pragmatic_introduction_to_python_language_1

March 4, 2021

1 A Pragmatic Introduction to Python

1.1 What is Python?

1.1.1 The Python Interpreter

- Python is an interpreted language
- The interpreter provides an interactive environment to play with the language

1.1.2 No Braces, only Spaces!

- Python uses indentation instead of braces to determine the scope of expressions
- All lines must be indented the same amount to be part of the scope (or indented more if part of an inner scope)
- This forces the programmer to use proper indentation since the indenting is part of the program!

1.2 Understanding Basic String Functions

1.2.1 What is a String?

A string is a basic unit of interaction between the world of computers and the world of humans. Initially, almost all raw data is stored as strings.

The case conversion functions return a copy of the original string s: `lower()` converts all characters to lowercase; `upper()` converts all characters to uppercase; and `capitalize()` converts the first character to uppercase and all other characters to lowercase. These functions don't affect non-alphabetic characters. Case conversion functions are an important element of normalization.

[1]:
```
message = 'The clock strikes at midnight!'
print(message)
```

```
The clock strikes at midnight!
```

[2]:
```
message = 'I am looking for someone to share in an adventure that I am␣
 ↪arranging, and it's very difficult to find anyone.'
```

```
  File "<ipython-input-2-a11d809302bd>", line 1
```

```
        message = 'I am looking for someone to share in an adventure that I am␣
↪arranging, and it's very difficult to find anyone.'
                                                                              ␣
↪                 ^
    SyntaxError: invalid syntax
```

```
[3]: message = 'I am looking for someone to share in an adventure that I am␣
      ↪arranging, and it\'s very difficult to find anyone.'
```

```
[4]: message = "I am looking for someone to share in an adventure that I am␣
      ↪arranging, and it's very difficult to find anyone."
     print(message)
```

```
I am looking for someone to share in an adventure that I am arranging, and it's
very difficult to find anyone.
```

```
[5]: message = 'WHAT is A string?'
     print(message)
     print(message.lower(), message.upper(), message.capitalize())
```

```
WHAT is A string?
what is a string? WHAT IS A STRING? What is a string?
```

In Python `lower()`, `upper()` and `capitalize()` are examples of **methods**. But what is a **method**?

> Before we get any farther into the Python language, we have to say a word about *objects*. We will not be teaching object oriented programming in this introduction, but you will encounter objects throughout Python (in fact, even seemingly simple things like ints and strings are actually objects in Python).
>
> An object is like a **bundled "thing" that contains within itself both data and functions that operate on that data**. For example, strings in Python are objects that contain a set of characters and also various functions that operate on the set of characters. When bundled in an object, these functions are called "methods".

Would you like to know more about **markdown** syntax? Click here.

Instead of the "normal" function(arguments) syntax, methods are called using the syntax `variable.method(arguments)`.

The *predicate functions* return `True` or `False`, depending on whether the string s belongs to the appropriate class: `islower()` checks if all alphabetic characters are in lowercase; `isupper()` checks if all alphabetic characters are in uppercase; `isspace()` checks if all characters are spaces; isdigit() checks if all characters are decimal digits in the range 0–9; and `isalpha()`checks if all characters are alphabetic characters in the ranges a–z or A–Z. You will use these functions to recognize valid words, nonnegative integer numbers, punctuation, and the like.

[6]:
```
message = 'THIS IS A STRING'

if(message.isupper()):
    print('The string is an uppercase string')
```

```
The string is an uppercase string
```

Conditional Statements Python **if Statement** is used for decision-making operations. It contains a body of code which runs only when the condition given in the if statement is true. If the condition is false, then the optional else statement runs which contains some code for the else condition.

Python if Statement Syntax:

```
if expression:
 Statement
else:
 Statement
```

[7]:
```
message = 'THIS is a string'

if(message.isupper()):
    print('The string is an uppercase string')
elif(message.islower()):
    print('The string is a lowercase string')
else:
    print('The string has not a definite case')
```

```
The string has not a definite case
```

Conditional Operators An if statement is known as a control structure because it controls whether the code in the indented block will be run.

Python supports a wide range of operators, such as < and >=, for testing the relationship between values. The full set of these relational operators are shown in the table below

Operator	Relationship
<	Less than
<=	Less than or equal to
==	Equal to (note this is two "=" signs not one!)
!=	Not equal to
>	Greater than
>=	Greater than or equal to

1.2.2 String Processing

The first step toward string processing is getting rid of unwanted whitespaces (including new lines and tabs). The functions `lstrip()` (left strip), `rstrip()` (right strip), and `strip()` remove all whitespaces at the beginning, at the end, or all around the string. (They don't remove the inner spaces.)

```
[8]: message = "       Hello world!     "
     print(message.strip())
```

```
Hello world!
```

```
[9]: print(message)
```

```
       Hello world!
```

```
[10]: message = message.strip()
      print(message)
```

```
Hello world!
```

The + Operator The `+` operator concatenates strings. It returns a string consisting of the operands joined together, as shown here:

```
[11]: "abc" + "def"
```

```
[11]: 'abcdef'
```

It should be underlined that a ‘string’ is a sequence of characters (letters, digits, spaces, punctuation, new lines, etc.)

For Loop Another control structure is the `for` loop. Try the following:

```
[12]: for c in message:
          if c != ' ':
              print(c)
```

```
H
e
l
l
o
w
o
r
l
d
!
```

The function print put a “new line” character by default at the end of the printed string. If we want to change this predefined behaviour, we can explicitly specify the character to use at the end, or we can decide to use no character at all like in the following example:

[13]:
```python
for c in message:
    if c != ' ':
        print(c, end='')
```

```
Helloworld!
```

> **Note on indentation:** Notice the indentation once we enter the for loop. Every idented statement after the for loop declaration is part of the for loop. This rule holds true for while loops, if statements, functions, etc. Required identation is one of the reasons Python is such a beautiful language to read.

If you do not have consistent indentation you will get an IndentationError. Fortunately, most code editors will ensure your indentation is correction.

> **NOTE** In Python the default is to use four (4) spaces for each indentation, most editros can be configured to follow this guide.

You will notice also that `if` and `for` statements have a colon at the end of the line, before the indentation begins. In fact, all Python control structures end with a colon. The colon indicates that the current statement relates to the indented block that follows.

[14]:
```python
wordlist = ['this','is','a','wonderful','python','tutorial']
# Often we want to loop over the indexes of a collection, not just the items
print(wordlist)

for i, word in enumerate(wordlist):
    print(i, word, wordlist[i])
```

```
['this', 'is', 'a', 'wonderful', 'python', 'tutorial']
0 this this
1 is is
2 a a
3 wonderful wonderful
4 python python
5 tutorial tutorial
```

[15]:
```python
# While loops are useful when you don't know how many steps you will need,
# and want to stop once a certain condition is met.
step = 0
prod = 1
while prod < 100:
    step = step + 1
    prod = prod * 2
    print(step, prod)

print('Reached a product of', prod, 'at step number', step)
```

```
1 2
2 4
3 8
```

```
4 16
5 32
6 64
7 128
Reached a product of 128 at step number 7
```

String template

```
[16]: template = "Contract number %s refers to %s"

      print(template % (112, 'Clark Kent'))
      print(template % (145, 'Bruce T. P. Wayne'))
```

```
Contract number 112 refers to Clark Kent
Contract number 145 refers to Bruce T. P. Wayne
```

Splitting Often a string consists of several tokens, separated by delimiters such as spaces, colons, and commas. The function `split(delim='')` splits the string s into a list of substrings, using `delim` as the delimiter. If the delimiter isn't specified, Python splits the string by all whitespaces and lumps all contiguous whitespaces together:

```
[17]: sentence = "One Ring to rule them all, One ring to find them, One ring to bring␣
       ↪them all and in the darkness bind them"
      print(sentence.split())
      print(sentence.split(sep=','))
```

```
['One', 'Ring', 'to', 'rule', 'them', 'all,', 'One', 'ring', 'to', 'find',
'them,', 'One', 'ring', 'to', 'bring', 'them', 'all', 'and', 'in', 'the',
'darkness', 'bind', 'them']
['One Ring to rule them all', ' One ring to find them', ' One ring to bring them
all and in the darkness bind them']
```

```
[18]: site  = 'www.networksciencelab.com'
      slist = site.split(sep='.')
      print(slist)
```

```
['www', 'networksciencelab', 'com']
```

```
[19]: print('1st item of list:')
      print(slist[0])
      print('2nd item of list:')
      print(slist[1])
      print('3rd item of list:')
      print(slist[2])
```

```
1st item of list:
www
2nd item of list:
networksciencelab
```

```
3rd item of list:
com
```

[20]:
```
print('1st item of list: ' + slist[0])
```

```
1st item of list: www
```

The sister function `join(ls)` joins a list of strings ls into one string, using the object string as the glue. You can recombine fragments with `join()`:

[21]:
```
''.join(slist)
```

[21]:
```
'wwwnetworksciencelabcom'
```

[22]:
```
' '.join(slist)
```

[22]:
```
'www networksciencelab com'
```

[23]:
```
'.'.join(slist)
```

[23]:
```
'www.networksciencelab.com'
```

[24]:
```
date = '2020-10-10'
date = date.split('-')
date = '/'.join(date)

print(date)
```

```
2020/10/10
```

[25]:
```
date = '2020-10-10'
print(date.replace('-','/'))
```

```
2020/10/10
```

[26]:
```
myString = 'This            string   has      so          many                ␣
 ↪spaces!'
myString = ' '.join(myString.split())
myString
```

[26]:
```
'This string has so many spaces!'
```

Searching Search a specific substring

[27]:
```
path = 'C:/folder1/folder2/folder3/filename.ext'
```

[28]:
```
print(path.find('ext'))
```

```
36
```

[29]:
```
print(path.count('/'))
```

```
4
```

[30]:
```
print(path.split('/'))
```

```
['C:', 'folder1', 'folder2', 'folder3', 'filename.ext']
```

1.3 Choosing the Right Data Structure

The most commonly used compound data structures in Python are:

- **lists**,
- **tuples**,
- **sets**, and
- **dictionaries**.

All four of them are collections.

1.3.1 Lists

Python implements lists as arrays. They have linear search time, which makes them impractical for storing large amounts of searchable data.

[31]:
```
# Lists are created with square bracket syntax
a = ['blueberry', 'strawberry', 'pineapple']
print(a, type(a))
```

```
['blueberry', 'strawberry', 'pineapple'] <class 'list'>
```

[32]:
```
a = [21, 32, 15, 34]
```

[33]:
```
len(a)
```

[33]:
```
4
```

[34]:
```
a[0]  # list indexing is zero-based
```

[34]:
```
21
```

[35]:
```
a[3]
```

[35]:
```
34
```

[36]:
```
for k in range(2, 10, 2):
  print(k)
```

```
2
4
```

```
6
8
```

[37]:
```python
for k in range(len(a)):
    print(a[k])
```

```
21
32
15
34
```

[38]:
```python
print(list(range(10)))
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Lists are *mutable*

[39]:
```python
a[2] = 42
a
```

[39]:
```
[21, 32, 42, 34]
```

[40]:
```python
a = list(range(100,120))
print(a)
```

```
[100, 101, 102, 103, 104, 105, 106, 107, 108, 109, 110, 111, 112, 113, 114, 115,
116, 117, 118, 119]
```

[41]:
```python
a[-1]
```

[41]:
```
119
```

[42]:
```python
a[5:] # elements up to but excluding 5th item
```

[42]:
```
[105, 106, 107, 108, 109, 110, 111, 112, 113, 114, 115, 116, 117, 118, 119]
```

[43]:
```python
a[:5] # elements starting from and including 5th item
```

[43]:
```
[100, 101, 102, 103, 104]
```

1.3.2 Tuples

Tuples create a bit of confusion for beginners because they're very similar to lists, but they have some subtle conceptual differences. Nonetheless, tuples do appear when programming in Python, so it's important to know about them.

Like lists, tuples are sequences of any type of object. **Unlike lists, they are immutable**. This means that: - Once constructed, they cannot be changed - i.e. you cannot append, insert or delete elements - Because they are immutable, they can be used as dictionary keys (lists cannot)

You declare tuples using () instead of []

```
[44]: a = (42, 34, 99)
      a[2]
```

```
[44]: 99
```

```
[45]: a[2] = 100
```

```
          ␣
    ↪---------------------------------------------------------------------------

           TypeError                                 Traceback (most recent call␣
    ↪last)

           <ipython-input-45-ee672155748c> in <module>
        ----> 1 a[2] = 100


           TypeError: 'tuple' object does not support item assignment
```

```
[46]: # another way of accessing the elements is to 'unpack' the tuple
      # this works with lists too, by the way
      x, y, z = (1, 2, 3)
      print(x)
      print(y)
      print(z)
```

```
1
2
3
```

```
[47]: a[2] = 87
```

```
          ␣
    ↪---------------------------------------------------------------------------

           TypeError                                 Traceback (most recent call␣
    ↪last)

           <ipython-input-47-520cdf962392> in <module>
        ----> 1 a[2] = 87


           TypeError: 'tuple' object does not support item assignment
```

In general, they're often used instead of lists: - to group items when the position in the collection is critical, such as coord = (x,y) - when you want to make prevent accidental modification of the items, e.g:

```
[48]: days = ('Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday', 'Saturday',␣
       ↪'Sunday')
```

1.3.3 Sets

Unlike lists and tuples, **sets** are not sequences: set items don't have indexes. Sets can store at most one copy of an item and have sublinear *O(log(N))* search time. They are excellent for membership look-ups and eliminating duplicates (if you convert a list with duplicates to a set, the duplicates are gone):

```
[49]: sentence = 'One Ring to rule them all, One ring to find them, One ring to bring␣
       ↪them all and in the darkness bind them'
      myList = sentence.split()
      print(myList)
      print(len(myList))
```

```
['One', 'Ring', 'to', 'rule', 'them', 'all,', 'One', 'ring', 'to', 'find',
'them,', 'One', 'ring', 'to', 'bring', 'them', 'all', 'and', 'in', 'the',
'darkness', 'bind', 'them']
23
```

```
[50]: mySet = set(myList)
      print(mySet)
      print(len(mySet))
```

```
{'darkness', 'the', 'them,', 'in', 'and', 'Ring', 'to', 'all,', 'find', 'them',
'ring', 'bring', 'rule', 'all', 'One', 'bind'}
16
```

```
[51]: print(sorted(mySet))
```

```
['One', 'Ring', 'all', 'all,', 'and', 'bind', 'bring', 'darkness', 'find', 'in',
'ring', 'rule', 'the', 'them', 'them,', 'to']
```

```
[55]: # creating a new list with lower case words from myList
      # first of all we define an empty list
      dummy = []
      for w in myList:
        # we use the metod append to increment items number of dummy
        dummy.append(w.lower())

      dummy
```

```
[55]: ['one',
        'ring',
        'to',
        'rule',
        'them',
        'all,',
        'one',
        'ring',
        'to',
        'find',
        'them,',
        'one',
        'ring',
        'to',
        'bring',
        'them',
        'all',
        'and',
        'in',
        'the',
        'darkness',
        'bind',
        'them']
```

```
[56]: # modify myList, in this case we can use the index
      for k, s in enumerate(myList):
        myList[k] = s.lower()

      myList
```

```
[56]: ['one',
        'ring',
        'to',
        'rule',
        'them',
        'all,',
        'one',
        'ring',
        'to',
        'find',
        'them,',
        'one',
        'ring',
        'to',
        'bring',
        'them',
        'all',
        'and',
        'in',
        'the',
        'darkness',
        'bind',
        'them']
```

```
[57]: myList = [x.lower() for x in myList]
```

```
[58]:  # what is this instruction? What does it mean?
      mySet  = set(myList)
      print(sorted(mySet))
```

```
['all', 'all,', 'and', 'bind', 'bring', 'darkness', 'find', 'in', 'one', 'ring',
'rule', 'the', 'them', 'them,', 'to']
```

Exercise - Find a way to remove the comma in final position of some words!

Answer

A simple solution is the following:

```
dummy=[]
for word in myList:
  if not ',' in word:
    dummy.append(word.lower())
```

```
[59]: dummy=[]
      for word in myList:
        if not ',' in word:
          dummy.append(word.lower())

      dummy
```

```
[59]: ['one',
       'ring',
       'to',
       'rule',
       'them',
       'one',
       'ring',
       'to',
       'find',
       'one',
       'ring',
       'to',
       'bring',
       'them',
       'all',
       'and',
       'in',
       'the',
       'darkness',
       'bind',
       'them']
```

[60]:
```
myList = [word.lower() for word in myList if not ',' in word]
mySet  = set(myList)
print(sorted(mySet))
```

```
['all', 'and', 'bind', 'bring', 'darkness', 'find', 'in', 'one', 'ring', 'rule',
'the', 'them', 'to']
```

[61]:
```
word = 'ring'
if word in mySet:
    print('word "' + word + '" found in mySet')
```

```
word "ring" found in mySet
```

1.3.4 Dictionaries

Dictionaries map keys to values. An object of data type number, Boolean, string or tuple can be a key, and different keys in the same dictionary can belong to different data types. There is no restriction on the data types of dictionary values.

Dictionaries are the collection to use when you want to store and retrieve things by their names (or some other kind of key) instead of by their position in the collection. A good example is a set of model parameters, each of which has a name and a value. Dictionaries are declared using {}.

You can create a dictionary from a list of `(key, value)` tuples, and you can use a built-in class constructor `enumerate(seq)` to create a dictionary where the key is the sequence number of an item in `seq`:

[62]:
```
seq = ["alpha", "bravo", "charlie", "delta"]

for k,v in enumerate(seq):
  print(k,v)
```

```
0 alpha
1 bravo
2 charlie
3 delta
```

[63]:
```
dict(enumerate(seq))
```

[63]:
```
{0: 'alpha', 1: 'bravo', 2: 'charlie', 3: 'delta'}
```

```
[64]: parameters = {'stock' : 100, 'strike' : 90, 'volatility' : 0.2, 'rate' : 0.1,␣
       ↪'maturity' : 1}
      print('option strike = ' + str(parameters['strike']))
```

```
option strike = 90
```

```
[65]: print('asset volatility = ' + str(parameters['Volatility']))
```

```
        ␣
  ↪---------------------------------------------------------------------------

        KeyError                                  Traceback (most recent call␣
  ↪last)

        <ipython-input-65-c26c87bd023a> in <module>
    ----> 1 print('asset volatility = ' + str(parameters['Volatility']))


        KeyError: 'Volatility'
```

```
[66]: print(parameters.keys())
```

```
dict_keys(['stock', 'strike', 'volatility', 'rate', 'maturity'])
```

```
[67]: parameters['exercise'] = 'european'
      print(parameters.keys())
```

```
dict_keys(['stock', 'strike', 'volatility', 'rate', 'maturity', 'exercise'])
```

```
[68]: print(parameters['exercise'])
```

```
european
```

```
[69]: # iterate over a dictionary (keys, by default)
      for k in parameters:
          print(k)
```

```
stock
strike
volatility
rate
maturity
exercise
```

```
[70]: # iterate over a dictionary by key and value
      for k, v in parameters.items():
```

```
    print(k, v)
```

```
stock 100
strike 90
volatility 0.2
rate 0.1
maturity 1
exercise european
```

[71]:
```
print(parameters.keys())
```

```
dict_keys(['stock', 'strike', 'volatility', 'rate', 'maturity', 'exercise'])
```

[72]:
```
print(parameters.values())
```

```
dict_values([100, 90, 0.2, 0.1, 1, 'european'])
```

Another smart way to create a dictionary from a sequence of keys (`kseq`) and a sequence of values (`vsec`) is through a built-in class constructor, `zip(kseq, vseq)` (the sequences must be of the same length):

[73]:
```
kseq   =␣
 ↪['cherries','raspberries','blueberries','strawberries','lemon','lime','apple','orange','ban
vseq   = [.4, .4, .8, .5, .6, .7, .3, .4, .5]
prices =dict(zip(kseq, vseq))
print(prices)
```

```
{'cherries': 0.4, 'raspberries': 0.4, 'blueberries': 0.8, 'strawberries': 0.5,
'lemon': 0.6, 'lime': 0.7, 'apple': 0.3, 'orange': 0.4, 'banana': 0.5}
```

[74]:
```
my_purchase = {'apple': 1,    'banana': 6}

grocery_bill = 0.0
for fruit in my_purchase:
    grocery_bill += prices[fruit] *  my_purchase[fruit]

print ('I owe the grocer $%.2f' % grocery_bill)
```

```
I owe the grocer $3.30
```

[75]:
```
my_purchase = {'strawberries': 10,    'orange': 6}

another_bill = sum(prices[fruit] * my_purchase[fruit] for fruit in my_purchase)
print ('I owe the grocer $%.2f' % another_bill)
```

```
I owe the grocer $7.40
```

Using a Dictionary to count different occurencies of characters

[76]:
```
famous_quote = "In those days spirits were brave, the stakes were high, men␣
↪were real men, women were real women \
                and small furry creatures from Alpha Centauri were real small␣
↪furry creatures from Alpha Centauri."

hashmap = {}
for c in famous_quote:
    if c in hashmap:
        hashmap[c] += 1
    else:
        hashmap[c]  = 1

print(hashmap)
```

```
{'I': 1, 'n': 8, ' ': 48, 't': 8, 'h': 6, 'o': 5, 's': 10, 'e': 27, 'd': 2, 'a':
15, 'y': 3, 'p': 3, 'i': 5, 'r': 22, 'w': 7, 'b': 1, 'v': 1, ',': 3, 'k': 1,
'g': 1, 'm': 8, 'l': 9, 'f': 4, 'u': 6, 'c': 2, 'A': 2, 'C': 2, '.': 1}
```

Generate a random list of characters

[77]:
```
import string      # import??????????? what does it means?????
import random

random.choice(string.ascii_letters)

letters = []
for k in range(100):
    letters.append(random.choice(string.ascii_letters))
```

Exercise Given an array of symbols. - A pair (i,j) is called good if item[i] == item[j] and i < j. - Return the number of good pairs.

Example

- Input: nums = [1,2,3,1,1,3]
- Output: 4
- Explanation: There are 4 good pairs (0,3), (0,4), (3,4), (2,5) 0-indexed.

link to the original problem

The first instinct that comes to mind is to use a nested for loop (aka, brutal force) to iterate over the sequence and return the count number in the end.

[78]:
```
import time

def good_pair_1(nums):
    count = 0
    for i in range(len(nums)):
        for j in range(i+1,len(nums)):
            if nums[i] == nums[j]:
                count+=1
    return count

# test case
nums = [1,2,3,1,1,3]
start = time.time()
n = good_pair_1(letters)
end = time.time()

print(n)
print(end-start)
```

```
90
0.0
```

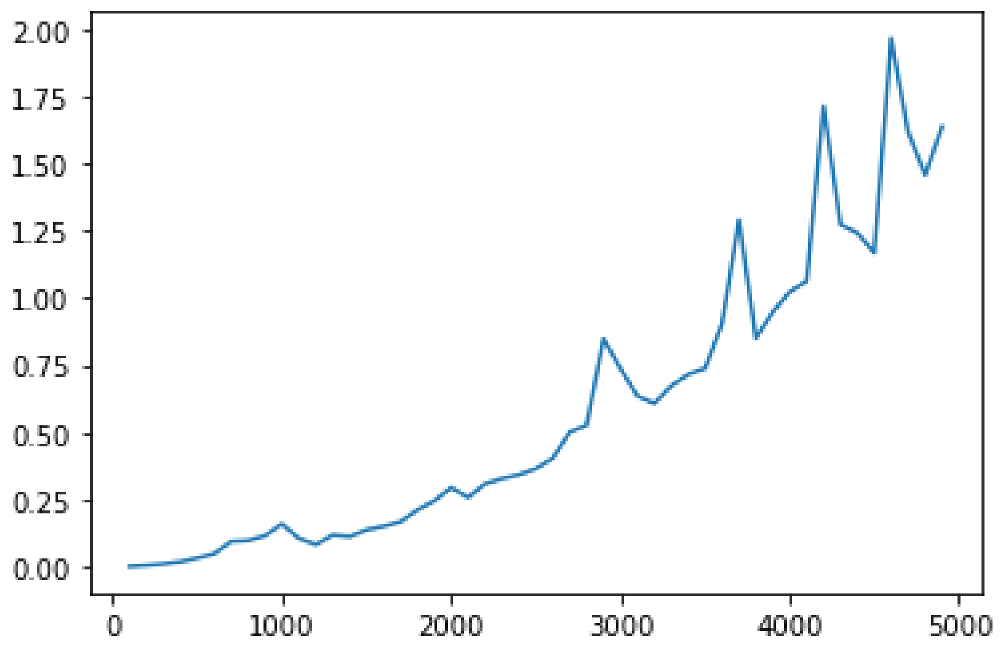
It works but slowly. One for loop is enough and not to mention a nested for loop, which has an O(N^2) time complexity!

[79]:
```
import sys

x = range(100,5000,100)
t = []

for l in x:
    letters = []
    for k in range(l):
        letters.append(random.choice(string.ascii_letters))

    start = time.time()
    m = good_pair_1(letters)
    end   = time.time()
    t.append(end-start)

    sys.stdout.write("\r" + str(l) + ' : ' + str(end-start))
    sys.stdout.flush()
```

```
4900 : 1.63747501373291026
```

[80]:
```
%matplotlib inline

import matplotlib
from matplotlib import pyplot as plt

plt.plot(x,t)
```

[80]:
```
[<matplotlib.lines.Line2D at 0x273b0ebc248>]
```



We will have a running time problem if the number of iteration is large. In this case, a dictionary is a better data type to store the data because of its key-value pair attribute. Beginners may know what a dictionary is but rarely be able to utilize the key-value feature. We treat the elements as the key and the occurrences as the value: increasing the value by 1 for each new encounter and set to 1 if it is the first time.

[81]:
```python
def good_pair_2(nums):
    hashmap = {}
    count   = 0
    for num in nums:                # read elements from nums sequentially
        if num in hashmap:
            count += hashmap[num]
            hashmap[num] += 1
        else:
            hashmap[num]  = 1
    return count

start = time.time()
m = good_pair_2(letters)
end   = time.time()

print(m)
print(end-start)
```

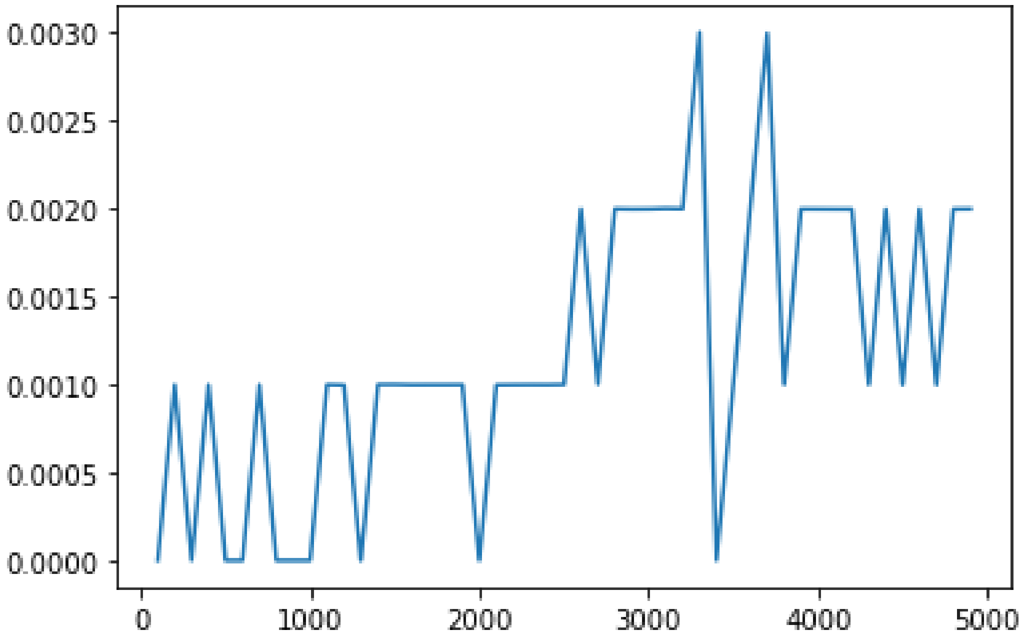
```
229845
0.002999544143676758
```

[82]:
```
import sys

x = range(100,5000,100)
t = []

for l in x:
    letters = []
    for k in range(l):
        letters.append(random.choice(string.ascii_letters))

    start = time.time()
    m = good_pair_2(letters)
    end   = time.time()
    t.append(end-start)

    sys.stdout.write("\r" + str(l) + ' : ' + str(end-start))
    sys.stdout.flush()
```

```
4900 : 0.0019993782043457034
```

[83]:
```
plt.plot(x,t)
```

[83]:
```
[<matplotlib.lines.Line2D at 0x273b111ac88>]
```



List Generator

Python implements `enumerate(seq)`, `zip(kseq, vseq)` and `range()` as *list generators*.

List generators provide an iterator >interface, which makes it possible to use them in for loops. Unlike a real list, a list generator produces the next element in a lazy way, only as needed. Generators facilitate working with large lists and even permit “infinite” lists. Remember that you can explicitly coerce a generator to a list by calling the `list()` function.

1.3.5 List Comprehension

List comprehension is an expression that transforms a collection (not necessarily a list) into a list. It is used to apply the same operation to all or some list elements, such as converting all elements to uppercase or raising them all to a power.

The transformation process looks like this: 1. The expression iterates over the collection and visits the items from the collection. 2. An optional Boolean expression (default True) is evaluated for each item. 3. If the Boolean expression is True, the loop expression is evaluated for the current item, and its value is appended to the result list. 4. If the Boolean expression is False, the item is ignored.

Here are some trivial list comprehensions:

[84]:
```
myList = [.34, -12,34,.32,-.928,32,.76543,-22.6, 556,3214,-6,.43, 0, -5]
# Copy myList; same as myList.copy() or myList[:], but less efficient
[x for x in myList]
```

[84]:
```
[0.34, -12, 34, 0.32, -0.928, 32, 0.76543, -22.6, 556, 3214, -6, 0.43, 0, -5]
```

[85]:
```
# Extract non-negative items
[x for x in myList if x >= 0]
```

[85]:
```
[0.34, 34, 0.32, 32, 0.76543, 556, 3214, 0.43, 0]
```

[86]:
```
# Build a list of squares
[x**2 for x in myList]
```

[86]:
```
[0.11560000000000002,
 144,
 1156,
 0.1024,
 0.8611840000000001,
 1024,
 0.5858830849000001,
 510.76000000000005,
 309136,
 10329796,
 36,
 0.18489999999999998,
 0,
 25]
```

```
[87]: # Build a list of valid reciprocals
      [1/x for x in myList if x != 0]
```

```
[87]: [2.941176470588235,
       -0.08333333333333333,
       0.029411764705882353,
       3.125,
       -1.0775862068965516,
       0.03125,
       1.3064551951190833,
       -0.04424778761061947,
       0.0017985611510791368,
       0.00031113876789047915,
       -0.16666666666666666,
       2.3255813953488373,
       -0.2]
```

1.4 Working with Files

Reading/Writing files are key tasks in data analysis. The Python Standard library has two methods you can use: write() e read(). Tese methods are appled to a built-in class you can create using the open() function, taking in input the file name and some additional arguments. Two arguments worth to be mentioned are the mode argument and encoding argument. Mode can be one of the following

Mode	Meaning
'r'	open for reading (default)
'w'	open for writing, truncating the file first
'x'	open for exclusive creation, failing if the file already exists
'a'	open for writing, appending to the end of the file if it exists
'b'	binary mode
't'	text mode (default)
'+'	open a disk file for updating (reading and writing)

Encoding is the name of the encoding used to decode or encode the file. The default encoding is platform dependent, to check it run the following lines

```
[88]: import locale
      locale.getpreferredencoding()
```

```
[88]: 'cp1252'
```

BE AWARE: Nowadays it is recommended to use UTF-8 encoding but you may work with text files with a different encoding and you may need to convert them in UTF-8 to represent them homogeneously in your system. The module codecs is very helpful in this case.

For the time being, we don't want to go too much in detail on this topic, you can read a gentle introduction to character sets here.

[89]:
```
if 'google.colab' in str(get_ipython()):
    from google.colab import files
    uploaded = files.upload()
    path = ''
else:
    path = './data/txt/'
```

If you want to know more about upload files in colab read this article.

[1]:
```
f = open(path + "beginning.txt", "r")
print(f.read())
f.close()
```

In the above approaches, every time the file is opened it is needed to be closed explicitly. If one forgets to close the file, it may introduce several bugs in the code, i.e. many changes in files do not go into effect until the file is properly closed. To prevent this `with` statement can be used. `with` statement in Python is used in exception handling to make the code cleaner and much more readable. It simplifies the management of common resources like file streams. Observe the following code example on how the use of with statement makes code cleaner. There is no need to call `file.close()` when using `with` statement. The `with` statement itself ensures proper acquisition and release of resources.

[91]:
```
with open(path + "beginning.txt", "r") as f:
    for x in f:
      x = x.strip()
      if x: print(x)
```

```
Galadriel: (speaking partly in Elvish)
(I amar prestar aen.)
The world is changed.
(Han matho ne nen.)
I feel it in the water.
(Han mathon ned cae.)
I feel it in the earth.
(A han noston ned gwilith.)
I smell it in the air.
Much that once was is lost, for none now live who remember it.
It began with the forging of the Great Rings. Three were given to the Elves,
immortal, wisest and fairest of all beings. Seven to the Dwarf-Lords, great
miners and craftsmen of the mountain halls. And nine, nine rings were gifted to
the race of Men, who above all else desire power. For within these rings was
bound the strength and the will to govern each race. But they were all of them
deceived, for another ring was made. Deep in the land of Mordor, in the Fires of
Mount Doom, the Dark Lord Sauron forged a master ring, and into this ring he
poured his cruelty, his malice and his will to dominate all life.
One ring to rule them all.
One by one, the free lands of Middle-Earth fell to the power of the Ring, but
```

```
there were some who resisted. A last alliance of men and elves marched against
the armies of Mordor, and on the very slopes of Mount Doom, they fought for the
freedom of Middle-Earth. Victory was near, but the power of the ring could not
be undone. It was in this moment, when all hope had faded, that Isildur, son of
the king, took up his fatherâ€ s sword.
Sauron, enemy of the free peoples of Middle-Earth, was defeated. The Ring passed
to Isildur, who had this one chance to destroy evil forever, but the hearts of
men are easily corrupted. And the ring of power has a will of its own. It
betrayed Isildur, to his death.
And some things that should not have been forgotten were lost. History became
legend. Legend became myth. And for two and a half thousand years, the ring
passed out of all knowledge. Until, when chance came, it ensnared another
bearer.
It came to the creature Gollum, who took it deep into the tunnels of the Misty
Mountains. And there it consumed him. The ring gave to Gollum unnatural long
life. For five hundred years it poisoned his mind, and in the gloom of
Gollumâ€ s cave, it waited. Darkness crept back into the forests of the world.
Rumor grew of a shadow in the East, whispers of a nameless fear, and the Ring of
Power perceived its time had come. It abandoned Gollum, but then something
happened that the Ring did not intend. It was picked up by the most unlikely
creature imaginable: a hobbit, Bilbo Baggins, of the Shire.
For the time will soon come when hobbits will shape the fortunes of all.
```

1.4.1 Processing CVS Files

A CSV file consists of columns representing variables and rows representing records. (Data scientists with a statistical background often call them observations.) The fields in a record are typically separated by commas, but other delimiters, such as tabs (tab-separated values), colons, semicolons, and vertical bars, are also common.

For convenience, the Python module csv provides a CSV reader and a CSV writer. Both objects take a previously opened text file handle as the first parameter (in the example, the file is opened with the newline=" option to avoid the need to strip the lines). You may provide the delimiter and the quote character, if needed, through the optional parameters delimiter and quotechar. Other optional parameters control the escape character, the line terminator,

[92]:
```python
if 'google.colab' in str(get_ipython()):
    from google.colab import files
    uploaded = files.upload()
    path = ''
else:
    path = './data/csv/'
```

[93]:
```python
with open(path + "countryriskdata.csv") as infile:
    reader = csv.reader(infile, delimiter=',')
```

```
     ␣
↪---------------------------------------------------------------------------
```

```
        NameError                                 Traceback (most recent call␣
↪last)

        <ipython-input-93-779bff26e8c1> in <module>
          1 with open(path + "countryriskdata.csv") as infile:
    ----> 2     reader = csv.reader(infile, delimiter=',')


        NameError: name 'csv' is not defined
```

We forget to include the "csv" module …

[94]:
```python
import csv

with open(path + "countryriskdata.csv") as infile:
    data = list(csv.reader(infile, delimiter=','))
```

Examine `data[0]`, which is the first record in the file. It must contain the column header of interest:

[95]:
```python
print(data[0])
```

```
['Country', 'Abbrev', 'Corruption', 'Peace', 'Legal', 'GDP Growth']
```

[96]:
```python
legalIndex = data[0].index('Legal')
print(legalIndex)
```

```
4
```

1.5 Functions

1.5.1 What is a Function?

A function is a block of organized, reusable code that is used to perform a single, related action. Functions provide better modularity for your application and a high degree of code reusing. These will be very familiar to anyone who has programmed in any language, and work like you would expect.

[97]:
```python
# There are thousands of functions that operate on things
print(type(3)) #type() function
print(len('hello')) #len() function
print(round(3.3))#round() funcction
```

```
<class 'int'>
5
3
```

[98]:
```
#to find out what a function does you can type it's name followed by a question␣
 ↪mark
round?
```

[99]:
```
# Not every function is available if you don't have imported it
log?
```

```
Object `log` not found.
```

[100]:
```
# this will cause an error because the log function has not been 'imported' yet
log(3)
```

```
      ␣
↪---------------------------------------------------------------------------

        NameError                                 Traceback (most recent call␣
↪last)

        <ipython-input-100-9be882efc3b0> in <module>
          1 # this will cause an error because the log function has not been␣
↪'imported' yet
    ----> 2 log(3)


        NameError: name 'log' is not defined
```

Many useful functions are not in the Python 'standard library', but in external packages which you can download or build by yourself. These need to be imported into your Python notebook (or program) before they can be used.

[2]:
```
# log() function is defined in math module

import math

#to show the module function (and methods)
#dir(math)
```

[102]:
```
math.log(3)
```

[102]:
```
1.0986122886681098
```

[103]:
```
# we don't want to write 'math.' every time
# we can use the 'from <module> import <function1>, <function2>, ...' syntax

from math import log, exp
print(log(3))
print(exp(3))
```

```
1.0986122886681098
20.085536923187668
```

1.5.2 Let's define some new Functions

First of all upload an entire book into a single variable

```
[104]: if 'google.colab' in str(get_ipython()):
           from google.colab import files
           uploaded = files.upload()
           path = ''
       else:
           path = './data/txt/'
```

```
[105]: with open(path + "carroll-alice.txt", "r") as f:
           alice = f.read()
```

```
[106]: def lexical_diversity(text):
           return len(text)/len(set(text))
```

In the definition of `lexical diversity()`, we specify a parameter labeled `text`. This parameter is a sort of "placeholder" for the actual text whose lexical diversity we want to compute, and reoccurs in the block of code that will run when the function is used, in line.

```
[107]: print(lexical_diversity(alice))
```

```
1978.013698630137
```

Uhmmmm … there is something strange… let's see why …

```
[108]: len(set(alice))
```

```
[108]: 73
```

```
[109]: print(set(alice), end='')
```

```
{'D', '"', 'J', 'W', 'h', '8', '5', ',', 'i', 'N', 'u', 'G', 'L', 'M', 'A', 'I',
'6', ')', '!', 'b', 'U', 'x', 'v', 'Q', 'S', '?', 'n', '-', 'f', 'a', 'c', 'j',
'X', ';', ']', 'V', 'z', "'", 'p', 'y', '\n', 'o', 'e', 'r', 'g', ' ', 'H', 't',
'(', 'q', 'F', 'T', 'E', 'm', 'Z', '1', 'B', '[', 'w', 'k', 'O', 's', 'd', 'K',
'R', '_', 'P', '.', ':', 'Y', 'C', '*', 'l'}
```

Tokenization In computer science, lexical analysis, lexing or tokenization is the process of converting a sequence of characters (such as in a computer program or web page) into a sequence of tokens (strings with an assigned and thus identified meaning). A program that performs lexical analysis may be termed a lexer, tokenizer, or scanner, although scanner is also a term for the first stage of a lexer.

[110]:
```
type(alice)
```

[110]:
```
str
```

[3]:
```
import nltk
nltk.download('punkt')

nltk_tokens = nltk.word_tokenize(alice)
#print (nltk_tokens)
```

```
[nltk_data] Error loading punkt: <urlopen error [Errno 11001]
[nltk_data]     getaddrinfo failed>


       ␣
 ↪---------------------------------------------------------------------------

       NameError                                 Traceback (most recent call␣
 ↪last)

       <ipython-input-3-73c3d37788dd> in <module>
         2 nltk.download('punkt')
         3
    ----> 4 nltk_tokens = nltk.word_tokenize(alice)
         5 #print (nltk_tokens)


       NameError: name 'alice' is not defined
```

[112]:
```
print(lexical_diversity(nltk_tokens))
```

```
10.51585557299843
```

[113]:
```
len(set(nltk_tokens))
```

[113]:
```
3185
```

[ ]:
```
#print(set(nltk_tokens))
```

To recap, we use or *call* a function such as `lexical_diversity()` by typing its name, followed by an open parenthesis, the name of the text, and then a close parenthesis. These parentheses will show up often; their role is to separate the name of a task—such as `lexical_diversity` from the data that the task is to be performed on—such as `nltk_tokens`. The data value that we place in the parentheses when we call a function is an **argument** to the function.

[115]:
```
strange_words = [w for w in nltk_tokens if len(w) > 15]
strange_words
```

```
[115]: ['WAISTCOAT-POCKET',
 'waistcoat-pocket',
 'bread-and-butter',
 'bread-and-butter',
 'bread-and-butter',
 'bread-and-butter',
 'bread-and-butter',
 'bread-and-butter']
```

Try to make an advanced search with google with both "waistcoat-pocket" and "bread-and-butter" …

2 Conclusion

Of course there's more to learn in Python, but these are standard things you need to know and they'll be good enough for you to jump into beginner projects. There's that tempting feeling that you need to finish lots of Python tutorials before you work on any project. Don't do it.

Many fall into the trap of learning back to back Python tutorials waiting to feel super ready. Instead, learn the basics first. Next, build some tiny projects. Then return to learning with more tutorials. Trust me, you can finish 100 Python tutorials and still feel you're not ready to build any projects. Studying alone is not enough for this reason in the next notebook you will find 10 exercises. Good luck.

3 Credits & References

Steven Bird, Ewan Klein and Edward Loper, **Natural Language Processing with Python**, O'REILLY

Andrea Gigli, **Python Course for Data Science** here the link to github