CMSC330: Python Basics

Chris Kauffman

Last Updated:
Thu Aug 31 09:23:44 AM EDT 2023

Logistics

Reading

The Python Tutorial:
https://docs.python.org/3/tutorial/index.html

- Skim, skip around, experiment
- Any other python reference should be good too
- Idea is to get a quick high level understanding

Goals

- Understand basic syntax of Python
- Relate Python to Java
- Identify imperative nature of both Languages

Python

- Development started in late 1980s
- Version 2 released in 2000, fairly recognizable
- Version 3 released in 2008, was NOT backwards compatible

```
print "Hello"    # version 2
print("Hello")   # version 3
```

- Created a vast schism; still some version 2 code in use out there today
- "Fun" to program in: do a lot with few lines of code
- Relatively straight-forward to interface with C
- Often used as an intro language due to its friendly looking syntax (both my old university did and UMD is rumored to be looking to try Python in 131)
- Wildly popular in all realms of computing from web frameworks to machine learning / data science to robotics, great to have on your resume

Python's Primary author is Dutch coder Guido von Rossum, dubbed "Benevolent dictator for life" by the development community.

Every Programming Language

Look for the following as it should almost always be there

- ☐ Comments
- ☐ Statements/Expressions
- ☐ Variable Types
- ☐ Assignment
- ☐ Basic Input/Output (printing and reading)
- ☐ Function Declarations
- ☐ Conditionals (if-else)
- ☐ Iteration (loops)
- ☐ Aggregate data (arrays, records, objects, etc)
- ☐ Library System

Exercise: Collatz Computation An Introductory Example

- `collatz.py` prompts for an integer and computes the Collatz Sequence starting there
- The current number is updated to the next in the sequence via

  ```
  if cur is EVEN cur=cur/2; else cur=cur*3+1
  ```

- This process is repeated until it converges to 1 (mysteriously) or the maximum iteration count is reached
- The code demonstrates a variety of Python features and makes for a great crash course intro
- With a neighbor, study this code and identify the features you should look for in every programming language

Exercise: Collatz Computation An Introductory Example

```python
# collatz.py: collatz computation
verbose = True                    # global var

def collatz(start,maxsteps):      # function
  cur = start
  step = 0
  if verbose:
    print("start:",start,"maxsteps:",maxsteps)
    print("Step  Current")
    print(f"{step:3}: {cur:5}")
  while cur != 1 and step < maxsteps:
    step += 1
    if cur % 2 == 0:
      cur = cur // 2
    else:
      cur = cur*3 + 1
    if verbose:
      print(f"{step:3}: {cur:5}")
  return (cur,step)

# executable code at global scope
start_str = input("Collatz start val:\n")
start = int(start_str)

(final,steps) = collatz(start, 500)
print(f"Reached {final} after {steps} iters")
```

Look for... Comments, Statements/Expressions, Variable Types, Assignment, Basic Input/Output, Function Declarations, Conditionals, Iteration, Aggregate Data, Library System

```
>> python collatz.py
Collatz start val:
10
start: 10 maxsteps: 500
Step  Current
  0:    10
  1:     5
  2:    16
  3:     8
  4:     4
  5:     2
  6:     1
Reached 1 after 6 iters
```

Answers: Collatz Computation An Introductory Example

- ☒ Comments: `# comment to end of line`
- ☒ Statements/Expressions: written plainly, no semicolons, stuff like `a+b` or `n+=2` is old hat; Boolean expressions available via `x and y` implicating `z or w` is likely around
- ☒ Variable Types: string, integer, boolean are obvious as values, no type names mentioned save the conversion from string to integer via the `int(str)` function
- ☒ Assignment: via `somevar = avalue`
- ☒ Basic Input/Output (printing and reading): `print()` / `input()`
- ☒ Function Declarations: `def funcname(param1,param2):`
- ☒ Conditionals (if-else): `if cond:` and `else:`, also `elif:`
- ☒ Iteration (loops): clearly `while cond:`, others soon
- ⊟ Aggregate data (arrays, records, objects, etc): `(python,has,tuples)` and others we'll discuss soon
- ☐ Library System: soon

A Few Oddities

- Python has two division operators `a / b` for floating point division, `a // b` for integer division. Dynamic types make this easy to forget and likely to cause errors

```
>>> 11 / 3              # float div
3.6666666666666665
>>> 11 // 3             # int div
3
>> 11.99 / 3.99         # float div
3.005012531328320 7
>>> 11.99 // 3.99       # what now?
3.0
```

- Python has several means of formatted output; we'll favor

```
print("substitue x: {} and y: {}".format(x,y)) # older, position subs
print(f"substitue x: {x} and y: {y}")          # newer, symbolic subs
#     ^ f for format
```

REPL: Read-Evaluate-Print Loop

- Python features a REPL to interactively interpret Python statements on the fly
- Allows for easy experimentation and testing of code
- REPLs appear in many forms, are closely associated with Dynamic languages (like command line shells)

```
shell>> python

Python 3.11.3 (main, Jun  5 2023, 09:32:32) [GCC 13.1.1 20230429] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> 1+2
3
>>> "hello" + " world"
'hello world'
>>> x=1+2
>>> s="hello" + " world"
>>> print(x)
3
>>> print(s)
hello world
>>> exit()

shell>>
```

Python Runs as a "Scripting" Language

- "Scripting Language" is a loose term associated with PLs that favor short programs in text files that are run through an interpreter program, in this case `python`
- Examples include Python (**caveats**), Javascript, Shell Scripts, Awk, Lua, Perl TCL, and myriad others
- *Usually* programs are directly executed by their interpreter by reading some statements, executing, reading more statements, executing, etc. (e.g. NOT compiled to a lower form)

```
shell>> cat expressions.py
1+2                         # expressions output in the REPL
"hello" + " world"          # but not in scripts

x=1+2                       # assignment has no output
s="hello" + " world"        # in the REPL or as scripts

print(x)                    # print() producees output in
print(s)                    # REPL and in scripts

shell>> python expressions.py
3
hello world

shell>>
```

The Whitespace Thing

- Python employs an unusual convention: it is NOT whitespace neutral
- A Colon (:) plus Indentation indicates nested elements in python like the bodies of functions, loops, conditionals, class bodies, and other syntatic elements of the language
- Python Zen: Beautiful is better than ugly.

  . . . and apparently {} is ugly
- Takes just a little some getting used to and enforces a more uniform style than is present in most other PLs

```
1  # indent_error.py:
2  # proper indent
3  if 5 > 2:
4    print("5 is bigger")
5    print("that is all")
6  else:
7    print("something is amiss")
8
9  # indentation error
10 if 6 > 2:
11   print("6 is bigger")
12     print("all is well") # !!!
13 else:
14   print("how strange")
15
```

```
shell>> python indent_error.py
  File "indent_error.py", line 12
    print("all is well")
IndentationError: unexpected indent

shell>> pylint indent_error.py
************* Module indent_error
indent_error.py:12:4: E0001: Parsing fail
'unexpected indent (<unknown>, line 11)'
```

Built-in Data Types

- One reason to program in Python is its out-of-the box support for common data types with built-in syntax
- Common tasks in computing benefit greatly from its "batteries included" approach
- **NOTE:** This course isn't about data structures, assumes familiarity with extensible arrays, hash tables, mathematical sets; if those aren't in your utility belt, review and **play with them in the REPL**

Tour (briefly) `datatypes.py` to examine some of these

Samples of datatypes.py

```
# LISTS: indexed, mutable collections
alist1 = ["a","b","c","c"]
alist2 = [4, "five", 6.7]

# numerically indexed
print(alist1[0])                # 'a'
print(alist1[2])                # 'c'
print(alist2[-1])               # 6.7, handy!

alist2[0]=3                     # mutable
# alist2 = [3, "five", 6.7]
alist1.append("d")              # extendable
# alist1=["a","b","c","c","d"]

for item in alist1:             # iterate
  print(item)

####################
# TUPLES: indexed, immutable collections
atup1 = (1,2,3)                 # a tuple
atup2 = (4,"five",6.7)

(x,y,z) = (1,"hi",True)         # destructure bind
```

```
####################
# DICTIONARY: key-value mappings
amap1 = {"a":1, "b":2}          # dictionary
amap2 = {3:"c", 6.7:True}       # e.g. hashmap

print(amap1["b"])               # lookup: 2
amap1["c"] = 3                  # new key/val

del amap2[3]                    # delete key
# amap2 = {6.7:True}
amap2.pop(6.7)                  # delete key
# amap2 = {}
amap2.pop(9.9,None)             # delete safely

if "a" in amap1:                # check for key
  print("a is present")

for key in amap1:               # iterate
  print(key,amap1[key])         # over keys

for key,val in amap1.items(): # iterate
  print(key,val)                # over key/vals
```

Examine these and comment on how this stuff differs from Java

Also present are basic types (string, Boolean) and Sets of unique values

Modules in Python

- Python code divides into **modules**, typically a collection of global variables / functions / classes in from the same file
- Modules assume the name of their source file
- Code from other modules is loaded via `import` statements
- In simplest case, functions are referenced via "dot" syntax from their module (similar to Java's conventions)

```
# moduleA.py: creates moduleA
import moduleB     # loads moduleB

def main():
  ...
  moduleB.bfunc()  # use code from moduleB
  ...
```

```
# moduleB.py: creates moduleB
def bfunc():
  ....
```

Namespaces

- **Namespace**: A collection of **unique** names grouped together, usually associated with functions / data in a source file
- Programming Environments take differing approaches to managing name spaces in code
  - C: one big namespace, all global names must be unique
  - Java: classes are namespaces, same named functions names can exist in multiple classes; same named classes must be in different packages, `import some.package.Class;`
  - Python: modules are namespaces, same named functions can exist in different modules
  - C++: namespaces are explicit, house all code in one, access other namespaces via `::` syntax, `using namespace std` imports members
- Python's module system has lots of nuance (skim) but is overall serviceable very serviceable
- Python `import` statements also give some control about naming things

import Statement Examples

```
# standard, simple module import
>>> import somelibrary
>>> somelibrary.fibiter(10)
55

# rename long module to abbreviation
>>> import somelibrary as sl
>>> sl.fibiter(10)
55

# import single (or multiple) elements
>>> from somelibrary import fibiter
>>> fibiter(10)
55

# import everything
>>> from somelibrary import *
>>> fibiter(10)
55
```

Python "main()" Functions

- Python applications larger than a single file should follow Entry Point conventions[1]
- `import moduleA` will execute any top-level code in the module when it loads
- Typically want only top-level definitions of functions and initialization of module-level (global) variables
- If a module has an entry point use this convention:

```
def afunc():                        # normal function
  ...

def main():                         # convention: module entry point
  print("In moduleA.main() with __name__",__name__)
  ...

if __name__ == '__main__':          # is module is the "application"?
  print("moduleA executable code found; running moduleA.main()")
  main()
```

Demo via `moduleA.py`

[1] Entry points are where code starts executing you run a whole application

Modes Dangers of Module-Level Executable Code

- Each `import` of a module executes all code in it
- If there is output, it will show on an `import`
- Avoid this or other costly setup
- Demo via `modLoud.py` and `modSleepy.py`

```
shell>> python
Python 3.11.3

>>> import modLoud
This is why
I don't leave the house
You say the coast is clear
But you won't catch me out

>>> import modSleepy
But I am le tired
Then, have a nap
<delay...>
Now fire!
```

Exercise: Standard Scoping Rules

- Below are two code examples
- Predict the output of each and explain your reasoning

```
# locals_shadow.py:
avar = 1

def afunc():
  avar = 5
  print("avar local:",avar)
  avar += 1
  print("avar local:",avar)

afunc()
print("avar global:",avar)
```

```
# globscope_fail.py:
theglob = 1                          # global variable

def print_theglob():                 # print it
  print("theglob:",theglob)

def inc_theglob():                   # increment it (??)
  theglob += 1

print_theglob()                      # print
inc_theglob()                        # increment
print_theglob()                      # print
```

Answers: Standard Scoping Rules

Python has slightly weird variable scoping rules

```
# locals_shadow.py:
avar = 1

def afunc():
  avar = 5
  print("avar local:",avar)
  avar += 1
  print("avar local:",avar)

afunc()
print("avar global:",avar)

# shell>> python locals_shadow.py
# avar local: 5
# avar local: 6
# avar global: 1
```

```
# globscope_fail.py:
theglob = 1                     # global variable

def print_theglob():            # print it
  print("theglob:",theglob)

def inc_theglob():              # increment it (??)
  # global theglob                # uncomment to fix
  theglob += 1

print_theglob()                 # print
inc_theglob()                   # increment
print_theglob()                 # print

# shell>> python globscope_fail.py
# theglob: 1
# Traceback (most recent call last):
#   File "globscope_fail.py", line 16, in <module>
#     inc_theglob()                   # increment
#     ^^^^^^^^^^^^
#   File "globscope_fail.py", line 13, in inc_theglob
#     theglob += 1
#     ^^^^^^^
# UnboundLocalError: cannot access local variable
# 'theglob' where it is not associated with a value
```

Object Oriented Programming (OOP) Support

- Python is object oriented with support for defining classes
- Weirdly, its syntax is somewhat awkward compared to Java and other OO languages
  - Opts object arg of methods explicit as `self`
  - Uses funky `__names__` for constructors, to-string
  - No declaration of fields
- Regrettable, but not matter
- Also interesting. . .
  - All fields of objects public, no `private` members
  - Objects are open: can have fields added dynamically

```
# oo_demo.py: demo of classes in python
class MyClass:
    i = 12345 # field / instance var

    # constructor
    def __init__(self,first_i):
        self.i = first_i

    # method
    def f(self):
        return 'hello world'
```

```
>>> import oo_demo
>>> mc = oo_demo.MyClass(2)
>>> mc.i
2
>>> from oo_demo import *
>>> mc5 = MyClass(7)
>>> mc5.i
7
>>> mc5.j = 8
>>> mc5.j
8
>>> print(mc5)
<oo_demo.MyClass object at 0x7f98eca99f50
```

Tour of `oo_collatz.py`

- Provided file which demos a semi-interesting class
- Makes the Collatz sequence computation more OO-like and demos many of the features we discussed
- Examine: `ool_collatz.py`