Data Foundations: Functions, Sets, and Regular Expressions

Instructor: Anthony Rios

Outline

Questions

Questions? Specific exercises you want to review?

What is a function?

A function is a **named sequence of statements** that performs a computation

You can **call** a function by invoking it:

```
>>> type(32)
<class 'int'>
```

What is a function?

A function **takes** and **argument** and **returns** a result.

The **result** is call the **return value**.

Functions we have seen before

```
>>> a = [0, 1, 2, 3]

>>> len(a) # Function call
4 # Return value

>>> newVar = len(a) # Return value stored in newVar

>>> myString = “This is a test”

>>> myString.split() # A object-specific function
[‘This’, ‘is’, ‘a’, ‘test’]
```

Random Functions

```
>>> import random

# Returns a random number between 0.0 and 1.0 (including 0, but
not 1)
>>> random.random()
0.11132867921152356

# Returns a random int between 5 and 10 (including both)
>>> random.randint(5, 10)
3
```

Adding New Functions

def <funcName>(par1, par2):

$\boxed{\text{function Body}}$

$\boxed{\text{return varName (optional)}}$

Adding New Functions

example.py

```
def print_lyrics():
    print("I'm a lumberjack, and I'm okay.")
    print("I sleep all night and I work all day.")
print_lyrics()
```

```
anthony@MacBook:~$ python example.py
I'm a lumberjack, and I'm okay.
I sleep all night and I work all day.
```

Void Functions

Functions that do **not return** a value are called **void functions**

example.py

```
def print_lyrics():
        print("I'm a lumberjack, and I'm okay.")
        print("I sleep all night and I work all day.")
revVal = print_lyrics()
print(retVal)
```

anthony@MacBook:∼$ python example.py

```
I'm a lumberjack, and I'm okay.
I sleep all night and I work all day.
None
```

Fruitful Functions

Functions that **return** a value are called **fruitful functions**

example.py

```
def addTwoNumbers(a, b):
        return a + b
revVal = addTwoNumbers(2,3)
print(retVal)
```

```
anthony@MacBook:~$ python example.py
5
```

Why functions?

- Creating a new function gives you an opportunity to **name a group of statements**, which makes your program easier to read, understand, and debug.
- Functions can make a program smaller by **eliminating repetitive code**. Later, if you make a change, you only have to make it in one place.
- Dividing a long program into functions allows you to **debug the parts one at a time and then assemble them** into a working whole.
- Well-designed functions are often useful for many programs. Once you **write and debug one, you can reuse it**.

Exercise 1

Write a function to calculate your pay given two arguments: hoursWorks and dollarsPerHour. The function should return how much you should be paid. When calculating the final amount, give the employee 1.5 times the hourly rate for hours worked above 40 hours.

⏱10 minutes

What is a Set?

A set is a **collection** which is **unordered** and **unindexed unique** objects.

Python Sets

```
>>> mySet = {‘This’, ‘is’, ‘a’, ‘set’}
```

```
>>> mySet[0]
Traceback (most recent call last):
File ”<stdin>”, line 1, in <module>
TypeError: ’set’ object does not support indexing
```

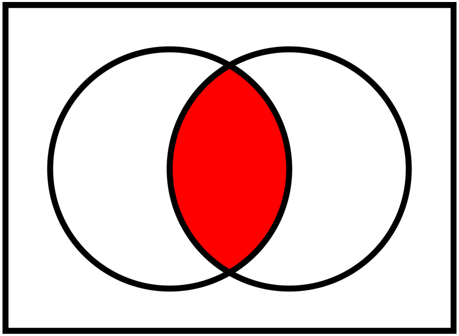
Python Sets: Unordered

example.py

```
mySet = {'This', 'is', 'a', 'set'}
for obj in mySet:
        print(obj)
```

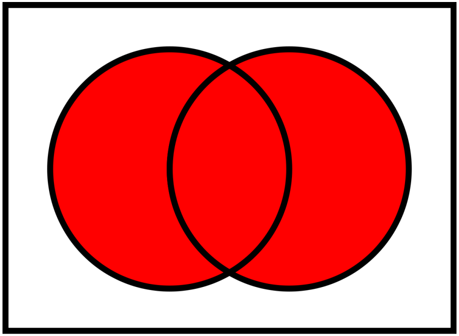
```
anthony@MacBook:~$ python example.py
set
a
This
is
```

Python Sets: Intersection



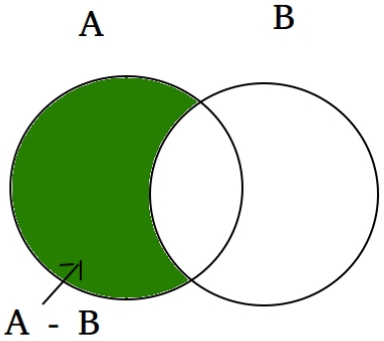
```
>>> set1 = {'This', 'is', 'a', 'set'}
>>> set2 = {'That', 'is', 'a', 'football'}
>>> set1.intersection(set2)
{'is', 'a'}
```

Python Sets: Union



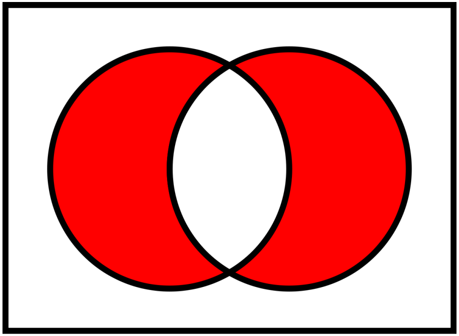
```
>>> set1 = {'This', 'is', 'a', 'set'}
>>> set2 = {'That', 'is', 'a', 'football'}
>>> set1.union(set2)
{'That', 'This', 'set', 'is', 'a', 'football'}
```

Python Sets: Difference



```
>>> set1 = {'This', 'is', 'a', 'set'}
>>> set2 = {'That', 'is', 'a', 'football'}
>>> set1.difference(set2)
{'This', 'set'}
```

Python Sets: Symmetric Difference



```
>>> set1 = {'This', 'is', 'a', 'set'}
>>> set2 = {'That', 'is', 'a', 'football'}
>>> set1.symmetric_difference(set2)
{'That', 'This', 'set', 'football'}
```

Sets vs Lists

Why should we use a set over a list?

1. Use a set if you need to hold **unique objects**
2. Searching a set for an object using a set is **faster that lists**

Exercise 2

Write code to count the number of times a “risk” word appears from the “risk_lexicon” variable in the string variable named ”text.” Please ignore case (i.e., you should lowercase everything).

The output of your code should look like the following:

```
Risk Count: 2
```

10 minutes

Introduction: Regular Expressions

A **regular expression** is a sequence of characters that forms a search pattern, mainly for use in pattern matching with strings, or string matching.

Regular Expressions and Python

```
>>> myString = 'test string abcd'

>>> import re

>>> re.search('From:', myString) # re.search(PATTERN, STRING)

>>> re.search('From:', myString) is None
True

>>> re.search('test', myString)
<_sre.SRE_Match object; span=(0, 4), match='test'>
```

If Statements and re.search

example.py

```
import re
myList = [‘line 1’, ‘test 1’, ‘line 2’, ’test 2’, ‘line 3’]
for item in myList:
        if re.search(’test’, item): # Short for “re.search is not None”
                print(item)
```

```
anthony@MacBook:∼$ python example.py
test 1
test 2
```

Exercise 3

Write code to loop over mbox.txt and counts the number of lines that contain the “From:” substring. Use the **re** package for this exercise.

5 minutes

Regular Expressions Question

If we can just use **“if ‘test’ in item”**, why should we use **re.search**?

Basic Notation

- (a | b) – a **OR** b
- (a | b)***** – **zero or more** occurrences of a OR b
- (a | b)**+** – **one or more** occurrences of a OR b

Basic Notation - Python

```
>>> myString = ‘The guppy, also known as rainbow fish, is one of the world’s most widely distributed tropical fish’
```

```
>>> re.search("gupp(y|ies)", myString) #match guppy or guppies
<_sre.SRE_Match object; span=(4, 9), match='guppy'>
```

```
>>> myString = ‘I have guppis’
>>> re.search("gupp(y|ies)", myString) # Returns None
```

Basic Notation - Python

```
>>> myString = ‘The Ohio State University’

>>> re.search(“The*”, myString)
<_sre.SRE_Match object; span=(0, 3), match='The'>

>>> myString = ‘Theeeeeee Ohio State University’

>>> re.search(“The*”, myString)
<_sre.SRE_Match object; span=(0, 9), match='Theeeeeee'>

>>> myString = ‘Th Ohio State University’

>>> re.search(“The*”, myString)
<_sre.SRE_Match object; span=(0, 2), match='Th'>
```

Basic Notation - Python

```
>>> myString = 'Th Ohio State University'
```

```
>>> re.search("The+", myString) # Returns None
>>>
```

Basic Notation

- [Bb] – match the character B OR b
- [0-9] – match all numbers 0 to 9
- [a-b] – all alphabets from a to z (lowercase)
- [A-Z] – all alphabets from A to Z (uppercase)
- [A-Za-z] – all alphabets from A to Z (uppercase or lowercase)

Basic Notation - Python

```
>>> myString = 'Woodchuck'

>>> re.search("[Ww]oodchuck", myString)
<_sre.SRE_Match object; span=(0, 9), match='Woodchuck'>

>>> myString = 'woodchuck'

>>> re.search("[Ww]oodchuck", myString)
<_sre.SRE_Match object; span=(0, 9), match='woodchuck'>

>>> myString = 'oodchuck'

>>> re.search("[Ww]oodchuck", myString) # Returns None
>>>
```

Language

A **language** is the countable set of **all possible strings** over a given alphabet.

Regular Expression vs Language

Regular Expression — Language

- a | b — {a,b}
- (a | b)(a | b) — {aa, ab, ba, bb}
- a | a*b — {a, b, ab, aab, a. . . ab}
  - a OR a (one or more) b

More Notation

- $\wedge$ a – strings that start with a
- a$ – strings that end with a
- . – matches any character except a newline
- \S – Matches any non-whitespace character

re.findall

```
>>> text = "He was carefully disguised but captured quickly by police."
```

```
>>> re.findall("\S+ly",text)
['carefully', 'quickly']
```

re.search vs re.findall

re.search searches through string looking for the **first location** where the regular expression pattern produces a match

re.findall returns **all non-overlapping matches** of pattern in string, as a list of strings.

Exercise 4

Write a program to look for lines of the form in the “mbox.txt” file:

New Revision: 39772

Extract the number from each of the lines using a regular expression and the findall() method. Compute the average of the numbers and print out the average.

12 minutes

The Regular Expression Workflow

Assume we want to count all the occurrences of the English article “the”. The following is a natural workflow for regex development:

- re.findall(“**the**”, myString)
  - This will ignore capital words (i.e., The)
- re.findall(“**[tT]he**”, myString)
  - This ignores word boundaries. We will match “other” and “theology”.
- re.findall(“**\b[tT]he\b** ”, myString)

The Regular Expression Workflow

Suppose we wanted to match "the" in more complex contexts where it may end with an underscore or numbers. For this we need to explicitly state the word boundaries:

- re.findall("**\b[tT]he\b** ", myString)
  - Will not match "the_" or "the123"
  - this could be useful when parsing twitter screen names.
- re.findall("**[∧a-zA-Z][tT]he[∧a-zA-Z]**", myString)
  - This will not match "the" at the beginning or end of a line.
- re.findall("**(∧ | [∧a-zA-Z])[tT]he[(dollar | ∧a-zA-Z])**", myString)

False Negatives/Positives

The process we just went through was based on fixing two kinds of errors: **false positives**, strings that we incorrectly matched like “other” or “there”, and **false negatives**, strings that we incorrectly missed, like “The”

Reducing the overall error rate for an application thus involves two antagonistic efforts:

- Increasing **precision** (minimize false positives)
- Increasing **recall** (minimize false negatives)

Exercise 5

For the following string:

text = “any machine with more than 6 GHz and 500 GB of disk space for less than $999.99”

Develop a regular expression that will extract **all** of the following information:

- 6 GHz
- 500 GB
- Mac
- $999.99

10 minutes