Python Preliminaries

CMPT 732, Fall 2022

We will assume the basics of Python.

This will be a few notes about things (useful in Spark programming) that many (beginner?) Python programmers don't know.

About Python

A high-level programming language. Very commonly used for data science work.

We'll be using Python 3 in this course (which has some minor incompatibilities with Python 2).

Python isn't noted for being fast: it seems like an odd choice for Big Data. (More on that later.)

Python **is** compiled. The compilation (to Python bytecode) happens as you start the program, not as a separate explicit step (like Java to Java bytecode).

It is a beautiful language to write and to read.

Data Types

Python has the basic data types you'd expect: booleans, integers, floating point values, strings.

Strings in Python 3 are Unicode strings that hold **characters**:

```
>>> s = 'ℬⅈℊ DÅ\u27d9\U0001D4D0'
>>> s
'ℬⅈℊ DÅ⟙𝓐'
>>> len(s)
8
>>> s.encode('utf-8')
b'\xe2\x84\xac\xe2\x85\x88\xe2\x84\x8a D\xc3\x85\xe2\x9f\x99\xf0\x9d\x93\x90'
>>> s.encode('utf-16')
b"\xff\xfe,!H!\n! \x00D\x00\xc5\x00\xd9'5\xd8\xd0\xdc"
```

Byte strings hold **bytes**. Character ≠ byte!

Python ***lists*** and ***tuples*** both hold ordered collections of values. By convention: lists hold data sets, tuples hold fixed-length structures. e.g.

- Bunch of lengths/view counts/temperatures: list.
- Function returning multiple results: tuple.
- Bunch of latitude/longitude values: list of tuples.

```
[(49.2, -123.0), (49.3, -123.1), (49.7, -122.8)]
```

Python ***dictionaries*** are maps/hash tables/associative arrays.

```
>>> num_words = { 2: 'two', 3: 'three', 4: 'four' }
>>> num_words[5] = 'five'
>>> num_words[3]
'three'
```

Unpacking Tuples

We will often have tuples with several values (pairs and more). There are built-in ways to unpack them:

```
val = (1, (2, 3))
a, p = val
assert a==1 and p==(2,3)
a, (b, c) = val
assert a==1 and b==2 and c==3
```

Compare:

```
a, (b, c) = val
result = a + b + c
```

vs

```
result = val[0] + val[1][0] + val[1][1]
```

If you write (the equivalent of) the second, I'm going to refuse to decode it if you need help.

Unpacking in function arguments was possible in Python 2, but no longer works:

```
def add_weird((a, (b, c)), mult=1): # Python 2 only
    return mult*(a+b+c)
val = (1, (2, 3))
res = add_weird(val, 3)
assert res==18
```

```
def add_weird(abc, mult=1):
    a, (b,c) = abc
    return mult*(a+b+c)
val = (1, (2, 3))
res = add_weird2(val, 3)
assert res==18
```

First-Class Functions

Functions are first-class values in Python. They can be assigned, passed, returned: everything you can do with any other value.

```
def double(x):
    return x+x
def apply_twice(f, x):
    return f(f(x))

assert apply_twice(double, 5) == 20
dbl = double
assert apply_twice(dbl, 5) == 20
```

We will be using this a lot: many of the Spark operation will be in the form “apply function `f` to all values (in parallel, we hope) and do *something* with the result.”

```
def double(x):
    return x+x

doubled_values = some_values.map(double)
```

Lambda Functions

The standard way to define functions in Python:

```
def pair_with_one(w):
    return (w, 1)

def add(a, b):
    return a + b
```

We will often need simple functions like these to do a step in our calculation.

... sometimes the logic will be so simple that defining a named function seems like a waste of time.

There is a ***lambda function*** or ***lambda expression*** syntax to define an anonymous function inline.

[If you have seen anonymous/unnamed/lambda functions in JavaScript or another language, it's the same idea.]

These are equivalent Python:

```
def add(a, b):
    return a + b

some_key_value_pairs.reduceByKey(add)
```

```
some_key_value_pairs.reduceByKey(lambda a, b: a + b)
```

The `lambda`: a function that takes the named arguments, and returns the result of the expression.

And actually, so is this, but it's odd style:

```
add = lambda a, b: a + b
some_key_value_pairs.reduceByKey(add)
```

... and in this case, there's a built-in version:

```
import operator
some_key_value_pairs.reduceByKey(operator.add)
```

Nothing annoys me more than:

```
some_data.map(lambda x: f(x))
```

That lambda is a function that takes `x` and returns `f(x)`. We already have a way to say that: `f`.

```
some_data.map(f)
```

It's easy to write hard-to-read lambda expressions.

```
some_key_value_pairs.map(lambda kv: kv[0][0])
```

Unless it's **extremely** simple, just write a named function (and use the name as free documentation).

```
def first_char_of_key(key_value):
    key, val = key_value
    return key[0]

some_key_value_pairs.map(first_char_of_key)
```

A lot of Spark tutorials make heavy use of lambda expressions. [For simple examples, they are easier to fit on a slide.]

Don't fall into the trap of using them too much (or thinking you *must* use lambda expressions for Spark calls).

Named functions can contain more logic, and the names provide free explanation of what they do. Lambdas are shorter to write for simple logic.

My rule: If the function can be understood in 1 second, it can be a lambda. Otherwise, name it. It's never *wrong* to use a regular named function.

Iterators and Generators

The `for` loop and other things iterate over collections, often lists. Don't forget that they can also work on arbitrary iterable things. e.g. in Python 3, `range` returns an iterable:

```
>>> r = range(10**18)
>>> len(r)
1000000000000000000
>>> type(r)
<class 'range'>
```

That didn't run out of memory.

These create lists and use lots of memory:

```
def squares(n):
    res = []
    for i in range(n):
        res.append(i*i)
    return res
ssq1 = sum(squares(100000))
ssq2 = sum([i*i for i in range(100000)])
```

Here, `squares` creates a big list and returns it; the list comprehension (`[…for…in…]`) also creates a list in memory.

These create generator objects and use little:

```
def squares(n):
    for i in range(n):
        yield i*i
ssq3 = sum(squares(100000))
ssq4 = sum(i*i for i in range(100000))
```

A Python function that `yield`s is conceptually returning a sequence of values, but only generating them as they are consumed (as a coroutine).

A generator expression does the same: creates many values, but only as they are consumed. They are never stored as a list in memory.

Imperative vs declarative

You are likely most used to *imperative programming* where the programmer expresses the steps a program should take to complete a task. (e.g. in C/Java/C# and in Python)

```
total = 0;
for ( i=0; i<n; i++ ) {
  total += lst[i];
}
```

```
total = 0
for i in range(n):
    total += lst[i]
```

On the other hand, in *declarative programming* the programmer expresses *what* is being calculated, but not the exact steps to do calculate it.

A place you have likely done declarative things: SQL.

```
SELECT SUM(val) FROM values;
```

Or maybe in higher-level tools in imperative languages.

```
import numpy
arr = numpy.random.randn(1000)
arr.sum()
```

In a lot of ways, MapReduce is quite declarative: you specify the things that need to be calculated, and trust the framework to get it done.

You *can* see all of the imperative details of MapReduce, but may be happiest (“say about 97% of the time”) if you just accept that the things you ask for will be calculated, and don't worry about how.

The same will be true with Spark: we will make very high-level requests for calculations and let Spark figure out how to calculate it.

Saying higher-level things lets a compiler/optimizer be clever on our behalf.

That's *usually* good, and we can ignore the details. Except when we can't.

Joel's Law of Leaky Abstractions