General Introduction, Basic Data Types, Functions, Conditionals and Looping

Christopher Barker

IRIS

October 15, 2013

Table of Contents

Instructor

Christopher Barker: `PythonCHB@gmail.com`

First computer: Commodore Pet – 8k RAM, Basic

Passed through: Pascal, Scheme, Fortran

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

PhD Coastal Engineering: Fortran, then Linux and MATLAB

Now: Discovered Python in 1998 – Never looked back

Python

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 IRIS?
- programing background (languages)

Class Structure

github project

`https://github.com/PythonCHB/IRIS_Python_Class`

Presentations, Sample Code, etc:

```
git clone https://github.com/PythonCHB/IRIS_Python_Class.git
```

Class Structure

Very informal structure: more tutorial/workshop than formal class

Class Time:

- Some lecture, lots of demos
- Lab time: lots of hand-on practice
- Later, Rinse, Repeat.....

Interrupt me with questions – please!

(Some of the best learning prompted by questions)

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 it 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 it 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 3.* (“py3k”)

Updated version – removed the “warts” allowed to break code

(but really not all that different)

Adoption is growing fast, but a few key packages still not supported. (`https://python3wos.appspot.com/`)

We’ll be using Python 2.7

Implementations

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

CPython: Interpreter implimented in C
– allows close connection with C libraries (and C++, Fortran, etc)

We will use CPython 2.7 for this workshop

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)

docstrings

"docstrings": docs embedded in the code

Designed to be read when working on/with the code

But can be accessed interactively

Best / Easiest way: IPython's ?

```
In [123]: list?
Type:       type
String Form:<type 'list'>
Namespace:  Python builtin
Docstring:
list() -> new empty list
list(iterable) -> new list initialized from iterable's iter
```

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

Get the gitHub project:

https://github.com/PythonCHB/IRIS_Python_Class

https://github.com/PythonCHB/IRIS_Python_Class.git

Lab

Getting everyone on-line and at a command line.

- Do a `git clone` of the project
- Start up the Python interpreter:
  `$ python` [`ctrl+D` (`ctrl+Z` on Windows)or `exit()` to exit]
- Run `hello.py` (in the `Session01/code` dir)
- Open `hello.py` in your editor, change it, and save it.
  - (Optional) Start up `IPython`
    `$ ipython` (also `ctrl+D`, etc. 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`
  ...

Code structure

Each line is a piece of code

Comments: everything following a # is a comment

Expression: something that results in a value: 3+4

Statement: Line of code that does not return a value: `print "this"`

Blocks of code are delimited by a colon and indentation:

```
def a_function():
    a_new_code_block
end_of_the_block
```

The print statement

Kind of obvious, but handy when playing with code:

`print something` prints `something` to the console.

Can print multiple things: `print "the value is", 5`

Automatically adds a newline.

You can suppress the newline with a comma:

```
print "the value is",
print 5
```

Any python object can be printed
(though it might not be pretty...)

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 don't 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)

+= also an assignment: `a += 1` same as `a = a+1`
also: `-=`, `*=`, `/=`, `**=`, `\%=`
(not quite – really in-place assignment for mutables....)

Multiple Assignment

You can assign multiple variables from multiple expressions in one statement

`i, j = 2 + x,  3 * y` # commas separate variables on lhs, exprs on rhs

Python evaluates all the expressions on the right before doing any assignments

`i, j = j, i` # parlor trick: swap in one statement

These are just tricks, but multiple assignment is more helpful in other contexts

(more on what's really going on later...)

(demo)

Deleting

You can't actually delete anything in python...

`del` only unbinds a name

```
a = 5
b = a
del a
```

The object is still there...python will only delete it if there are no references to it.

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

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

What is a function?

A function is a self-contained chunk of code

You use them when you need the same code to run multiple times, or in multiple parts of the program.

(DRY)

Or just to keep the code clean

Functions can take and return information

Functions

Minimal Function does nothing

```
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: 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 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
- calls another function to do part of its job.
- Problems in `Session01\draw_grid.rst`

Functions: local vs. global

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

fun(3,4)
```

```
32 3 4
```

x is global, y and z local

Use global variables mostly for constants

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.

(demo: factorial)

Tuple Unpacking

Remember: `x,y = 3,4` ?

Really “tuple unpacking”: `(x, y) = (3, 4)`

This works in function arguments, too:

```
>>> def a_fun( (a, b), (c, d) ):
...     print a, b, c, d
...
>>> t, u = (3,4), (5,6)
>>>
>>> a_fun(t, u)
3 4 5 6
```

(demo)

Lab: more with functions

Write a function that:

- computes the distance between two points:
  dist = sqrt((x1-x2)**2 + (y1-y2)**2)
  using tuple unpacking...
- Take some code with functions, add this to each function:
  `print locals()`
- Computes the Fibonacci series with a recursive function:
  f(0) = 0; f(1) = 1
  f(n) = f(n-1) + f(n-2)
  0, 1, 1, 2, 3, 5, 8, 13, 21, ...
  (If time: a non-recursive version)

Follow Up

Recommended Reading:

- Think Python: Chapters 1–7
- Dive Into Python: Chapters 1–3
- LPTHW: ex. 1–10, 18-21

Coding is the only way to learn to code: CodingBat exercises are a good way to build skills.
visit `http://codingbat.com`
Do a few – its fun!