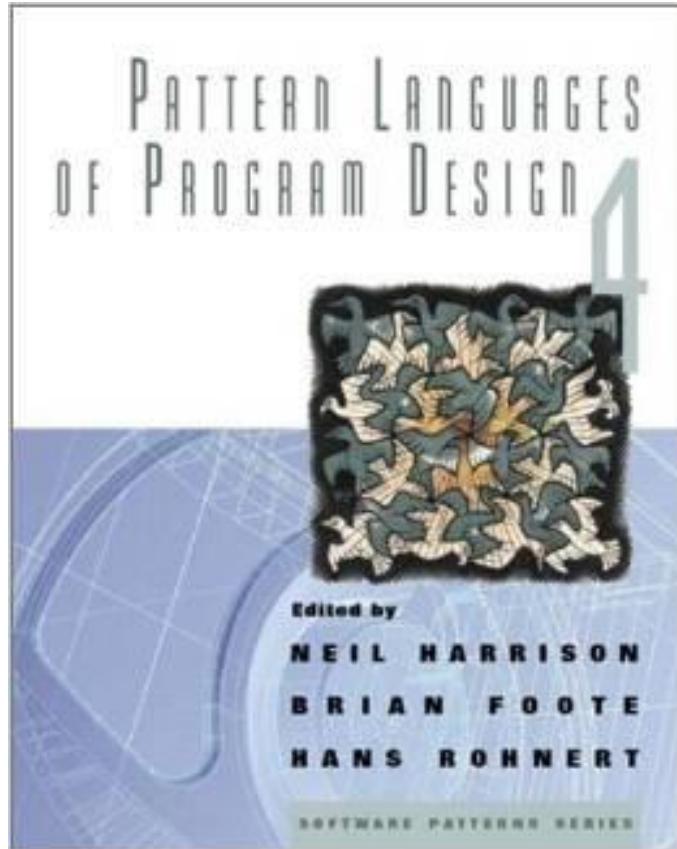
Exercise 16:

Programming patterns



- Introduced in the context of object-oriented programming by the gang of 4 in 1993
- Used extensively in the context of Smalltalk
- Credited with dismissal of Smalltalk
- Strongly criticized by modern-day programmers
- Is still frequently encountered in OO code & communities

Programming patterns



- More and more useful patterns exist, even if there is no repertory for them
- In general, a working module is worth much more than a pattern system
- Except for those maintaining it

Design Patterns + a couple of additional ones

- | | | | | |
|----------------|-----|-----|--------------------------|-----|
| • Iterator | (P) | (D) | • Façade | (D) |
| • Map-reduce | | | • Observer | (D) |
| • Memoization | | (D) | • Adapters | (D) |
| • Singleton | (P) | (D) | • Protocols | (D) |
| • Shared State | | (D) | • Reference counting (P) | (D) |
| • Wrapper | (P) | (D) | • Agent programming | (D) |
| • Closure | | | • Flyweight | (D) |
| • Composition | (P) | (D) | | |

(P): a core abstraction in Python

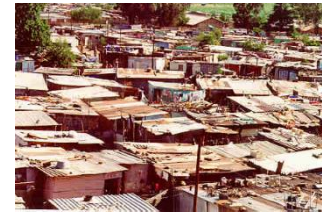
(D): members of traditional pattern lists

Refactoring

And other architectural fun

Complexity control & (re-)factoring a system

- You need to deliver quality software on time, and under budget.
- Therefore, focus first on features and functionality, then focus on architecture and performance.
- You need an immediate fix for a small problem, or a quick prototype or proof of concept.
- Therefore, produce, by any means available, simple, expedient, disposable code that adequately addresses just the problem at-hand.



Complexity control & (re-)factoring a system

- Master plans are often rigid, misguided and out of date. Users' needs change with time.
- Therefore, incrementally address forces that encourage change and growth. Allow opportunities for growth to be exploited locally, as they occur. Refactor unrelentingly.

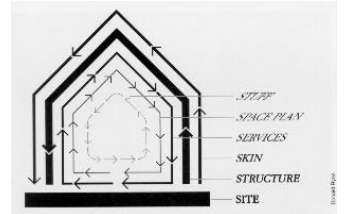


- Maintenance needs have accumulated, but an overhaul is unwise, since you might break the system.
- Therefore, do what it takes to maintain the software and keep it going. Keep it working.



Complexity control & (re-)factoring a system

- Different artifacts change at different rates.
- Therefore, factor your system so that artifacts that change at similar rates are together.



- Overgrown, tangled, haphazard spaghetti code is hard to comprehend, repair, or extend, and tends to grow even worse if it is not somehow brought under control.
- Therefore, if you can't easily make a mess go away, at least cordon it off. This restricts the disorder to a fixed area, keeps it out of sight, and can set the stage for additional refactoring.



Complexity control & (re-)factoring a system

- Your code has declined to the point where it is beyond repair, or even comprehension.
- Therefore, throw it away and start over.



<http://www.laputan.org/mud/>

In case you already have a Python library and need to refactoring it

- <http://clonedigger.sourceforge.net/index.html>

Source file "/home/peter/clone_digger/biopython-1.44/Bio/SwissProt/SProt.py" The first line is 224	Source file "/home/peter/clone_digger/biopython-1.44/Bio/PubMed.py" The first line is 39
def __init__(self, delay=5.0, parser=None): self.parser = parser self.limiter = RequestLimiter(delay)	def __init__(self, delay=5.0, parser=None): self.parser = parser self.limiter = RequestLimiter(delay)
def __len__(self): raise NotImplementedError, "SwissProt contains lots of entries"	def __len__(self): raise NotImplementedError, "PubMed contains lots of entries"
def clear(self): raise NotImplementedError, "This is a read-only dictionary"	def clear(self): raise NotImplementedError, "This is a read-only dictionary"
def __setitem__(self, key, item): raise NotImplementedError, "This is a read-only dictionary"	def __setitem__(self, key, item): raise NotImplementedError, "This is a read-only dictionary"
def update(self): raise NotImplementedError, "This is a read-only dictionary"	def update(self): raise NotImplementedError, "This is a read-only dictionary"
def copy(self): raise NotImplementedError, "You don't need to do this..."	def copy(self): raise NotImplementedError, "You don't need to do this..."
def keys(self): raise NotImplementedError, "You don't really want to do this..."	def keys(self): raise NotImplementedError, "You don't really want to do this..."
def items(self): raise NotImplementedError, "You don't really want to do this..."	def items(self): raise NotImplementedError, "You don't really want to do this..."
def values(self): raise NotImplementedError, "You don't really want to do this..."	def values(self): raise NotImplementedError, "You don't really want to do this..."
def has_key(self, id): try: self[id] except KeyError: return 0 return 1	def has_key(self, id): try: self[id] except KeyError: return 0 return 1
def get(self, id, failobj=None): try: return self[id] except KeyError: return failobj raise "How did I get here?"	def get(self, id, failobj=None): try: return self[id] except KeyError: return failobj raise "How did I get here?"

Core abstractions of programming paradigms

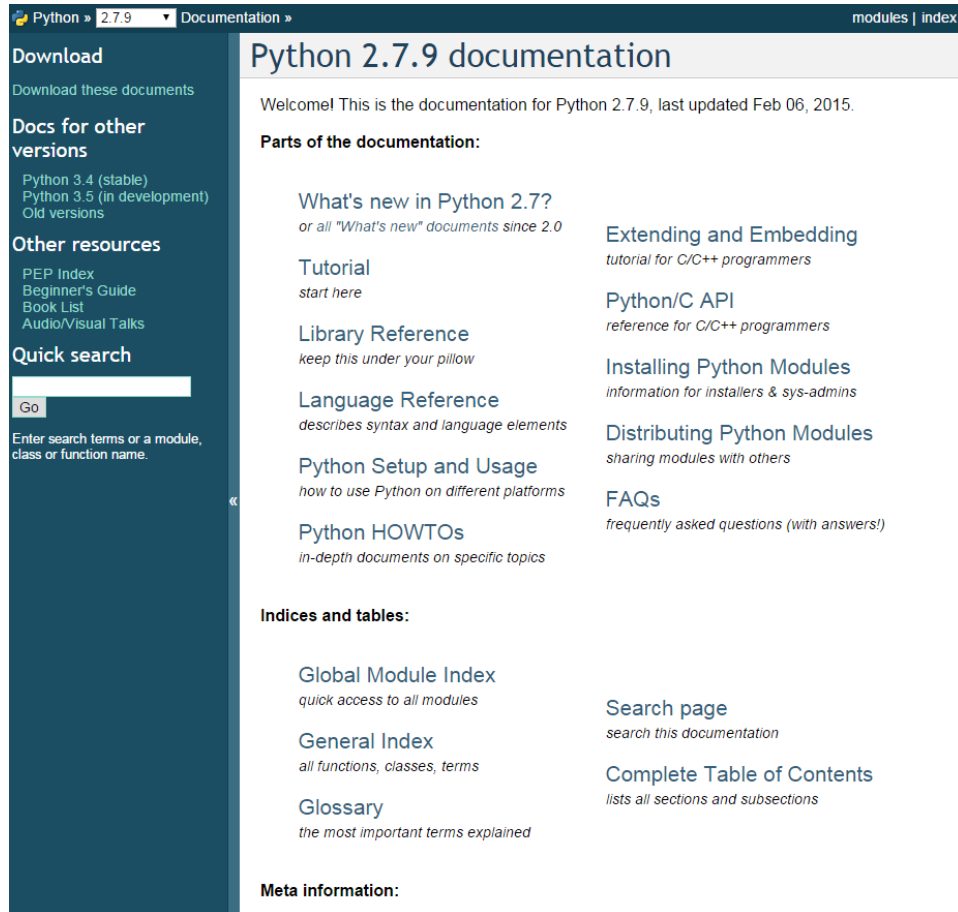
- Procedural
- Object Oriented
- Functional
- Aspect Oriented
- Logic programming
- Symbolic programming

Typically, an aspect is *scattered* or *tangled* as code, making it harder to understand and maintain. It is scattered by virtue of the function (such as logging) being spread over a number of unrelated functions that might use *its* function, possibly in entirely unrelated systems, different source languages, etc. That means to change logging can require modifying all affected modules. Aspects become tangled not only with the mainline function of the systems in which they are expressed but also with each other. That means changing one concern entails understanding all the tangled concerns or having some means by which the effect of changes can be inferred.

Documenting

There is one right way of doing it

Sphinx, API-doc and Read the Docs



The screenshot shows the Python 2.7.9 documentation page. The left sidebar contains links for downloading documents, documentation for other versions (Python 3.4, 3.5, and old versions), other resources (PEP Index, Beginner's Guide, Book List, Audio/Visual Talks), a quick search bar, and indices and tables (Global Module Index, General Index, Glossary). The main content area is titled 'Python 2.7.9 documentation' and includes a welcome message, a list of parts of the documentation (What's new in Python 2.7?, Tutorial, Library Reference, Language Reference, Python Setup and Usage, Python HOWTOs, Extending and Embedding, Python/C API, Installing Python Modules, Distributing Python Modules, FAQs), and meta-information (Search page, Complete Table of Contents).

`abc.abstractmethod(function)`

A decorator indicating abstract methods.

Using this decorator requires that the class's metaclass is `ABCMeta` or is derived from it. A class that has a metaclass derived from `ABCMeta` cannot be instantiated unless all of its abstract methods and properties are overridden. The abstract methods can be called using any of the normal 'super' call mechanisms.

Dynamically adding abstract methods to a class, or attempting to modify the abstraction status of a method or class once it is created, are not supported. The `abstractmethod()` only affects subclasses derived using regular inheritance; "virtual subclasses" registered with the ABC's `register()` method are not affected.

Usage:

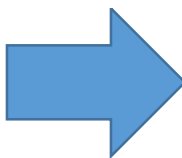
```
class C:
    __metaclass__ = ABCMeta
    @abstractmethod
    def my_abstract_method(self, ...):
        ...
```

Note: Unlike Java abstract methods, these abstract methods may have an implementation. This implementation can be called via the `super()` mechanism from the class that overrides it. This could be useful as an end-point for a super-call in a framework that uses cooperative multiple-inheritance.

Sphinx, API-doc and Read the Docs

```
53 class abstractstaticmethod(staticmethod):
54     """
55     A decorator indicating abstract staticmethods.
56
57     Similar to abstractmethod.
58
59     Usage:
60
61         class C(metaclass=ABCMeta):
62             @abstractstaticmethod
63             def my_abstract_staticmethod(...):
64                 ...
65
66     'abstractstaticmethod' is deprecated. Use 'staticmethod' with
67     'abstractmethod' instead.
68     """
69
70     __isabstractmethod__ = True
71
72     def __init__(self, callable):
73         callable.__isabstractmethod__ = True
74         super().__init__(callable)
```

[sphinx.ext.autodoc](#)



```
.. decorator:: abstractstaticmethod
```

A subclass of the built-in :func:`staticmethod`, indicating an abstract staticmethod. Otherwise it is similar to :func:`abstractmethod`.

This special case is deprecated, as the :func:`staticmethod` decorator is now correctly identified as abstract when applied to an abstract method::

```
class C(metaclass=ABCMeta):
    @staticmethod
    @abstractmethod
    def my_abstract_staticmethod(...):
        ...
```

```
.. versionadded:: 3.2
```

```
.. deprecated:: 3.3
```

It is now possible to use :class:`staticmethod` with :func:`abstractmethod`, making this decorator redundant.

Sphinx, API-doc and Read the Docs

- To get Started:
 - <http://raxcloud.blogspot.com/2013/02/documenting-python-code-using-sphinx.html>
 - https://www.youtube.com/watch?feature=player_embedded&v=oJsUvBQyHBs

*Args and **kwargs arguments

Sharing your code with others

Or how to contribute to pip

While you are still developing

- Git - Github:
 - The learning curve is a little bit steep
 - However it is very hard to get things wrong, even if you are a bare beginner
 - Github is the default go-to repository of the code
 - Easy to follow/edit projects you want to work or want to use
 - Easy to contribute back to other people's projects
 - Easy to follow up on issues, such as bugs that arise in your code
 - Suggested moves:
 - Set-up github account
 - Share a project you are working on github so that you can work on it from several locations at the same time

Git Cheat Sheet

<http://git.or.cz/>

Remember: `git command --help`

Global Git configuration is stored in `$HOME/.gitconfig` (`git config --help`)

Create

From existing data

```
cd ~/projects/myproject
git init
git add .
```

From existing repo

```
git clone ~/existing/repo ~/new/repo
git clone git://host.org/project.git
git clone ssh://you@host.org/proj.git
```

Show

Files changed in working directory

```
git status
```

Changes to tracked files

```
git diff
```

What changed between \$ID1 and \$ID2

```
git diff $id1 $id2
```

History of changes

```
git log
```

History of changes for file with diffs

```
git log -p $file $dir/ec/tory/
```

Who changed what and when in a file

```
git blame $file
```

A commit identified by \$ID

```
git show $id
```

A specific file from a specific \$ID

```
git show $id:$file
```

All local branches

```
git branch
```

(star "*" marks the current branch)

Concepts

Git Basics

```
master : default development branch
origin  : default upstream repository
HEAD    : current branch
HEAD^   : parent of HEAD
HEAD~4  : the great-great grandparent of HEAD
```

Revert

Return to the last committed state

```
git reset --hard
```

⚠ you cannot undo a hard reset

Revert the last commit

```
git revert HEAD
```

Creates a new commit

Revert specific commit

```
git revert $id
```

Creates a new commit

Fix the last commit

```
git commit -a --amend
```

(after editing the broken files)

Checkout the \$id version of a file

```
git checkout $id $file
```

Branch

Switch to the \$id branch

```
git checkout $id
```

Merge branch1 into branch2

```
git checkout $branch2
git merge branch1
```

Create branch named \$branch based on the HEAD

```
git branch $branch
```

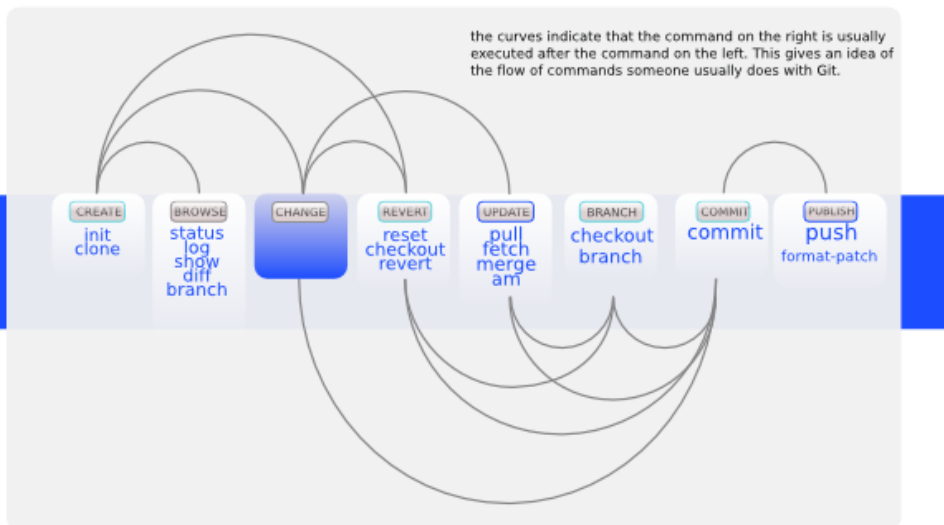
Create branch \$new_branch based on branch \$other and switch to it

```
git checkout -b $new_branch $other
```

Delete branch \$branch

```
git branch -d $branch
```

Commands Sequence



Update

Fetch latest changes from origin

```
git fetch
```

(but this does not merge them).

Pull latest changes from origin

```
git pull
```

(does a fetch followed by a merge)

Apply a patch that some sent you

```
git am -3 patch.mbox
```

(in case of a conflict, resolve and use `git am --resolved`)

Publish

Commit all your local changes

```
git commit -a
```

Prepare a patch for other developers

```
git format-patch origin
```

Push changes to origin

```
git push
```

Mark a version / milestone

```
git tag v1.0
```

Useful Commands

Finding regressions

```
git bisect start (to start)
git bisect good $id ($id is the last working version)
git bisect bad $id ($id is a broken version)
```

```
git bisect bad/good (to mark it as bad or good)
git bisect visualize (to launch gitk and mark it)
git bisect reset (once you're done)
```

Check for errors and cleanup repository

```
git fsck
git gc --prune
```

Search working directory for foo()

```
git grep "foo()"
```

Resolve Merge Conflicts

To view the merge conflicts

```
git diff (complete conflict diff)
git diff --base $file (against base file)
git diff --ours $file (against your changes)
git diff --theirs $file (against other changes)
```

To discard conflicting patch

```
git reset --hard
git rebase --skip
```

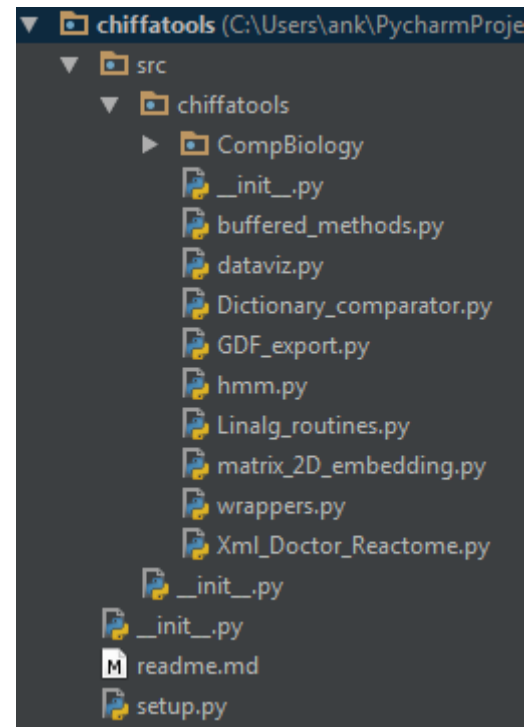
After resolving conflicts, merge with

```
git add $conflicting_file (do for all resolved files)
git rebase --continue
```

Cheat Sheet Notation

\$id : notation used in this sheet to represent either a commit id, branch or a tag name
\$file : arbitrary file name
\$branch : arbitrary branch name

1) Package your source code



2.1) Write a setup.py – import all the stuff

```
from setuptools import setup, find_packages # Always prefer setuptools over distutils
from codecs import open # To use a consistent encoding
from os import path

here = path.abspath(path.dirname(__file__))

# Get the long description from the relevant file
with open(path.join(here, 'readme.md'), encoding='utf-8') as f:
    long_description = f.read()
```

2.2) Write a setup.py – name & version

```
setup(  
    name='chiffatools',  
  
    # Versions should comply with PEP440. For a discussion on single-sourcing  
    # the version across setup.py and the project code, see  
    # http://packaging.python.org/en/latest/tutorial.html#version  
    version='0.0-alpha.14',  
  
    description='A collection of useful but shorts tools for computational biologists and other data scientists',  
    long_description=long_description,  
  
    # The project's main homepage.  
    url='https://github.com/ciffa/chiffatools',  
  
    # Author details  
    author='Andrei Kucharavy',  
    author_email='andrei.chiffa136@gmail.com',  
  
    # Choose your license  
    license='BSD',
```


2.3) Write a setup.py – classify and index

```
# See https://pypi.python.org/pypi?%3Aaction=list_classifiers
classifiers=[
    # How mature is this project? Common values are
    # 3 - Alpha
    # 4 - Beta
    # 5 - Production/Stable
    'Development Status :: 3 - Alpha',

    # Indicate who your project is intended for
    'Intended Audience :: Computational biologists, General scientific computing users',
    'Topic :: Scientific computing',

    # Pick your license as you wish (should match "license" above)
    'License :: OSI Approved :: BSD 3-clause license',

    # Specify the Python versions you support here. In particular, ensure
    # that you indicate whether you support Python 2, Python 3 or both.
    'Programming Language :: Python :: 2.7',
],

# What does your project relate to?
keywords='improved visualization, linear algebra, GDF exporting, Gene Ontology, Uniprot, HMM',
```

2.4) Write a setup.py – show where your package contents are and what they require

```
# You can just specify the packages manually here if your project is
# simple. Or you can use find_packages().
packages=['chiffatools']+find_packages(exclude=['contrib', 'docs', 'tests*']),
package_dir = {'chiffatools':'src/chiffatools'},
# List run-time dependencies here. These will be installed by pip when your
# project is installed. For an analysis of "install_requires" vs pip's
# requirements files see:
# https://packaging.python.org/en/latest/technical.html#install-requires-vs-requirements-files
install_requires=[ 'numpy',
                   'scipy',
                   'matplotlib',
                   'scikit-learn',
                   'python-Levenshtein',],

# List additional groups of dependencies here (e.g. development dependencies).
# You can install these using the following syntax, for example:
# $ pip install -e .[dev,test]
extras_require = {
    'dev': [],
    'test': [],
},
```

2.5) Write a setup.py – provide entry point (usually empty, but required)

```
# To provide executable scripts, use entry points in preference to the
# "scripts" keyword. Entry points provide cross-platform support and allow
# pip to create the appropriate form of executable for the target platform.
entry_points = """
"""
```

3) Upload to github and install with pip

- > pip install -U <https://github.com/chiffa/chiffatools.git>
- Do not forget that after you update the version, you need to bump the version for pip to update it!

4) Add to Pypi (optional)

> python setup.py sdist

> python setup.py register

> python setup.py sdist upload

Don't forget the docs

Adding packages to PyPi works best after you wrote the documentation, generated the readthedocs with it and pushed to readthedocs.

Now, time for hardcore scientific
python

Yay!

Second parameter in program efficiency: how much memory it takes to run?

- CPU load (time to execute) v.s. Memory load
- Most of the time the “freezes” are caused by infinite loops or a process running out of RAM

Latency Numbers Every Programmer Should Know

■ 1 ns

■ L1 cache reference: 0.5 ns

■ Branch mispredict: 5 ns

■ L2 cache reference: 7 ns

■ Mutex lock/unlock: 25 ns

■ = 100 ns

■ Main memory reference: 100 ns

■ = 1 μ s

■ Compress 1 KB with Zippy: 3 μ s

■ = 10 μ s

■ Send 1 KB over 1 Gbps network: 10 μ s

■ SSD random read (1Gb/s SSD): 150 μ s

■ Read 1 MB sequentially from memory: 250 μ s

■ Round trip in same datacenter: 500 μ s

■ = 1 ms

■ Read 1 MB sequentially from SSD: 1 ms

■ Disk seek: 10 ms

■ Read 1 MB sequentially from disk: 20 ms

■ Packet roundtrip CA to Netherlands: 150 ms

Source: <https://gist.github.com/2841832>

Exercise 17:

Numpy array v.s. matrix

- Numpy = np
- np.array != np.matrix
 - They work very differently
 - They interpret identical operations differently
- For the sake of compatibility, everyone uses np.array



Array indexing routines

- Indexing starts at 0, ends and N-1, like in the rest of Python:
 - `range(0, 4) => [0, 1, 2, 3]`
 - But `np.linspace(0, 4, 5) => [0, 1, 2, 3, 4]`
- Indexing with negatives starts the counting from the end of array
 - `np.linspace(0, 4, 5)[-1] => 4`
- `:` denotes an indexing range
 - `np.linspace(0, 4, 5)[1 : -1] => [1, 2, 3]`
- `::N` denotes stepping, picking each Nth element:
 - `np.linspace(0, 4, 5)[::2] => [0, 2, 4]`
- All indexing routines are expandable with the respect to dimensions

Inlined operations and boolean indexing

- Operations on arrays are element-wise
 - `Lines*columns`
 - `Lines *= 2`
 - `Lines /= 2`
- Matrix operations are called through specific functions
 - `np.dot(lines,columns)`
 - `np.linalg.eigh(np.dot(lines,columns))`
- Indexing and array logic is straightforward
 - `lines>1`
 - `Lines[lines>1]`
 - Array indexing with Booleans leads to a flattened array
 - This flattened array can be used to set values in an other array, where the values to set were selected in a Boolean way
 - `new_arr[lines>1] = lines[lines>1]`

Arrays are C objects

- As C objects, they get modified by reference (pointers). They need to be explicitly copied to become independent:
 - `A = np.zeros(2,2)`
 - `B = A` (correct way: `B = A.copy()`)
 - `B[0, 0] = 1`
 - `print A`
- As C objects, they also have an explicit class that can be accessed via `dtype()` and modified via `astype()`:
 - `A.dtype`
 - `A.astype(np.string)`

Array shape modification

- `Np.pad`:
 - `np.pad(A, ((1, 2), (2, 3)), 'edge')`
- `Np.reshape`
 - `np.reshape(arr, (2, 4, 4))`
- `Np.newaxis`
 - `A[:, :, np.newaxis]`
- `Np.repeat`
 - `np.repeat(A, 2, axis=1)`
- `Np.concatenate`
 - `np.concatenate((A, B), Axis=1)`
- `Np.rollaxis`
 - `Np.rollaxis(arr, 2)`

Numpy array creation routine

- `Np.rand`
- `Np.zeros`
- `Np.ones`
- `Np.linspace+reshape`
- `Np.mesh`
- `Np.diag`

Numpy array indexing conventions

- Numpy is the bedrock of linear algebra in Python
- Several other libraries optimized for different task exists that respect same indexing conventions as numpy arrays but do very different things beneath the surface.
 - `scipy.sparse.linalg`
 - `sckits-cuda`

Exercise 18:

Exercise 19:

Function fitting in Python

- `scipy.stats`
- `scipy.optimize.curve_fit` (wrapper around `scipy.optimize.minimize`)
 - Standard fitting with no error on the x axis and error on the y axis
- `scipy.odr`
 - Error on both x and y axes

Exercise 20:

Matplotlib pyplot basics

- `Pyplot.plot`:
 - Plots a line or a set of points
- `Pyplot.show`:
 - Shows whatever has been previously plotted
- `Pyplot.imshow`:
 - Shows a 2D matrix as an image
- `Pyplot.colorbar`:
 - Allows to show to what value a color corresponds in a `plt.imshow` image
- `Pyplot.subplot(lines, columns, currently_filled)`
- `Pyplot.title()`, `.set_xlabel()`, `.set_ylabel()`

Exercise 21:

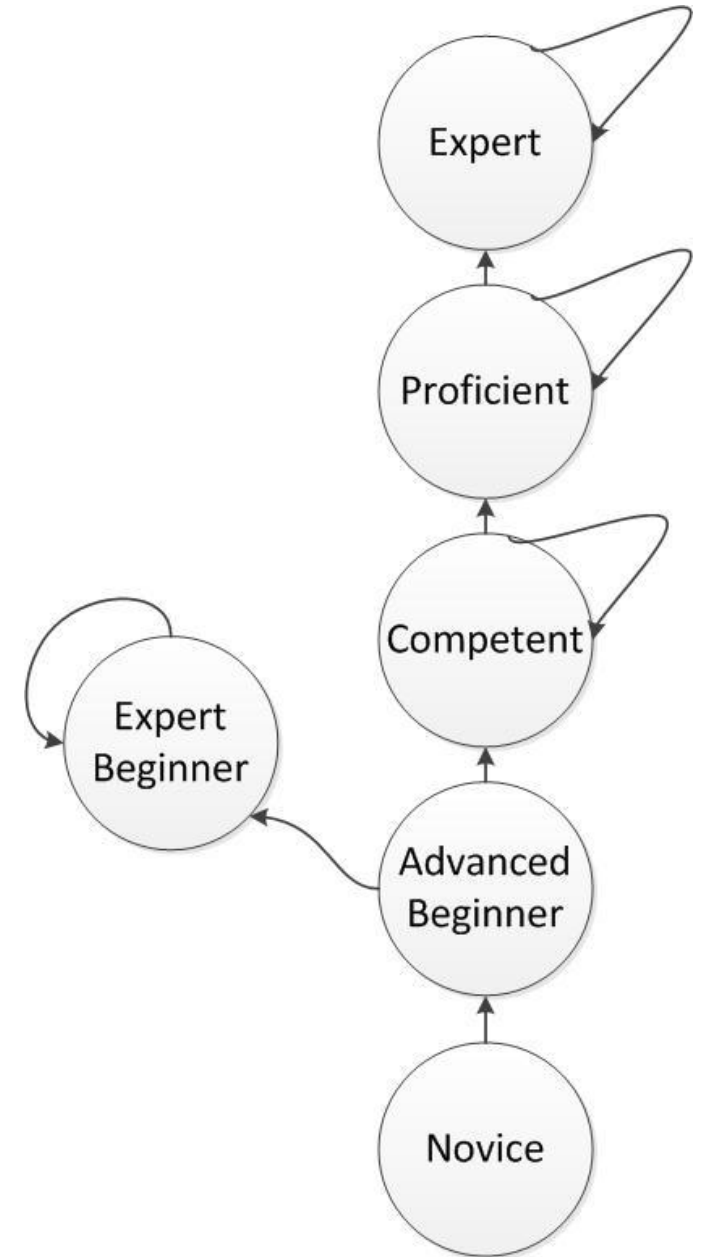
Pandas and Seaborn

- For the R addicts
 - Pandas ~ R dataframe
 - Seaborn ~ ggplot

Instead of a conclusion

<http://www.daedtech.com/how-developers-stop-learning-rise-of-the-expert-beginner>

Thanks to Dar Dahlen and Malcolm Cook for the advice and input



If you could not follow, my bad.

You can find similar concepts from this presentation presented much more professionally here:

<https://github.com/jrjohansson/scientific-python-lectures>

<http://intermediate-and-advanced-software-carpentry.readthedocs.org/en/latest/>

Going further

- <http://www.pyvideo.org/video/2623/python-epiphanies-1>
- <http://www.pyvideo.org/video/2647/designing-poetic-apis>
- <http://www.pyvideo.org/video/2571/all-your-ducks-in-a-row-data-structures-in-the-s>
- <http://www.pyvideo.org/video/2627/fast-python-slow-python>
- <http://www.pyvideo.org/video/2648/unit-testing-makes-your-code-better>
- <http://programmers.stackexchange.com/questions/155488/ive-inherited-200k-lines-of-spaghetti-code-what-now>
- <https://google-styleguide.googlecode.com/svn/trunk/pyguide.html>

- <https://github.com/jrjohansson/scientific-python-lectures>
- <http://intermediate-and-advanced-software-carpentry.readthedocs.org/en/latest/>

- <http://www.pyvideo.org/video/2674/getting-started-testing>
- <http://www.pyvideo.org/video/2577/ipython-in-depth-high-productivity-interactive-a-1>
- <http://www.pyvideo.org/video/2574/decorators-a-powerful-weapon-in-your-python>
- <http://youtu.be/AmMaN1AokTI>
- <http://pyvideo.org/video/2616/getting-hy-on-python-how-to-implement-a-lisp-fro>

In case you are fighting with installing modules on Windows

- <http://www.lfd.uci.edu/~gohlke/pythonlibs/>