Going From C++ to Python

Aaron Kaloti, VP of CS Tutoring Club in Winter 2019

March 5, 2019

Contents

1 Introduction

This is a guide to introductory Python 3 intended for those with a C++ background. It reviews the differences between C++ and Python 3 in introductory concepts such as conditional statements, strings, and functions. It is intended to make our UC Davis Computer Science tutors more comfortable tutoring the introductory Python courses – ECS 32A, 32B, and 36A – newly offered by UC Davis, as our school has shifted from teaching C/C++ at the introductory level to teaching Python.

Want a short version of the major differences? Besides syntactic differences, look at: // operator, ** operator, string immutability, range-based for loops, tuples, and dictionaries.

2 (Quickly) Setting Up Python

Don't have Python 3? Here are two solutions:

1. Python 3 is already on the CSIF.
2. Download Python3 from here. You may wish to download Python IDLE, to have a GUI (if you don't prefer using the terminal).

3 Running a Python Program

- Run a Python program like so, using the 'python3' command. (If you're using IDLE, then do Run Module.) **Python is an interpreted language – no compiler needed.**

```
aaron123@ad3.ucdavis.edu@pc25:~$ cat hello-world.py
    def do_stuff():
        print("I did stuff")
    # Call the function we just defined.
    do_stuff()
aaron123@ad3.ucdavis.edu@pc25:~$ python3 hello-world.py
I did stuff
aaron123@ad3.ucdavis.edu@pc25:~$
```

  - Note that we don't do 'python hello-world.py', as on the CSIF, 'python' would run Python 2 instead of Python 3.

- Use Interpreter Mode to try out things in Python.

```
aaron123@ad3.ucdavis.edu@pc25:~$ python3
Python 3.6.7 (default, Oct 22 2018, 11:32:17)
[GCC 8.2.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> a = 3
>>> a
3
>>> b = a
>>> b
3
>>> quit()
aaron123@ad3.ucdavis.edu@pc25:~$
```

4 General Differences Between C++ and Python

In Python:

- Lines don't end in semi-colons.
- Types of variables and function parameters aren't explicitly specified. If you do an operation/function on the wrong type of variable, you will get a runtime error.
- Python has automatic garbage collection and thus will "free" your no-longer-needed variables for you, so there's no malloc()/free() or new/delete.
- Comments are indicated by # instead of //.
- "import" instead of "#include".

5 Numbers and Arithmetic

- +, -, *, and % are the same.
- Unlike in C++, / doesn't truncate in Python when both operands are integers. You must use // to cause truncation.

```
>>> 3 / 2
1.5
>>> 3 // 2
1
>>>
```

- Use ** for exponentiation.

```
>>> 5 ** 2  # 5 squared
25
```

- Prefix (++a) and postfix (a++) increments don't exist. Use a += 1 instead. (Note: Unexpectedly, ++a is a valid operation but DOES NOTHING.)

6 Standard Input/Output

- Use print() to print to standard output.
  - For fans of C++'s printf():

```
>>> print("{} says {}".format("Aaron","hi"))
Aaron says hi
```

- Use input() for basic standard input.

```
>>> name = input("Enter your name: ")
Enter your name: Aaron
>>> name
'Aaron'
>>>
```

7 Lists

- Lists in Python are arrays in C++, but you needn't do any special allocation stuff. The major list operations are demonstrated below.
- Define a list.

```
>>> mylist = ['a','b','c','d','e']  # list of characters
```

- Access element of a list.

```
>>> mylist[0]  # Python uses zero-based indexing like C++
'a'
>>> mylist[4]
'e'
```

- Access element of a list, starting from the back.

```
>>> mylist[-1]  # get first element from back
'e'
>>> mylist[-2]
'd'
```

- Get length of a list.

```
>>> len(mylist)  # get length of list
5
```

- Splice a sublist from the list.

```
>>> mylist[0:2]  # splice from index 0 to before index 2
['a', 'b']
>>> mylist[1:4]  # splice from index 1 to before index 4
['b', 'c', 'd']
>>> mylist[2:]  # splice from index 2 to end
['c', 'd', 'e']
>>> mylist[-2:]  # splice from second-to-last element onwards
['d', 'e']
```

- Concatenate two lists.

```
>>> ['t','u','v'] + ['w','x']  # concatenation
['t', 'u', 'v', 'w', 'x']
```

- Modify a specific list element.

```
>>> mylist[2] = 'x'  # modification: replace 'c' with 'x'
>>> mylist
['a', 'b', 'x', 'd', 'e']
```

- Append an element to a list.

```
>>> mylist.append('z')  # append 'z' to back of list
>>> mylist
['a', 'b', 'x', 'd', 'e', 'z']
```

- Delete an element from the list.

```
>>> del mylist[1]  # delete 'b' from list
>>> mylist
['a', 'x', 'd', 'e', 'z']
```

- Get the type of this list.

```
>>> type(mylist)
<class 'list'>
```

- You may hear a Python list be called a "sequence data type", since it supports indexing.

8 Strings

- Strings in Python are strings in C++, but Python has no characters (a character is a string of length 1).
- No difference between single quote and double quote.

- Ignoring modification operations, strings and lists have the same operations.
- Define a string.

```
>>> mystr = "aaron kaloti"
```

- Access element of a string.

```
>>> mystr[0]  # again, zero-based indexing
'a'
>>> mystr[-2]  # second-to-last character
't'
```

- Get length of a string.

```
>>> len(mystr)
12
```

- Splice a substring from the string.

```
>>> mystr[:5]  # splice to get my first name
'aaron'
>>> mystr[6:]  # splice to get my last name
'kaloti'
```

- Concatenate two strings.

```
>>> "concat" + "enation"
'concatenation'
```

- Get the type of this string.

```
>>> type(mystr)
<class 'str'>
```

- IMPORTANT: In Python, we call strings "immutable". This means that, unlike with a list, **you can't modify an individual element in a string**. You must instead use concatenation to create a new string.

```
>>> mystr
'aaron kaloti'
>>> mystr[1] = 'd'
>>> mystr[3] = 'i'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
>>> mystr = mystr[:3] + 'i' + mystr[4:]
>>> mystr
'aarin kaloti'
```

- You may hear a Python string be called a "sequence data type", since it supports indexing.

9 Conditional Statements

- If/else statements are the same, besides syntactic differences (no parentheses around the condition, condition ends with a colon, indentation indicates the body of the if/else, and we use "elif" instead of "else if"):

```
>>> x = 8
>>> if x < 0:
...     print('x is negative')
... elif x == 0:
...     print('x is zero')
... else:
...     print('x is positive')
...
x is positive
>>>
```

- "and" instead of &&. "or" instead of ||. For negation, statements like "a != b" still work, but "not a == b" is allowed too.

10 Iteration

- While loops are the same, besides minor syntactic differences:

```
>>> mylist = ['dog','cat','mouse']
>>> i = 0
>>> while i < len(mylist):
...     print(mylist[i])
...     i += 1
...
dog
cat
mouse
>>>
```

- Range-based for loop: for loop to iterate across a range of values (here, the variable **i** needn't be initialized prior):

```
>>> for i in range(2,6): # last value (6) isn't included
...     print(i)
...
2
3
```

```
4
5
>>> for i in range(3):  # range starts at 0 if only give one value
...     print(i)
...
0
1
2
>>>
```

- For loop to iterate across the values in a list:

```
>>> people = ['Aaron','Aakash','Matthew']
>>> for person in people:
...     print(person)
...
Aaron
Aakash
Matthew
>>>
```

  - Note that when using this syntax, we can't change the values in the list.

```
>>> people = ['Aaron','Aakash','Matthew']
>>> for person in people:
...     person = "Alex"
...
>>> people  # note that the list is unaffected
['Aaron', 'Aakash', 'Matthew']
>>>
```

- **break** and **continue** work the same.

11 Functions

- Types of function parameters aren't specified.
- A return type can't be specified in Python, so a function can return different types of values (or no value at all).
- Here is an example to illustrate syntactic differences:

```
>>> def isEven(val):
...     if val % 2 == 0:
...             return True
...     else:
```

```
...             return False
...
>>> isEven(3)
False
>>> isEven(4)
True
>>>
```

- Default argument values:

```
>>> def returnInput(val=8):
...     return val
...
>>> returnInput(3)
3
>>> returnInput()  # use default argument
8
>>>
```

12 Tuples

- A tuple is basically a list, except each individual element is immutable.
- Define a tuple.

```
>>> t = (8,5.3,'blah')
>>> t
(8, 5.3, 'blah')
>>> t = 8,5.3,'blah'  # you might see it without the parentheses
>>> t
(8, 5.3, 'blah')
>>>
```

- Access element in a tuple.

```
>>> t[1]
5.3
>>> t[-1]
'blah'
```

- Get length of a tuple.

```
>>> len(t)
3
```

- Splice from a tuple.

```
>>> t[1:]
(5.3, 'blah')
```

- Concatenate two tuples.

```
>>> ('part',1) + ('part',2)
('part', 1, 'part', 2)
```

- Get the type of this tuple.

```
>>> type(t)
<class 'tuple'>
```

- IMPORTANT: In Python, we call tuples "immutable", so – as with strings – **you can't modify an individual element in a tuple**. You must instead create a new tuple with concatenation.

```
>>> t[2] = 'nah'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
>>>
```

- You may hear a Python tuple be called a "sequence data type", since it supports indexing.

13 Sets (might not be covered in ECS 32A, 32B, or 36A)

- A set is a collection/container that **ignores duplicates**.
- Define a set with curly braces. Note the removal of the "Billy" duplicate.

```
>>> people = {"Bob", "Billy", "Blake", "Billy"}
>>> people
{'Bob', 'Billy', 'Blake'}
```

- A set is *not* a sequence data type and thus **cannot be indexed**.

```
>>> people[2]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'set' object does not support indexing
```

- However, membership testing can be performed.

```
>>> 'Bob' in people
True
>>> 'Ryan' in people
False
```

- **For those curious about standard library implementations**, note that **C++'s std::set is not akin to Python's set**. C++'s std::set sorts its elements and is implemented by a self-balancing binary search tree (a red-black tree, I believe), but Python's set does *not* sort its elements and is implemented by a hash table. Thus, **Python's set is akin to C++'s std::unordered_set**.

14 Dictionaries

- Dictionaries are indexed by key and return a value.

```
>>> d = {'apple': 33, 'teehee': 21}
>>> d
{'apple': 33, 'teehee': 21}
>>> d['apple']
33
```

- Append to a dictionary like so (note that keys and values needn't be consistent types).

```
>>> d['blah'] = 'hi'
>>> d[4] = 8
>>> d
{'apple': 33, 'teehee': 21, 'blah': 'hi', 4: 8}
>>>
```

- Get length:

```
>>> len(d)
4
```

- Delete an element:

```
>>> del d['teehee']
>>> d
{'apple': 33, 'blah': 'hi', 4: 8}
```

- One way to iterate through a dictionary:

```
>>> for k in d:
...     print(k,d[k])
...
```

```
apple 33
blah hi
4 8
```

- Another way to iterate through a dictionary:

```
>>> for k,v in d.items():
...     print(k,v)
...
apple 33
blah hi
4 8
```

15 File Input/Output

- Opening a file, reading all of it, and closing it:

```
aaron123@ad3.ucdavis.edu@pc25:~$ cat poem.txt
There once was a man from Peru
Who dreamed he was eating his shoe.
He woke up with a fright
In the middle of the night
To find that his dream had come true.
aaron123@ad3.ucdavis.edu@pc25:~$ python3
>>> fr = open('poem.txt', 'r')
>>> fr.read()  # read entire file
'There once was a man from Peru\nWho dreamed he was eating his shoe.\nHe
>>> fr.readline()  # nothing left to read
''
>>> fr.close()
```

- Can read a file line by line.

```
>>> fr = open('poem.txt', 'r')
>>> fr.readline()
'There once was a man from Peru\n'
>>> fr.readline()
'Who dreamed he was eating his shoe.\n'
```

- Writing to a file:

```
>>> fw = open('poem.txt', 'w')  # clears file's contents
>>> fw.write("nah\n")  # returns number of characters written
4
>>> fw.close()
>>> quit()
```

```
aaron123@ad3.ucdavis.edu@pc25:~$ cat poem.txt
nah
aaron123@ad3.ucdavis.edu@pc25:~$
```

- Use 'r+', NOT 'rw', to open a file for reading and writing.

16 Command-line arguments

- Minor syntactic differences, demonstrated below. Python has no argc; use len(sys.argv) instead.

```
aaron123@ad3.ucdavis.edu@pc25:~$ cat print-args.py
import sys
print("Arguments: ", sys.argv)
print("Number of arguments: ", len(sys.argv))
aaron123@ad3.ucdavis.edu@pc25:~$ python3 print-args.py aaron kaloti
Arguments:  ['print-args.py', 'aaron', 'kaloti']
Number of arguments:  3
aaron123@ad3.ucdavis.edu@pc25:~$ python3 print-args.py
Arguments:  ['print-args.py']
Number of arguments:  1
aaron123@ad3.ucdavis.edu@pc25:~$
```

17 Exceptions

- Minor syntactic differences, demonstrated below (example's source).

```
>>> while True:
...     try:
...         x = int(input("Please enter a number: "))
...         break
...     except ValueError:
...         print("Oops!  That was no valid number.
Try again...")
...
Please enter a number: abc
Oops!  That was no valid number.  Try again...
Please enter a number: 5
>>>
```

18 Basic User-Defined Classes (for small part of ECS 32B)

- NOTE: User-defined classes shouldn't come up in ECS 32A or 36A, but they do come up briefly in Kurt Eiselt's ECS 32B. Kurt doesn't cover inheritance, but if a future ECS 32B instructor does, I'll add content about inheritance to this guide.
- In Python, we use "self" instead of "this". Python's "self" is always required, unlike C++'s "this", which is implicit in a lot of cases.
- Unlike C++, Python doesn't truly support private members (just ways to make "private" members harder to access, but I won't get into that here.)
- Here is a sample user-defined class:

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def get_age(self):
        return self.age

    def set_age(self, new_age):
        self.age = new_age
```

- Let's play with this class:

```
aaron123@ad3.ucdavis.edu@pc25:~$ python3
>>> from sample_class import Person
>>> a = Person("Bob", 34)
>>> a.get_age()
34
>>> a.set_age(88)
>>> a.age  # no privacy
88
>>> a.name  # no privacy
'Bob'
```

19 Other Brief Tutorials

- W3Schools' new Python tutorial, provided here.
- Tutorial in the Python 3 documentation, although I find it too extensive (as it was intended to be), provided here.