Introduction to Python
General Introduction, Basic Data Types, Functions

Christopher Barker

UW Continuing Education

October 1, 2013

Table of Contents

Instuctors

Christopher Barker: `PythonCHB@gmail.com`

TA:

Chris' History

First computer:

- Commodore Pet – 8k RAM
  - Basic

High School:

- PDP 11 – paper printer terminal 200baud modem
  - Basic

Chris' History

First computer:

- Commodore Pet – 8k RAM
  - Basic

High School:

- PDP 11 – paper printer terminal 200baud modem
  - Basic

College:

- Pascal: VAX/VMS 750
- Scheme: Unix VAX 780

Chris' History

First computer:

- Commodore Pet – 8k RAM
  - Basic

High School:

- PDP 11 – paper printer terminal 200baud modem
  - Basic

College:

- Pascal: VAX/VMS 750
- Scheme: Unix VAX 780

Then a long Break: Theater Arts Major, Scenery, Lighting...

Chris' History

Back to School: PhD Coastal Engineering

- DOS / Windows 3.1
  - FORTRAN
  - MATLAB
  - Discovered Linux (RedHat 2.0)

Chris' History

Back to School: PhD Coastal Engineering

- DOS / Windows 3.1
  - FORTRAN
  - MATLAB
  - Discovered Linux (RedHat 2.0)

Now:

- Oceanographer for NOAA
- Oil Spill Modeling
- Software Development

Chris' History

Back to School: PhD Coastal Engineering

- DOS / Windows 3.1
  - FORTRAN
  - MATLAB
  - Discovered Linux (RedHat 2.0)

Now:

- Oceanographer for NOAA
- Oil Spill Modeling
- Software Development

Gave TCL a try.....

Chris' History

Back to School: PhD Coastal Engineering

- DOS / Windows 3.1
  - FORTRAN
  - MATLAB
  - Discovered Linux (RedHat 2.0)

Now:

- Oceanographer for NOAA
- Oil Spill Modeling
- Software Development

Gave TCL a try.....

Gave Perl a try.....

Chris' History

Discovered Python in 1998

- It could do what Perl could do,
  - what TCL could do, what MATLAB could do,
- But I liked it – it fit my brain

My Python use now:

- Lots of text file crunching / data processing
- Desktop GUIs (wxPython)
- computational code
- wrapping C/C++ code
- web apps (Pylons, Pyramid)
- GIS processing
- Ask me about "BILS"

Who are you?

A bit about you:

- name
- What do you do at Islion?
- programing background (languages)

Class Structure

github project

`https://github.com/UWPCE-PythonCert/IntroToPython`

Syllabus:

`https://github.com/UWPCE-PythonCert/IntroToPython/blob/master/Syllabus.rst`

Code, etc:

git:

`https://github.com/UWPCE-PythonCert/IntroToPython.git`

Class Structure

Class Time

- Some lecture
- Lots of demos
- Lots of hand-on practice
- Interrupt me with questions – please!

Homework

- Assigned at each class
- Due Sunday night
- I'll review at the next class

Lightning Talks

Lightning talks

- 5 minutes (including setup) - no kidding!
- Every student will give one
- Purposes: introduce yourself, share interests, also show Python applications
- Any topic you like, that is related to Python – according to you!

Python Ecosystem

Used for:

- CS education (this course!)
- Application scripting (GIS, GNU Radio, Blender...)
- Systems administration and "glue"
- Web applications (Django etc. etc. etc.)
- Scientific/technical computing (a la MATLAB, Mathematica, also BioPython etc. ..)
- Software tools (automated software testing, distributed version control, ...)
- Research (natural language, graph theory, distributed computing, ...)

An unusually large number of niches – versatile

Python Ecosystem

Used by:

- Beginners
- Professional software developers, computer system administrators, ...
- Professionals OTHER THAN computer specialists: biologists, urban planners, ...

An unusually large number of types of users – versatile

You can be productive in Python WITHOUT full-time immersion!

Python Features

Gets many things right:

- Readable – looks nice, makes sense
- No ideology about best way to program – object-oriented programming, functional, etc.
- No platform preference – Windows, Mac, Linux, ...
- Easy to connect to other languages – C, Fortran - essential for science/math
- Large standard library
- Even larger network of external packages
- Countless conveniences, large and small, make it pleasant to work with

What is Python?

- Dynamic
- Object oriented
- Byte-compiled
- interpreted
- ....

Python Features

Features:

- Unlike C, C++, C#, Java ... More like Ruby, Lisp, Perl, Matlab, Mathematica ...
- Dynamic - no type declarations
  - programs are shorter
  - programs are more flexible
  - less code means fewer bugs
- Interpreted - no separate compile, build steps - programming process is simpler

What's a Dynamic language

Strong, Dynamic typing.
- Type checking and dispatch happen at run-time

```
X = A+B
```

What's a Dynamic language

Strong, Dynamic typing.
- Type checking and dispatch happen at run-time

```
X = A+B
```

- What is A?
- What is B?
- What does is mean to add them?

What's a Dynamic language

Strong, Dynamic typing.
- Type checking and dispatch happen at run-time

```
X = A+B
```

- What is A?
- What is B?
- What does is mean to add them?

- A and B can change at any time before this process

Duck Typing

"If it looks like a duck, and quacks like a duck – it's probably a duck"

Duck Typing

"If it looks like a duck, and quacks like a duck – it's probably a duck"

If an object behaves as expected at run-time, it's the right type.

Python Versions

Python 2.*

"Classic" Python – evolved from original

Python 3.* ("py3k")

Updated version – removed the "warts" allowed to break code (but really not all that different). Not all that well adopted yet – many packages not supported.

This class uses Python 2.7 not Python 3

Implementations

- Jython (JVM)
- Iron Python (.NET)
- PyPy – Python written in Python (actually RPy...)

We will use CPython 2.7 from python.org for this course.

A Tiny Bit of History

Invented/developed by Guido van Rossum in 1989 – first version was written on a Mac. Time of origin similar to TCL and Perl.

Date	Version
Dec 1989	started
Feb 1991	0.9.0
Jan 1994	1.0.0
Apr 1999	1.5.2
Sept 2006	2.5
Dec 2008	3.0
Jul 2010	2.7

GvR at Google – still the BDFL

Code swarm for Python history: `http://vimeo.com/1093745`

Using Python

All you need for Python:

- A good programmer's text editor
  - Good Python mode
  - Particularly indentation!
- The command line to run code
- The interactive shell
  - regular interpreter
  - `IPython` is an excellent enhancement `http://ipython.org/`

There are lots of Editors, IDES, etc.:
maybe you'll find one you like.

Running Python Code

- At an interpreter prompt:

```
$ python
>>> print 'Hello, world!'
Hello, world!
```

Running Python Modules

Running Modules

– a file that contains Python code, filename ends with `.py`

1. `$ python hello.py` – must be in current working directory
2. `$ python -m hello` – any module on PYTHONPATH anywhere on the system
3. `$ ./hello.py` – put `\#!/usr/env/python` at top of module (Unix)
4. `$ python -i hello.py` – import module, remain in interactive session
5. `>>> import hello` – at the python prompt – importing a module executes its contents
6. `run hello.py` – at the IPython prompt – running a module brings the names into the interactive namespace

Documentation

`www.python.org` docs:
`http://docs.python.org/index.html`

Particularly the library reference:
`http://docs.python.org/library/index.html`

(The tutorial is pretty good, too)

PEPs

`http://www.python.org/dev/peps/`

PEP 1 PEP Purpose and Guidelines

PEP 8 Style Guide for Python Code

PEP 20 the Zen of Python (`import this`)

pydoc

Suite of tools for processing "docstrings"
And an online source at the interpreter:

```
>>> from pydoc import help
>>> help(int)
Help on class int in module __builtin__:

class int(object)
 |  int(x[, base]) -> integer
 |
 |  Convert a string or number to an integer, if possible.
 ...
```

or: `$ pydoc`
(but I prefer IPython's ?)

Documentation

google

But be careful!

Lots of great info out there!

Most of it is opinionated and out of date.
(might still be correct, though!)

Lab

Getting everyone on-line and at a command line.

- Log in
- Do a `git clone` or `svn checkout` of the project
- Start up the Python interpreter:
  `$ python` (`ctrl+D` to exit)
- Run `hello.py` (in the `week-01/code` dir)
- Create a file in your editor and save it
- Start up `IPython`
  `$ ipython` (also `ctrl+D` to exit)
- Run `hello.py` in `IPython`
- use `?` in `IPython` on anything...
- if you have time:
  `http://learnpythonthehardway.org/book/ex1.html`
  `http://learnpythonthehardway.org/book/ex2.html`

Values, expressions, and types

Values (data) vs. variables (names with values)

- Values are pieces of unnamed data: `42`, `'Hello, world'`,
- In Python, all values are objects
  Try `dir(42)` - lots going on behind the curtain! (demo)
- Every value belongs to a type: integer, float, str, ... (demo)
- An expression is made up of values and operators, is evaluated to produce a value: `2 + 2`, etc.
- Python interpreter can be used as a calculator to evaluate expressions (demo)
- Integer vs. float arithmetic (demo)
- Type errors - checked at run time only (demo)
- Type conversions (demo)

Variables

Variables are names for values - objects

- Variables dont have a type; values do – this is where the dynamic comes from

```
>>> type(42)
<type 'int'>
>>> type(3.14)
<type 'float'>
>>> a = 42
>>> b = 3.14
>>> type(a)
<type 'int'>
>>> a = b
>>> type(a)
<type 'float'>
```

Assignment

Assignment is really name binding:

- Attaching a name to a value
- A value can have many names (or none!)

= assigns (binds a name)

`del` only unbinds a name

Multiple Assignment

```
a, b = 1, 2
```

This will come in handy later...

(demo)

equality and identity

== checks equality

`is` checks identity

`id()` queries identity

(demo)

Operator Precedence

Operator Precedence determines what evaluates first:

`4 + 3 * 5 != (4 + 3) * 5` – Use parentheses !

Precedence of common operators:

Arithmetic

`**`

`+x, -x`

`*, /, %`

`+, -`

Comparisons:

`<, <=, >, >=, !=, ==`

Boolean operators:

`or, and, not`

Membership and Identity:

`in, not in, is, is not`

string literals

```
'a string'
"also a string"
"a string with an apostophe: isn't it cool?"
' a string with an embedded "quote" '
""" a multi-line
string
all in one
"""
"a string with an \n escaped character"

r'a "raw" string the \n comes through as a \n'
```

key words

A bunch:

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

and the built-ins..

Try this:

```
>>> dir(__builtins__)
```

Lab

From LPTHW

`http://learnpythonthehardway.org/book/ex3.html`

`http://learnpythonthehardway.org/book/ex4.html`

`http://learnpythonthehardway.org/book/ex5.html`
(and 6 – 8 if you get bored...)

Functions

Minimal Function

```
def <name>():
    <statement>
```

Functions

Minimal Function

```
def <name>():
    <statement>
```

Pass Statement (Note the indentation!)

```
def <name>():
    pass
```

Functions: def

`def` is a statement:

- it is executed
- it creates a local variable

function defs must be executed before the functions can be called

Functions: def

`def` is a statement:

- it is executed
- it creates a local variable

function defs must be executed before the functions can be called

functions call functions – this makes a stack – that's all a trace back is

Functions: Call Stack

```
def exceptional():
    print "I am exceptional!"
    print 1/0
def passive():
    pass
def doer():
    passive()
    exceptional()
```

Functions: Tracebacks

```
I am exceptional!
Traceback (most recent call last):
  File "functions.py", line 15, in <module>
    doer()
  File "functions.py", line 12, in doer
    exceptional()
  File "functions.py", line 5, in exceptional
    print 1/0
ZeroDivisionError: integer division or modulo by zero
```

Functions: return

Every function ends with a `return`

```
def five():
    return 5
```

Actually simplest function

```
def fun():
    return None
```

Functions: return

if you don't put `return` there, python will:

```
In [123]: def fun():
   .....:     pass
In [124]: result = fun()
In [125]: print result
None
```

note that the interpreter eats `None`

Functions: return

Only one return statement will ever be executed.

Functions: return

Only one return statement will ever be executed.

Ever.

Functions: return

Only one return statement will ever be executed.

Ever.

Anything after a executed return statement will never get run.

This is useful when debugging!

Functions: return

functions can return multiple results

```
def fun():
    return 1,2,3

In [149]: fun()
Out[149]: (1, 2, 3)
```

Functions: return

remember multiple assignment?

```
In [150]: x,y,z = fun()

In [151]: x
Out[151]: 1

In [152]: y
Out[152]: 2

In [153]: z
Out[153]: 3
```

Functions: return

Actually a tuple of results...

```
In [154]: t = fun()

In [155]: t
Out[155]: (1, 2, 3)

In [156]: type(t)
Out[156]: tuple
```

Multiple assignment is really "tuple unpacking"

Functions: parameters

function parameters: in definition

```
def fun(x, y, z):
    q = x + y + z
    print x, y, z, q
```

x, y, z are local names – so is q

Functions: arguments

function arguments: when calling

```
def fun(x, y, z):
    print x, y, z

In [138]: fun(3, 4, 5)

3 4 5
```

Functions: local vs. global

```
x = 32
y = 33
z = 34
def fun(y, z):
     print x, y, z

In [141]: fun(3,4)

32 3 4
```

x is global, y, z are local

Functions: local vs. global

```
x = 3
def f():
    y = x
    x = 5
    print x
    print y
```

What happens when we call `f()`?

Functions: local vs. global

Gotcha!

```
In [134]: f()
---------------------------------------------------------------
UnboundLocalError                                Traceback (most r
/Users/Chris/<ipython-input-132-9225fa53a20a> in f()
      1 def f():
----> 2     y = x
      3     x = 5
      4     print x
      5     print y
```

you are going to assign x – so it's local

Scopes

There is a `global` statement

Scopes

There is a `global` statement

Don't use it!

Scopes

good discussion of scopes:

`http://docs.python.org/tutorial/classes.html#python-scopes-and-namespaces`

Recursion

Recursion is calling a function from itself.

Max stack depth, function call overhead.

Because of these two(?), recursion isn't used **that** often in Python.

Lab: functions

write a function that:

- takes a number and returns the square and cube of that number – use local variables to store the results
- takes a string and a number, and returns a new string containing the input string repeated the given number of times
- uses both global and local variable to compute a result.
- calls another function to do part of its job.
- take some code with functions, add this to each function:

```
print locals()
```

- computes the factorial with a recursive function (needs something we haven't covered yet..)

Lightning Talks

Assign times for lightning talks

Let's use Python for that!

Wrap Up

Assignment – Due midnight, Sun, June 24.

Think Python: Chapters (1), 2, 3, 4, 5, 6, 7, 8

Pick something you'd like to automate that Python may be able to do. Write out a description of the problem

Coding is the only way to learn to code.
CodingBat exercises are a good way to build skills.

- visit `http://codingbat.com`
- sign up for an account
- goto prefs page
- Share To: `PythonCHB@gmailcom`

Do two exercises from CodingBat: Warmup-1