Introduction to Data Science – Week 13

NETA LIVNEH

COURSE 55793

2019-2020

Last week recap

More Pandas

This week

Last Pandas examples

Exceptions

Dates and time (just barely)

Programming tips

Course recap

Dates and time

The representation of dates and time is a bit tricky. For example, we have Unix time, local time, and many different writing methods.

We use datetime module which includes sub classes of date, time and datetime

```
In [1]: import datetime
In [2]: x = datetime.datetime.now()
   ...: x
Out[2]: datetime.datetime(2020, 1, 27, 17, 40,
33, 925869)
In [3]: print(x.year)
2020
In [4]: print(”%A"))
Monday
In [5]: x.strftime("%d-%m-%Y")
Out[5]: '27-01-2020'
```


Exceptions

```
[In [1]: int('a')
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-1-233884bacd4e> in <module>
----> 1 int('a')

ValueError: invalid literal for int() with base 10: 'a'
```

What just happened?

When there is an error in the code that is not handled, Python is prints a massage that is called ”Traceback”. It tells us where the error happened in the code and which type of error we encountered.

ValueError is a subclasses of BaseException which is a superclass for all exceptions in Python.

Raising exceptions

```
def mysterious_function(num):
    '''Takes a number 'num' and returns some other number.
    at some cases this function raises ValueError.
    '''
    if num % 3 != 0:
        raise ValueError(f'{num} must divide by 3.’)
    return int(num / 3)
```


Handling exceptions (try and except)

Let say we want to do something if an error was raised.

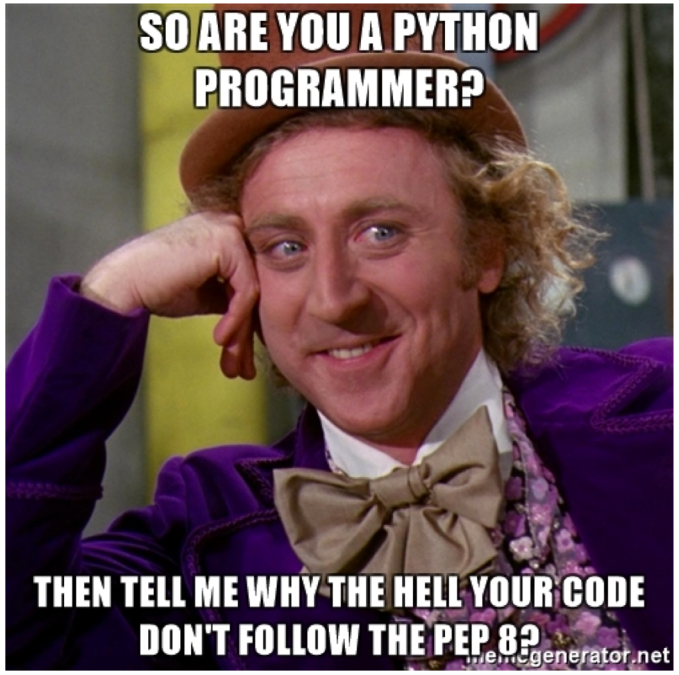
```
def check_password(password):
    if password != 'admin123’:
        raise ValueError('Wrong password')
```

```
def authenticate():
    with open('password_history.txt', 'w') as history:
        while True:
            try:
                password = input('Password: ‘)
                check_password(password)
            except ValueError as e:
                print(e)
                passed = False
            else:
                print('Welcome!’)
                passed = True
                return
            Finally:
                history.write(f'{password} (passed={passed})\n')
```


Programing tips

We would follow [Pep8 – Style guide for python code]

- Naming conventions
- Style
- Separate into functions
- Docstring
- Comments
- Other tips



Naming conventions

Function and variable names should be lowercase, with words separated by underscores as necessary to improve readability.

Never use the characters 'l' (lowercase letter el), 'O' (uppercase letter oh), or 'I' (uppercase letter eye) as single character variable names.

Never use saved names in python (e.g., len, min, max, str, string, list, etc.).

Class names should normally use the CapWords convention.

Script order

Importing

Global constants

Functions

Other code

```python
import csv


def read_csv_file(file_name):
    """
    The function reads file_name using csv.
    the function returns a dictionary with bussiness id as key and its count as
    value.
    """
    business = dict()
    with open(file_name, 'r', newline='') as my_file:
        csv_reader = csv.reader(my_file)
        # header = next(csv_reader)
        for row in csv_reader:
            bus_id = row[0]
            business[bus_id] = business.setdefault(bus_id, 0) + 1
    return business


file_name = 'yelp.csv'

if __name__ == '__main__':
    dict_business = read_csv_file(file_name)

```


Importing

```
# don't do:
from time import *
(unless needed, then use the __all__ mechanism to prevent exporting globals)
# do:
from time import time
# or:
import time
# then prepends each method with module name:
time.time()
```

```
# stdlib
import datetime
import logging
import time

# pip installed
import feedparser
import tweepy

# app modules
from config import ...
...
```

1. Standard library imports.
2. Related third party imports.
3. Local application/library specific imports.

- You should put a blank line between each group of imports.

Whitespace (1)

Immediately inside parentheses, brackets or braces:

Yes: spam(ham[1], {eggs: 2})

No: spam(ham[1], { eggs: 2 })

Between a trailing comma and a following close parenthesis:

Yes: foo = (0,)

No: bar = (0,)

Immediately before a comma, semicolon, or colon:

Yes: if x == 4: print x, y; x, y = y, x

No: if x == 4 : print x , y ; x , y = y , x

Immediately before the open parenthesis that starts the argument list of a function call:

Yes: spam(1)

No: spam (1)

Immediately before the open parenthesis that starts an indexing or slicing:

Yes: dct['key'] = lst[index]

No: dct ['key'] = lst [index]

Always surround these binary operators with a single space on either side:

Assignment (=), augmented assignment (+=, -=etc.), comparisons (==, <, >, !=, <>, <=, >=, in, not in, is, is not), Booleans (and, or, not).

Whitespace (2)

If operators with different priorities are used, consider adding whitespace around the operators with the lowest priority(ies). Never use more than one space, and always have the same amount of whitespace on both sides of a binary operator.

Yes:

```
i = i + 1 submitted += 1 x = x*2 - 1 hypot2 = x*x + y*y c = (a+b) * (a-b)
```

No:

```
i=i+1 submitted +=1 x = x * 2 - 1 hypot2 = x * x + y * y c = (a + b) * (a - b)
```

Function annotations: for Don't use spaces around the = sign when used to indicate a keyword argument, or when used to indicate a default value for an *unannotated* function parameter.

Yes:

```
def complex(real, imag=0.0):
    return magic(r=real, i=imag)
```

No:

```
def complex(real, imag = 0.0):
    return magic(r = real, i = imag)
```


Key to a good function

Is sensibly named

- Prefer full English words to abbreviations and non-universally known acronyms

Has a single responsibility

Includes a docstring

Returns a value

Is not longer than 50 lines

Is *idempotent* and, if possible, *pure*

Docstring

Every function requires a docstring

Use proper grammar and punctuation; write in complete sentences

Begins with a one-sentence summary of what the function does

Uses prescriptive rather than descriptive language

One-liners are for really obvious cases. They should really fit on one line. For example:

```
def kos_root():
    """Return the pathname of the KOS root directory."""
    global _kos_root
    if _kos_root: return _kos_root
    ...
```


Docstring

Multi-line docstrings consist of a summary line just like a one-line docstring, followed by a blank line, followed by a more elaborate description.

A docstring should document the script's function and command line syntax, environment variables, and files.

```
def complex(real=0.0, imag=0.0):
    """Form a complex number.

    Keyword arguments:
    real -- the real part (default 0.0)
    imag -- the imaginary part (default 0.0)
    """
    if imag == 0.0 and real == 0.0:
        return complex_zero
    ...
```


Comments

Comments should be complete sentences. The first word should be capitalized, unless it is an identifier that begins with a lower case letter (never alter the case of identifiers!).

Block comments:

- generally apply to some (or all) code that follows them, and are indented to the same level as that code. Each line of a block comment starts with a single hash sign (#) and a single space.
- Block comments generally consist of one or more paragraphs built out of complete sentences, with each sentence ending in a period.
- You should use two spaces after a sentence-ending period in multi- sentence comments, except after the final sentence.

Inline comments:

- A comment on the same line as a statement.
- Inline comments should be separated by at least two spaces from the statement. They should start with a hash sign (#) and a single space.
- Use inline comments sparingly only when they do not state the obvious.

Other tips

Use 4 spaces per indentation level. Use spaces, not tabs

Don’t re-invent the wheel

- use str.startswith/endswith instead of slicing

Use ''.join(list) instead of string concatenation which is slow

boolean checks don't need == True/False,

be consistent in return statements (even if returning None, use return None, not just return).

Exceptions: two common mis-uses are putting too much in the try block, or catching exceptions that are too generic, so train yourself to use specific exceptions: 'except SomeError' instead of the catch-all 'except’.

How to check Booleans: newcomers usually check False like 'if len(somelist) == 0', the Pythonic way is to just to use the fact that non-empty values are implicitly True, so in this case you could just do: 'if not somelist'.

Course recap: goals

Learn basic programming skills

Get experienced with Python

Acquire confidence in programing

Be able to use your new skills for data science tasks

Course recap: syllabus

1. Introduction
   I. What is programming?
   II. Development environment
      - Downloading Python 3
      - Installing an editor (IDE)
2. Working with Python
   I. Python shell
   II. Python as a scripting language
3. Variables
4. Python data structures (Object types)
   I. Numbers
   II. Strings
   III. Lists
   IV. Tuples
   V. Dictionaries
   VI. Sets
   VII. Boolean
5. Control Flow
   I. conditions
   II. Loops
6. Functions
7. Libraries
   I. Built in
   II. Third parties
   III. Pathlib, random, requests
8. Read and write to files
9. Regular expressions
10. Objects and Classes
11. Basic data analysis
    1. Jupyter notebook
    2. Pandas

Course Recap: topics we didn't cover

Generators (and yield)

Useful libraries: matplotlib (and other plotting libraries as seaborn and plotly), machine learning libraries (keras, sklearn), text analysis libraries (spacy)

Unicode and Byte strings

Statistics

Thanks you for attending!