Python programming and data analysis

Lecture 2

Crash course continued 2

Robert Szmurło

e-mail: robert.szmurlo@ee.pw.edu.pl 2020L

Lecture outline

- Quick review of key elements of previous lecture
- List comprehensions (syntactic sugar only?)
- Generators (simple + prime number)
- Variable number of arguments (tuple unpacking)
- Zipping, Un-zipping, enumerating, iter()
- Object oriented Programming
  - simple classes (constructor, class and instance attributes)
  - basic inheritance (**mro** - *ang. Method Resolution Order*)
- Working with text files
  - `open` + basic readline and while
  - `with` - keyword
- contextlib (supporting the `with` keyword)

Next lecture

- decorators
- metaprogramming

Quick review of 1st lecture

- Data types
  - Numbers
  - Strings
  - Printing (% notation, .format, Formatted String Literals (PEP 498 Python >= 3.6, sys.version))
  - Lists, Dictionaries
  - slicing notation (a[start:end:step], start and step can be negative)
  - Booleans
  - Tuples
  - Sets
- Comparison Operators
  - if, elif, else Statements
- for Loops
- while Loops
- range()
- list comprehension
- functions
- lambda expressions
- map and filter functions

List comprehensions

List comprehensions provide a concise way to create lists with initialized values. Common applications are to make new lists where each element is the result of some operations applied to each member of another sequence or iterable, or to create a subsequence of those elements that satisfy a certain condition.

```
squares = [x**2 for x in range(10)]
```

```
>>> [(x, y) for x in [1,2,3] for y in [3,1,4] if x != y]
[(1, 3), (1, 4), (2, 3), (2, 1), (2, 4), (3, 1), (3, 4)]
```

- can be used to filtering values

Review:

- `map(expr,items)` - function which allows to process items in a list with some expression
- `filter(cond,items)` - function allowing to filter items with an expression evaluating to True/Fals for each item

But these are working on existing collections. The list comprehension can do both - create a new collection, filter and process in a single step.

Python generators

Generators are structures which allow to generate and iterate over sequences of values. The main advantage is that the generator behaves likes an iterated list but does not need memory to keep the list contents. The generators return next values on the fly - when the **next** magic function is called.

1. Traditional function generating a vector (list) of values example (This is not yet a generator, just a consuming memory list!).
2. Inline vector values generator (real list) - squared braces
3. Inline generator of values - braces (generator expressions)
4. Function generator - yield keyword

Example of inline generator:

```python
# You can iterate over a generator only once.

v = (1/(i+1) for i in range(n) )
print(v)
print(v) # no problem yet :-)

print("First run:")
for i in v:
    print(i)

print("Second run:")
for i in v:
    print(i)

print("done")
```

Function generators - `yield` keyword example

Function genrators a simply function which when called instead of a single result return a generator, which can then be iterated.

```python
def gen_vals():
    print("first instr")
    yield 1
    print("second instr")
    yield 2
    yield 3
    yield 4

g = gen_vals()
print("After created")
print(next(g))
print(" After first next")
```

Classical class generators

```python
# Class generator

class GenValsClass(object):
    def __init__(self, n):
        self.n = n;
        self._i = 0;

    def __iter__(self):
        return self

    def __next__(self):
        return self._nextval()

    def _nextval(self):
        #print(f"next {self._i}")
        if (self._i <= self.n):
            val = 1 / (self._i +1)
            self._i += 1
            return val
        else:
            raise StopIteration()

vc = GenValsClass(n)
#print(list(vc))
for i in vc:
    print(i)

vc = GenValsClass(n)

v = next(vc, None)
while(v is not None):
    print(v)
    v = next(vc, None)
```

Ohhh, noooo... this so long and hard to follow code comparing to the same function generator!

Generators recap

Why generators are used in Python?

1. They are easy to implement and read
2. Memory efficient
3. Represent infinite stream (we iterate that many times as we need)
4. Pipelining generators (memory efficient)

Pipelining example:

Assume that the log file contains in the 4th column the number of students enrolled to the course. We want to count all enrollments.

```python
with open('isod_logins.log') as file:
    count_col = (line[3] for line in file)
    count_col_nums = (int(x) for x in count_col if x != 'N/A')
    print("Total students enrolled = ",sum(count_col_nums))
```

Variable number of arguments

```python
# variable number of arguments to functions

def fun(required_arg):
    print(required_arg)

def fun1(not_required_arg='default'):
    print(not_required_arg)

def fun2(required_arg, not_required_arg='default'):
    print(required_arg)
    print(not_required_arg)

fun2("d")

def fun3(required_arg, not_required_arg='default',  *args, **kwargs):
    print(required_arg)
    print(not_required_arg)
    for i,a in enumerate(args):
        print(f"[{i}] = {a}")
    for k,v in kwargs.items():
        print(f"{k} = {v}")

fun3("d", "f", some_argument="bla", any_other_sytupid="poaaaa")

# The function below should fail
#fun3("d", "f", some_argument="bla", not_required_arg="akuku")

def fun4(required_arg,  *args, not_required_arg='default', **kwargs):
    print(required_arg)
    print(not_required_arg)
    for i,a in enumerate(args):
        print(f"[{i}] = {a}")
    for k,v in kwargs.items():
        print(f"{k} = {v}")

fun4("d", "f", some_argument="bla", not_required_arg="akuku")
```

Zipping, unzipping, unpacking Tuples

This a *pythonic* style of programming. Thep purpose: match list by the same indexes into `Tuples`. We have to separate lists, but the items in both of them correspond to each other.

```
names = ['Staś', 'Jasia', 'Zbyszek', 'Jusia']
ages = [6,4,2,0]

z = zip(names, ages)
l = list(z)
unpacked_names, unpacked_ages = zip(*l)
print(unpacked)
print(lists)
```

Classes and basic object oriented programming

- type() method
- class Dog(object): …
- **init**(self)
- class scoped and instance scoped attributes
- basic inheritance (Python supports multiple)
- constructing classes with type(method)

Example of a class with *magic methods*

```
class MyClass:
    def __init__(self):
      # there is no need to implement this :-), it is here just to show the syntax
      pass

    def whoami(self):
        print("Robert")

    def __len__(self):
        return 10

    def __str__(self):
        return "Robert"

    def __repr__(self):
        return "MaClass: Robert"

obj = MyClass()
obj.whoami()
len(obj)
```

Reading files

- simple open and iterating over lines
- for iterating with iter()
- with structure

Just to start

```
f = open("data.txt", "w")
f.write("col1, col2,col3\n1 2 3\n4 5 6\n7 8 9")
f.close()

v = []
with open("data.txt") as f:
    f.readline()
    for line in f.readlines():
        row = []
        for col in line.split():
            row.append(int(col))
        v.append(row)

print(v)
```

Custom objects for `with`

```python
class MyCtx(object):
  def __init__(self, gender):
    self.gender = gender

  def __enter__(self):
    print("Entering")
    return self

  def __exit__(self, exc_type, exc_value, traceback):
    print("Leaving")

with MyCtx("male") as boy:
  print(boy.gender)
```

Next time: when the decorators contextlib

This module provides utilities for common tasks involving the with statement.

Prime numbers

Additional example:

```
%%timeit -n 100
def isprime(x):
  d = 2
  #r = int(x**0.5)
  r = x
  while d <= r:
    if x % d == 0:
      return False
    d += 1
  return True

#infgen = (x for x in (y for y in range(1000)) if isprime(x))
infgen = (x for x in filter(isprime,(y for y in range(1000))))
for i,n in enumerate(infgen):
  #print(f"Prime number: {n}")
  if i>40:
    break
```

Thank you