python-fundamentals

October 17, 2017

1 Reference

- Python Standard Library modules
- `builtin` functions and classes
- Idiomatic Python
- Python Style Guides:
  - PEP 8
  - Google Python Style Guide
- Magic methods:
  - official specs
  - great summary
- Scipy Lectures
- Numpy Tutorial
  - Numpy for Matlab Users
- Matplotlib Plotting Tutorial
  - gallery

2 Numbers, Strings, and Math

```
In [ ]: # basic math
        3 + 7
```

```
In [ ]: 2+4
```

```
In [ ]: # scientific notation
        2 * 6.78E3
```

```
In [ ]: import math
        i = math.pi
        i
```

```
In [ ]: math.e
```

```
In [ ]: 7E3
In [ ]: 4E-4
```

```
In [ ]: # complex numbers
        (3+2j) * (3-2j)
```

```
In [ ]: val = (3+2j) * (3-2j)
```

```
In [ ]: val
```

```
In [ ]: type(val)
```

```
In [ ]: # exponentiation
        2**11
```

```
In [ ]: 2**32
```

```
In [ ]: math.e**((1j)*math.pi)
```

```
In [ ]: # strings
        "This is a very short string"
```

```
In [ ]: 'This string uses single quotes'
```

```
In [ ]: '\nThis is a multi-line string\nwhich can be delimited by \neither triple single quotes
```

```
In [ ]: # long strings
        '''
        This is a multi-line string
        which can be delimited by
        either triple single quotes (')
        or triple double quotes (")
        '''
```

Add number and a string

```
In [ ]: # experiment!
        4 + '6'
```

```
In [ ]: # but what about adding strings?
        'Hello ' + ' there!'
```

```
In [ ]: 'Hello '.__add__("there!")
```

```
In [ ]: # or multiplying strings?
        'my ' * 2
```

3 References (aka *variables*)

```
In [ ]: # expressions create objects which last as long as there is a reference to them
        a = 42
```

```
In [ ]: a
```

```
In [ ]: a / 2
```

```
In [ ]: a + 10
```

```
In [ ]: b = a
```

```
In [ ]: b
```

```
In [ ]: type(a)
```

```
In [ ]: type(b)
```

```
In [ ]: # objects have id-s as well. You cannot do much with them, though
        id(a)
```

```
In [ ]: id(b)
```

```
In [ ]: c = 555
```

```
In [ ]: d = c
```

```
In [ ]: e = 555
```

```
In [ ]: id(c)
```

```
In [ ]: id(d)
```

```
In [ ]: id(e)
```

```
In [ ]: c == d
```

```
In [ ]: c == e
```

```
In [ ]: c is d
```

```
In [ ]: c is e
```

```
In [ ]: x = 128
        y = x
        z = 128
```

```
In [ ]: x == y
```

```
In [ ]: x == z
In [ ]: x is y
```

```
In [ ]: x is z
```

```
In [ ]: id(x)
```

```
In [ ]: id(y)
```

```
In [ ]: id(z)
```

```
In [ ]: x
```

```
In [ ]: y
```

```
In [ ]: y = y + 10
```

```
In [ ]: y
```

```
In [ ]: id(y)
```

```
In [ ]: x
```

```
In [ ]: id(x)
```

- these are typically called "variables", but a better name in Python is "reference"
- a "reference" refers to an object
- objects are Autonomous (no scope, heap allocated) and Anonymous (no name)
- only references have a scope
- references can be re-assigned to a new object at any time
- allowed to have multiple references to the same object

```
In [ ]: # all objects have an ID, but don't read too much into this: it is a unique number assig
```

```
In [ ]: # can delete references, but this doesn't delete the object
        a = 43
```

```
In [ ]: b = a
```

```
In [ ]: del a
```

```
In [ ]: a
```

```
In [ ]: b
```

4 Lists

- lists preserve order
- have no constraint on duplicate entries
- can be changed, reordered, added to, or removed from
- IMPORTANT: really just contain a collection of "unnamed" references, not objects

```
In [ ]: nums = [3, 7 , 6, 3, 0, 3, 4, 6, 5,5]
```

```
In [ ]: nums
```

```
In [ ]: # function calls! type names!
        type(nums)
```

```
In [ ]: len(nums)
```

```
In [ ]: nums.__len__()
```

```
In [ ]: # referencing FROM ZERO
        nums[0]
```

```
In [ ]: nums.__getitem__(0)
```

```
In [ ]: nums[-1]
```

```
In [ ]: colors = 'red green blue yellow white black pink brown'.split()
```

```
In [ ]: colors
```

```
In [ ]: colors[0]
```

```
In [ ]: type(colors)
```

```
In [ ]: # negative indexing ("back from end")
```

```
In [ ]: colors[-1]
```

```
In [ ]: len(colors)
```

```
In [ ]: colors[len(colors)-1]
        # colors[len(colors)]
```

```
In [ ]: colors[7]
```

```
In [ ]: colors[len(colors) - 1]
```

```
In [ ]: colors[-1]
```

```
In [ ]: # selecting a subset of objects from a list (NOTE: last index is not included)
```

```
In [ ]: colors
In [ ]: colors[2:4]

In [ ]: nums[2:7]

In [ ]: nums

In [ ]: colors

In [ ]: colors[:3]

In [ ]: colors[:-1]

In [ ]: colors[-4:-1]

In [ ]: colors[3:]

In [ ]: colors[3:7]

In [ ]: colors[-4:]

In [ ]: id(colors)

In [ ]: dupe = colors

In [ ]: id(dupe)

In [ ]: copy = colors
        id(colors)

In [ ]: id(copy)

In [ ]: dupe

In [ ]: copy = colors[:] # slice from beginning to end

In [ ]: copy

In [ ]: id(colors)

In [ ]: id(copy)

In [ ]: colors[3]

In [ ]: colors[3] = 'mauve'

In [ ]: colors

In [ ]: dupe

In [ ]: copy

In [ ]: colors[1]
In [ ]: id(colors[1])
```

```
In [ ]: id(dupe[1])
```

```
In [ ]: id(copy[1])
```

```
In [ ]: copy.pop()
```

```
In [ ]: copy.pop()
```

```
In [ ]: copy
```

```
In [ ]: copy.append('purple')
```

```
In [ ]: copy
```

```
In [ ]: colors
```

```
In [ ]: dupe.sort()
```

```
In [ ]: dupe
```

```
In [ ]: colors
```

```
In [ ]: copy
```

5 Looping

```
In [ ]: nums
```

```
In [ ]: # indentation, scoping, print() function
        for c in colors:
            print 'color is', c # Python 3: print('number is', x)
```

```
In [ ]: # enumerate() is a built-in function of Python.
        # It allows us to loop over something and have an automatic counter.

        print 'START'

        print nums

        for index, x in enumerate(nums):

            y = 10*x + 3
            z = 4*x**2 + 7*x -5
            print index, (x, y, z)

        print 'END'
```

```
In [ ]: #nums
        # type(enumerate(nums))
        type(nums)
        print nums

In [ ]: person = ('John', 41, 'Smith')

In [ ]: type(person)

In [ ]: person

In [ ]: person[0]

In [ ]: name, age, surname = person # tuple unpacking

In [ ]: name

In [ ]: age

In [ ]: surname
```

6 Functions

```
In [ ]: # hello
        def hello():
            print 'Hello John'

In [ ]: hello

In [ ]: hello()

In [ ]: hi = hello

In [ ]: hi

In [ ]: hi()

In [ ]: # hello name
        def hello(name):
            print "Hello", name

In [ ]: hello('Andrew')

In [ ]: hi()

In [ ]: hi('Tom')

In [ ]: id(hi)

In [ ]: id(hello)
In [ ]: # Python disassembler
        from dis import dis

In [ ]: dis(hi)

In [ ]: dis(hello)

In [ ]: # scoping of 'name'
        name = 'Mary'

In [ ]: # variable name defined inside function hello is scoped inside that function
        hello('Jane')

In [ ]: name

In [ ]: # average (print)
        nums = [1, 3, 6, 2, -4, 2, 3, 6]

In [ ]: running = 0
        for n in nums:
            running += n**2 # sort-of equivalent to running = running + n**2
            y = 3*n + 2*n**2
            print n, n**2, n+10, y, running

In [ ]: def f(x):
            ' A function f of x that calculates a second order function for y and also z'
            y = 3*x + 2*x**2
            z = 5*x**2 + 4*y**(0.5)
            return y, z # tuple packing

In [ ]: f

In [ ]: help(f)

In [ ]: f.__doc__

In [ ]: f.__doc__ = 'a function that calculates y and z, returns two tuple (y,z)'

In [ ]: help(f)

In [ ]: f.color = 'green'

In [ ]: f.color

In [ ]: f.__dict__

In [ ]: f(3)

In [ ]: result = f(3)

In [ ]: result
In [ ]: type(result)
```

```
In [ ]: a = result[0]
        b = result[1]
```

```
In [ ]: a
```

```
In [ ]: b
```

```
In [ ]: type(a)
```

```
In [ ]: type(b)
```

```
In [ ]: f(4)
```

```
In [ ]: s, t = f(4) # tuple unpacking
```

```
In [ ]: s
```

```
In [ ]: t
```

```
In [ ]: nums
```

7 Calculate Average

```
In [ ]: def myaverage(numlist):
            'calculate the mean from an iterable of numbers'
            print 'given numlist:', numlist
            total = 0
            for x in numlist:
                total += x
            mean = total / float(len(numlist))
            print "Got a mean of", mean
            return mean
```

```
In [ ]: nums
```

```
In [ ]: avg = myaverage(nums)
```

```
In [ ]: avg
```

8 Scripts

- Python commands inside a file
- will be run in order from top to bottom
- only `print()` function calls will result in any output to the screen
- file can have any name
- run it with:

```
python path/to/scriptname
```

- Or for Mac and Linux add a shbang line and make the file executable
  - shbang: `#!/usr/bin/env python`
  - executable: `chmod a+x path/to/scriptname`

Demonstration
Use *Anaconda Launcher* to start *Spyder*

8.1 Q. Do I Have To Write My Own Functions?

Answer: Generally, no, you *shouldn't* write your own functions if they already exist.

Q. So what should I do instead?

.

.

A. Use code that others have already written. In Python there are 3.5 ways to do this:

- Built-in functions
  - 50 of them
- Standard library
  - *Batteries Included* means everyone has these
  - 300 packages (aka *modules*), each with many functions included (and *classes*)
- Python Package Index (PyPI): http://pypi.python.org
  - Anaconda includes about 200 of these out of the box
- *methods* on objects ($\frac{1}{2}$)
  - methods are a kind of function
  - this relies on using *classes* that have been written by someone else

9 Reserved Words vs. Built-in Functions

Reserved Words are part of the grammar of Python, in the way brackets, operators, colons, and other symbols are used -- **these are not objects or functions**. Which ones have we seen so far?

.

.

Answer: `for in del def return try except`
There are only 33 of them, about half of which we'll see at least once today:

- 30 reserved words (aka keywords)
  - http://docs.python.org/3.5/reference/lexical_analysis.html#keywords
  - *logic:* `and, not, or, True, False`
  - *namespaces:* `import, from, as, del, global, nonlocal`
  - *object creation:* `class, def, lambda`
  - *functions:* `return, yield`
  - *looping:* `while, for, break, continue`
  - *conditional:* `if, else, elif`
  - *exeptions:* `try, except, finally, raise`
  - *misc:* `pass, assert, with, in, is, None`

Built-in functions are functions that you can use *"out of the box"* with Python. We've only seen a few of these so far. What are they?

.

.

Answer: `id() len() print()`
NOTICE: the difference between `return` as a *reserved word* and `print()` as a *built-in function* * In Python 1.x and 2.x `print` was a *reserved word*, but it always should have been a *function*
Question: How many do you think there are in total? How many do you think there are in Matlab, just for comparison?

10 `__builtin__`

- The Python Language defines a special module called ***builtin*** that is part of the Standard Library

- It contains *functions*, *exceptions*, and *classes* that are very common:
  - 10 core types
    - *int, long, float, bool, complex, str, list, dict, tuple, set*
  - 20 supporting types
    - *file, range, object, ...*
  - 40 exceptions (upper camel case, mostly ending in *Error* or *Warning*)
  - 50 functions
    - Math: *abs min max pow round sum divmod*
    - Logic: *all any apply map filter reduce*
    - Iterable: *len range zip iter next sorted*
    - Misc: *print format reload*
    - File: *open*
    - Check: *callable isinstance issubclass*
    - Convert: *bin chr hex cmp coerce oct ord unichr*
    - Introspect: *dir id vars locals globals hasattr getattr setattr delattr compile eval execfile intern hash repr*
- Any reference lookup that doesn't find the reference in the *local* namespace (first) or the *global* (which means *module*) namespace (second) will check the `__builtin__` modules namespace (third)
- CPython automatically provides a reference to the `__builtin__` module in every *global* namespace but gives it the name `__builtins__`
  - under normal use, you never need to use this module reference
- If the *local* or *global* namespace has a reference that is found in `__builtin__` then the `__builtin__` reference will be masked

```
In [ ]: # average: for loop -> sum
```

```
In [ ]: # average the new way:
```

```
In [ ]: # max, min
```

11 Tuple

- light-weight data structure
- associate a number of entries
- ordered (index look-up)
- like a C `struct`
- immutable

```
In [ ]: # Person
        ian = ('Ian', 42, 'Canadian')
        maggie = ('Margaret', 11, 'British')
```

```
In [ ]: ian[0]
In [ ]: maggie[0]

In [ ]: ian.append('Syracuse')

In [ ]: ian[1]

In [ ]: ian[1] = 41

In [ ]: # person, "constants"
        hilary = ('Hilary', 8, 'American')

In [ ]: hilary

In [ ]: family = [ian, maggie, hilary]

In [ ]: family

In [ ]: emily = ('Emily', 40, 'American')

In [ ]: family.append(emily)

In [ ]: family

In [ ]: family.append('banana')

In [ ]: family

In [ ]: hello

In [ ]: family.append(hello)

In [ ]: family

In [ ]: family[-1]('Steve')

In [ ]: family.pop()

In [ ]: family.pop()

In [ ]: family

In [ ]: family.sort()

In [ ]: family

In [ ]: # good use of a list

In [ ]: # addition of numbers, strings, lists

In [ ]: # multiplication of numbers, strings, lists

In [ ]: # (x,y) points
        points = [(3,7),
                 (4,2),
                 (8,6),
                 (1,5),
                 (6,7)]

In [ ]: for pt in points:
            print pt

In [ ]: for x, y in points:
            print 'x is', x, 'and y is', y
```

12 Dictionary

- light-weight "associative array" data structure
- aka "map" or "hash map"
- associate a number of entries
- name each entry
- unordered (name look-up)
- mutable

Also: * foundational data structure in Python (*"everything is a `dict`"*) * highly optimized (don't bother writing your own hash map) * Python 3.6 provided even more memory optimization (20-25% savings in most cases!)

```
In [ ]: # standard dict creation syntax
        person = {'name': 'John',
              'age': 42,
              'surname': 'Smith'}
```

```
In [ ]: # look-up
        person
```

```
In [ ]: person['age']
```

```
In [ ]: person['surname']
```

```
In [ ]: # change
        person['age'] = 41
```

```
In [ ]: person
```

```
In [ ]: # add
        person['city'] = 'Providence'
```

```
In [ ]: person
```

```
In [ ]: # remove entry
        del person['city']
```

```
In [ ]: person
```

```
In [ ]: # dict constructor
        maggie = dict(name='Maggie', age=11, surname='Jones')
```

```
In [ ]: maggie
```

```
In [ ]: # (k,v) constructor
        person.items()
```

13 Class

- created with a `class` statement
- methods are "just" functions with special invocation handling
  - descriptor protocol (advanced topic, not for now)
  - instance object is passed automatically as first argument
- **dunder** (double-underscore) methods have pre-defined semantics
  - only use ones that are specifed by the language
  - don't make up your own

WARNING What we're doing next is to help you understand how classes work in Python. Only at the **end** will we finally see the conventional way to define a class. Along the road, however, we'll gain insights into Python's handling of classes.

```
In [ ]: # define class
        class Person:
            def __init__(self, name, age, natl): # "self" comes from __new__ calling __init__
                'add attributes to a person instance object'
                self.name = name
                self.age  = age
                self.natl = natl
```

```
In [ ]: Person
```

```
In [ ]: # help
        'Person' in dir()
```

```
In [ ]: help(Person)
```

```
In [ ]: # instance
        joe = Person( "Joe",32,"Irish")
```

```
In [ ]: joe
```

```
In [ ]: joe.__class__
```

```
In [ ]: joe.age
```

```
In [ ]: joe.name
```

```
In [ ]: # change attributes
        joe.name = 'Joseph'
        joe.age  = 42
        joe.natl = 'Canadian'
```

```
In [ ]: joe.name
```

```
In [ ]: joe
In [ ]: joe.age

In [ ]: joe.__dict__

In [ ]: joe.__dict__['natl']

In [ ]: joe.__dict__['color'] = 'green'

In [ ]: joe.__dict__

In [ ]: joe.color

In [ ]: # where are those attributes?

In [ ]: # instantiation via person_init function
        def person_init(p, name, age, natl):
            'add attributes to a person instance object'
            p.name = name
            p.age  = age
            p.natl = natl

In [ ]: maggie = Person( "Maggie",11,"British")

In [ ]: maggie.name

In [ ]: person_init(maggie, 'Margaret', 11, 'British')

In [ ]: maggie.name

In [ ]: maggie.__dict__

In [ ]: # put function into class
        class Person:
            def __init__(p, name, age, natl): # "p" comes from __new__ calling __init__
                'add attributes to a person instance object'
                p.name = name
                p.age  = age
                p.natl = natl

In [ ]: hilary = Person('Hilary', 81, 'American')

In [ ]: hilary.name

In [ ]: hilary.age

In [ ]: # rename function into conventional dunder name
        class Person:
            def __init__(self, name, age, natl): # "p" comes from __new__ calling __init__
                'add attributes to a person instance object'
                self.name = name
                self.age  = age
                self.natl = natl
```

```
In [ ]: # define "birthday()" method to increment age
        class Person:
            def __init__(self, name, age, natl): # "p" comes from __new__ calling __init__
                'add attributes to a person instance object'
                self.name = name
                self.age  = age
                self.natl = natl

            def birthday(self):
                self.age += 1

            # method for readable print
            def __str__(self):
                return "{n} is {a} years old and comes from {c}".format(n=self.name,
                                                                        a=self.age,
                                                                        c=self.natl)
            # method for unambiguous print
            def __repr__(self):
                return "Person('{n}', {a}, '{c}')".format(n=self.name,
                                                        a=self.age,
                                                        c=self.natl)
```

```
In [ ]: emily = Person('Emily', 34, 'American')
```

```
In [ ]: emily
```

```
In [ ]: print emily
```

```
In [ ]: emily.birthday()
```

```
In [ ]: emily
```

```
In [ ]: emily.birthday()
```

```
In [ ]: emily
```

14 Standard Library and Namespaces

```
In [ ]: sin(3.14/2)
        # sin(theta) = opposite/adjacent (in radians)
```

```
In [ ]: import math #  ? have we just done the same as #include <math.h> ???
```

```
In [ ]: 'sin' in dir(math)
```

```
In [ ]: sin(3.14/2)
```

```
In [ ]: # import math
In [ ]: # sin @ pi/2
        math.sin(3.14/2)

In [ ]: # namespaced

In [ ]: # cos for comparison
        math.cos(0)

In [ ]: # atan2 (div by pi mult by 180) for angles of points
        s = math.sin # create local namespace alias to math.sin

In [ ]: s(3.14/2)

In [ ]: import math as m # import the math module, but use "m" as the local reference
                         # saves us from doing "m = math"

In [ ]: m.sin(3.14/2)

In [ ]: from math import atan2

In [ ]: from math import sin

        def f(x):
            y = x + sin(x)
            return y

In [ ]: from dis import dis
        dis(f)

In [ ]: from math import sin

        def f(x):
            s = sin
            y = x + s(x)
            return y

In [ ]: dis(f)

In [ ]: from math import acos as ac #selective import with alias

In [ ]: dir(math)

In [ ]: # what else is in the math namespace? dir()

In [ ]: # that isn't very helpful! What did we use before to find out about "average()"?
```

14.1 Import Aliasing

But I use sin and cos a lot! This namespace thing is going to be very burdensome!

```
In [ ]: # option 1: setup your own alias "m"
        m = math
```

```
In [ ]: # option 2: alias "s" and "c"
        s = math.sin
        c = math.cos
```

```
In [ ]: # Scratch that, there are better ways to do both of those things:
        # option 3: alias the module at import
        import math as m
```

```
In [ ]: # option 4: selectively import object from module into current namespace
        from math import sin, cos
```

```
In [ ]: # option 5: selectively import AND alias
        from math import sin as s, cos as c
```

14.2 Anything Else About The Python Standard Libary?

Only that it is totally amazing and you should avoid it at your peril: * stable * always available * optimized * written in C if necessary * documented * 100% test coverage * used extensively in the field * how likely do you think it is you'll be the first to discover a bug?

14.3 How Do I Learn More?

- Come to events like this!
- Google for what you need. GIYF: it will most likely point you to python.org and if not?
- Skim Chapter 10 of the Python Tutorial: A Brief Tour of the Standard Library
  - and also possibly Chapter 11: Part 2
- Just browse or search the Official Standard Library Module Index
  - make sure you're checking the version that matches your Python version
- Ask a friend!
- As a last resort, search the Complete Index Of Everything Inside The Standard Library

14.4 So what's this Anaconda thing, then?

- 200+ of the most commonly used, publicly available, open source, libraries and tools for computational science in Python **that are not found in the Standard Library**
- They all "Just Work" no compilation or build chains or manual dependency resolution required
- A further 200 packages that you can install on-demand:
  - `conda install biopython`
    - 2MB -- do this if you want to try it out

- 250 MB instead of 25 MB
- Available for free, for ever, for Windows, Mac, Linux (and Raspberry Pi)
- More than just Python (don't do these now!)
  - `conda install -c r r-essentials`
  - `conda install -c ijstokes julia`
    - v0.3.10, OS X only!

14.5 But Python Package Index and `pip`?

- the Python Package Index has over 70,000 community contributed packages
- `pip install fred`
- plays nicely with Anaconda and `conda` (so feel free to mix-and-match)
- remember that no one checks the packages in PyPI: *caveat emptor*!

15 Files

- best bet is to use a data-format-specific library that can read and write files from disk for you (e.g. HDF5)
- but sometimes you need to DIY
- and you should know how to do this anyway

```
In [ ]: # save points to a file
        points
        with open('xy_basic.tab', 'w') as fh:
            for x, y in points:
                fh.write('{x}\t{y}\n'.format(x=x, y=y))
```

```
In [ ]: !cat xy_basic.tab
```

```
In [ ]: # read in file, shifted by (5, 5)
        # comment: split, append, int, close
        result = []
        with open('xy_basic.tab') as datafile:
            for line in datafile:
                x, y = line.split()
                x = int(x)
                y = int(y)
                result.append((x,y)) # inner round-brackets create 2-tuple (x,y)
```

16 Methods

- operations you can perform on an object -- e.g.
  - Perl: `sort(array)` is a function call, which returns a sorted array
  - Python: `array.sort()` is a method call, which acts on the array and sorts it

```
In [ ]: # copy with slice
In [ ]: # reverse

In [ ]: # append

In [ ]: # extend

In [ ]: # multi-list stuff

In [ ]: # wrong append

In [ ]: # make a prediction: what is nums[-1]?

In [ ]: # What is the correct way to "add" a list?

In [ ]: # list addition (numbers)

In [ ]: # list addition (strings)

In [ ]: # addition

In [ ]: # but has nums changed?

In [ ]: # self-increment

In [ ]: # check id()

In [ ]: # not the same as "a = a + [1,2,3]"

In [ ]: # sentence (string)

In [ ]: # title

In [ ]: # lower

In [ ]: # split

In [ ]: # chain: lower split

In [ ]: # split on letter

In [ ]: # shortcut: how to make a list of words (and methods on literals)

In [ ]: # format (name, age)

In [ ]: # number formatting :.2f
```

More references for string formatting: * https://mkaz.github.io/2012/10/10/python-string-format/ * https://pyformat.info/

17 Booleans, Conditionals, Comprehensions

```
In [ ]: # -12 to 20, steps of 3
        vals = range(-12,20,3)
```

```
In [ ]: # threes and evens
        [v for v in vals if v % 2 == 0]
```

```
In [ ]: [v for v in vals if abs(v) == 3]
```

```
In [ ]: # keep short colors 'green red blue black yellow pink gold silver'
        colors = 'green red blue black yellow pink gold silver'.split()
```

```
In [ ]: [c for c in colors if len(c) <= 4]
```

```
In [ ]: # How would you create a list of 20 random integers?
```

17.1 Comprehension Exercise

Create list comprehensions that: * filter the list selecting only values greater than 3 * create a new list containing only the odd numbers, but multiply these by 10 * odd numbers can be found by testing if `v%2 == 1`

Time: 5 minutes

18 Numpy and Pandas

Numpy (released 2006 by Travis Oliphant, founder of Continuum) provides the foundation for numerical computing in Python: * defines an `ndarray` that can be used for vector (and matrix) computations * functions and methods supporting linear algebra * implemented in C * fast * memory efficient

Pandas (released 2009 by Wes McKinney, maintainer is now Jeff Reback from Continuum) * provides R-style `DataFrame` class * many convenience functions * facilitates interaction with spreadsheets (Excel, CSV) or database tables * built on top of Numpy

19 Next Steps

- find a project where you can start using Python
- refer to the follow-up tutorials listed at the top
- join one of the Boston area Python and Data Science meetups:
  - Boston Python User Group
  - Search for "Python" or "Data Science" at meetup.com near you
    - pro-tip: make sure the group is active and relevant before signing up!