Python Programming for Data Science

Week 39, Friday

Computational complexity

Exceptions

A few comments regarding handin3

Some general comments:

- Put import statements at the top of the file (generally)
- Put things that don't change outside your loop. Example: `re.compile`.
- In most cases, for-loops are more appropriate than while-loops

```
for i,line in enumerate(lines):
    print(i, line)
```

```
i = 0
while i<len(lines):
    print(i, lines[i])
    i += 1
```

Remember, there will be a complete walk-through at the Help session this afternoon.

Computational Complexity

How long does it take to find a recipy in a cookbook?

How can we answer a question like this?

Exercise: Finding a recipy in a cookbook

Let's say our cookbook contains 100 recipes.
How many recipes would we (on average) need to look at before finding the one we are looking for?

Exercise: Finding a recipy in a cookbook — solution

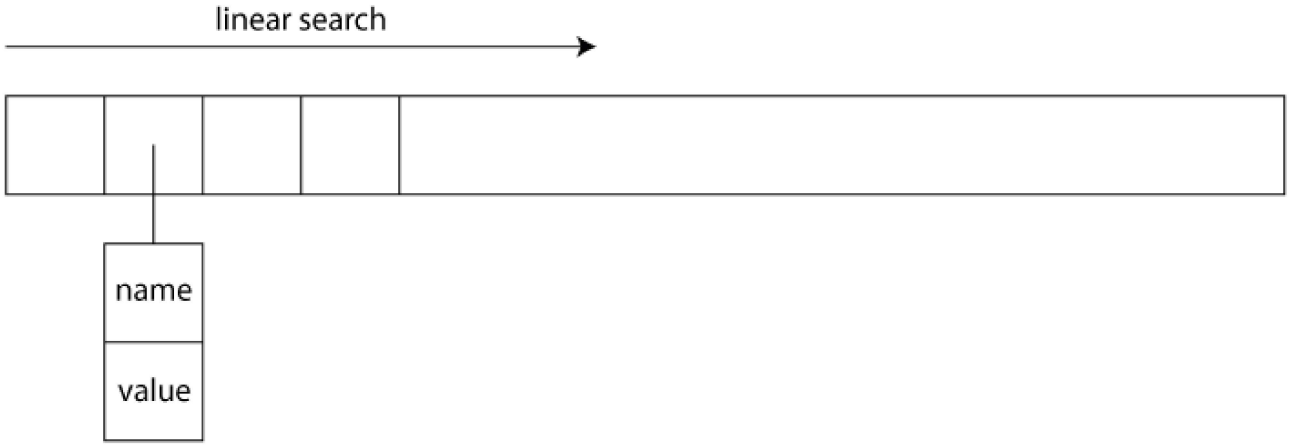
How many recipes would we (on average) need to look at before finding the one we are looking for?
Algorithm: We look at the recipes from the beginning to the end, and stop when we find it. On average, we will have to look at $100/2$ recipes.

Exercise 1

Data: wordlist with 20.000 words and an explanation of their meaning. How do we represent this?

1. Could we use a list? How many words would we need to look at to find a particular one?
2. Would it help if we sorted the elements in the list?
3. How many words would we need to look at to find a particular word if we used a dictionary?

Exercise 1.1 - solution: Linear search



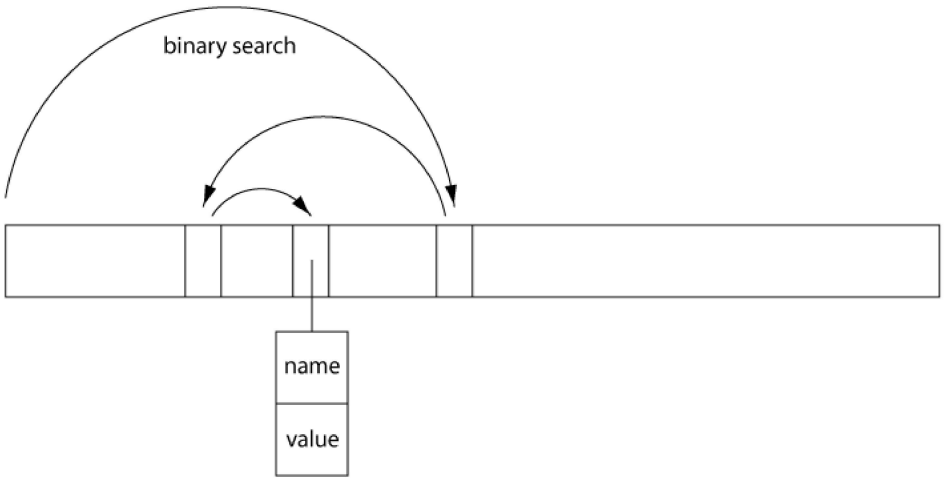
Q: How many elements will we (on average) have to look at before finding our element?
A: On average, we will have to look at 25.000/2 elements before we find the element we want

Exercise 1.2 - Does sorting help?

In other words: Is it easier to find a word in a dictionary than a recipe in a cook book?

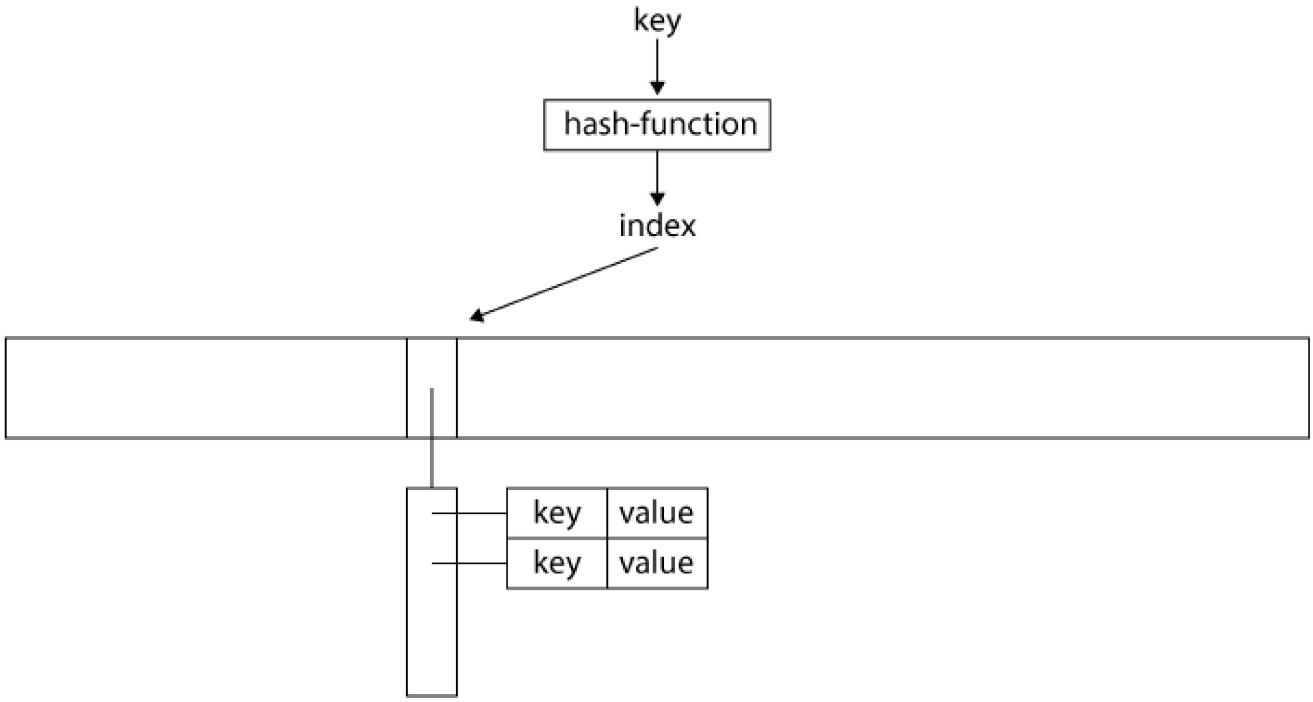
Yes.

Exercise 1.2 - solution: Binary search



Using binary search, we exclude half of the list each time. This means we have to check roughly $\log_2(25000) \approx 15$ elements before we find the element we want.
(note: sorting also takes time: $(\sim 25000 * \log_2(25000))$.

Exercise 1.3 - solution: Dictionary



A dictionary will automatically create a function that maps key values to indices in a list. This means it can do lookups in very few operations.

Structuring data

- In data science, you are often in a situation where you need to represent a large amount of data in your program.
- We have seen several examples of situations where we read in data and store it in some data structure, for instance a list of lists or a dictionary of lists.
- The speed by which a given task can be solved is often very dependent on how you represent the data.

Quantifying "the speed"

Different algorithms can be faster or slower
We want to express the efficiency of an algorithm independent on the specific choice of cookbook, and independently of whether we run on a slow or a fast computer.
"How long" can therefore not really be measured in minutes or seconds.
Instead, we can express it in terms of the *input size*, n — in our case the number of recipes in the cookbook.

Computional complexity

To quantify how "fast" an algorithm is, computer scientists introduced the concept of *computational complexity*:

- a conservative (worst case) estimate of the number of steps necessary to complete the algorithm
- expressed in terms of the size of the input n
- formally described using the big-O notation (e.g. $O(n)$)

Big-O notation

Mathematical notation that describes the asymptotic behaviour of a function.
Think of the time complexity of a program as a function $f(n)$ of the input size n.

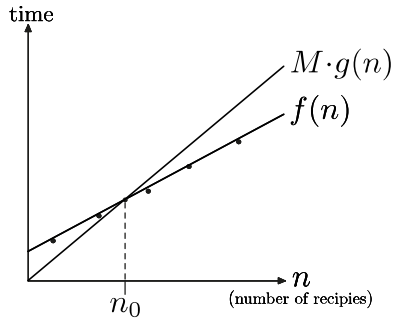
- As the input size grows, the execution time typically grows
- We don't know *exactly* how it grows
- The O formalism is designed to give an upper bound.

Big-O notation - Intuition

- If I iterate once over a list of n elements, the complexity is $O(n)$
- If I iterate twice over a list of n elements, the complexity is $O(n)$
- If I iterate any constant number of times over a list of n elements, the complexity is $O(n)$
- If I iterate n times over a list of n elements, the complexity is $O(n^2)$

Big-O notation - for the mathematically inclined

We say that
$f(n) \in O(g(n))$
if there exists values n_0 and M, such that
$|f(n)| < M \cdot |g(n)|$
for all $n > n_0$



Big-O notation - Examples

$n^2 + n + 3$	$\in$	$O(n^2)$
$3n^8 + 4n + 7n^{10}$	$\in$	$O(n^{10})$
$3n + 10$	$\in$	$O(n)$
$n + 100000$	$\in$	$O(n)$

Complexity - Exercise 2

1. What is the complexity of:

```
print(l)                          # l is a list of
length n
```

2. What is the time complexity of the following code? And what does it do?

```
for i in range(len(l)):           # l is a list of
length n
    for j in range(len(l)):
        if l[i] > l[j]:
            l[i], l[j] = l[j], l[i]
print(l)
```

3. What about this one?

```
for i in range(len(l)):           # l is a list of
length n
    for j in range(i,len(l)):
        if l[i] > l[j]:
            l[i], l[j] = l[j], l[i]
print(l)
```

Complexity - Exercise 2 - Solution (1)

```
l = [4,6,2,5,7,8,9]
print(l)
```

Even though it is just a single command, print will have to look at all elements in the list in order to print them. The complexity is therefore $O(n)$.

Complexity - Exercise 2 - Solution (2)

```
l = [4,6,2,5,7,8,9]
for i in range(len(l)):
    for j in range(len(l)):
        if l[i] > l[j]:
            l[i], l[j] = l[j], l[i]
print(l)
```

Complexity: $O(n^2 + n) = O(n^2)$
This algorithm is called *bubble sort*. It sorts the lists.
The built in sort method in python lists is faster. It has complexity $O(n \cdot \log(n))$

Complexity - Exercise 2 - Solution (3)

```
l = [4,6,2,5,7,8,9]
for i in range(len(l)):
    for j in range(i,len(l)):
        if l[i] > l[j]:
            l[i], l[j] = l[j], l[i]
print(l)
```

Complexity: $O(n(n+1)/2+n) = O(n^2)$
We can improve the bubble sort by starting j at index i in the inner loop. This speeds up the algorithm by a factor 2 - but does not change the complexity.

Complexity - Exercise 3

1. What is the complexity of looking up an element in a list using []?
2. What is the complexity of removing an element in a Python list (using pop())?
3. What is the complexity of the strip() method in the string class?
4. What is the complexity of your handin3_test.py?

Complexity - Exercise 3 - solutions

1. What is the complexity of looking up an element in a list using []? $O(1)$
2. What is the complexity of removing an element in a Python list (using pop())? $O(1)$ (at end) or $O(n)$ (at arbitrary position)
3. What is the complexity of the strip() method in the string class? Trick question: $O(n)$ - strings are immutable
4. What is the complexity of your handin3_test.py? $O(n)$, where n is the number of lines in the file.

Exceptions

Exceptions

Exceptions are used to indicate that something exceptional (usually bad) has happened in a program
Raising an exceptions immediately interrupt the program's execution.
Exceptions are mainly used for:

- Errors and error recovery
- Notifications of special events

We have already seen exceptions in action many times. Every time you have written a program that crashed this has been due to an exception.

Exceptions - example

```
exceptions.py
a = ?
```

```
output

  File "exceptions.py", line 1
    a = ?
        ^
SyntaxError: invalid syntax
```

The program crashes because we assign ? to the variable a.

This does not make sense, so Python complains.

Python automatically raises built-in exception when it discovers an error, and terminates the program (default behavior).

Exceptions - more examples

exceptions.py

```
d = {'Mon':'Monday', 'Tue':'Tuesday'}
print(d[Wed])
```

output

```
Traceback (most recent call last):
  File "exceptions.py", line 2, in
    print(d[Wed])
NameError: name 'Wed' is not defined
```

Q: What's wrong here? What is Python trying to tell us?

A: Python thinks Wed is a variable, which is not defined

Exceptions - more examples (1)

```
exceptions.py
print(d['Wed'])      # Specify Wed as a string
```

```
output
Traceback (most recent call last):
  File "exceptions.py", line 2, in
    print(d['Wed'])
KeyError: 'Wed'
```

Q: What's wrong this time? What is Python trying to tell us?

A: The key 'Wed' is not present in our dictionary.

Catching exceptions

Rather than letting the program crash, you can define how to respond to an exception
This is done using *try-catch* statements

errors.py
```
d = {'Mon':'Monday', 'Tue':'Tuesday'}
try:
    d['Wed']      # you can have as many lines as you want in here
except:
    # print keys to screen when error occurs.
    print("Key not found in dictionary. Available keys:", d.keys())
```

output
```
Key not found in dictionary. Available keys:  ['Mon', 'Tue']
```

Now that the exception has been caught, it will no longer crash your program.

Catching specific exceptions

Rather than catching *any* type of exception, you can specify that you only want to catch exceptions of a certain type.

errors.py
```
d = {'Mon':'Monday', 'Tue':'Tuesday'}
try:
    d['Wed']
except KeyError:  # <- Only catch KeyError exceptions
    # print keys to screen when error occurs.
    print("Key not found in dictionary. Available keys: ",
d.keys())
```

output
```
Key not found in dictionary. Available keys:  ['Mon', 'Tue']
```

Catching specific exceptions (continued)

If we catch the wrong exception type, we will get the original behavior:

errors.py

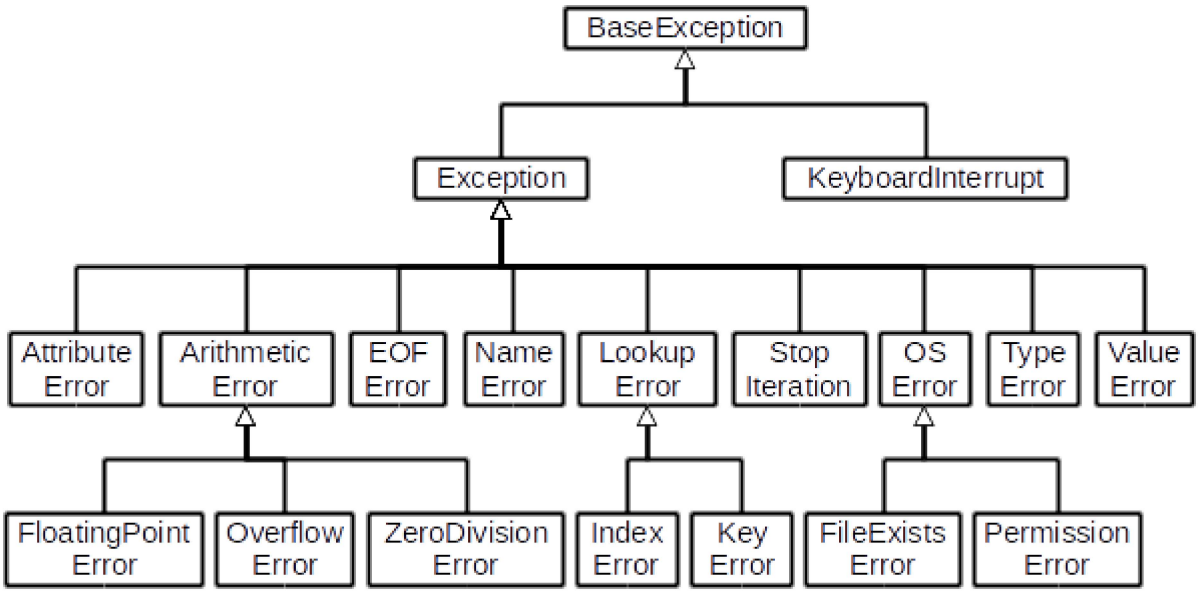
```
d = {'Mon':'Monday', 'Tue':'Tuesday'}
try:
    d['Wed']
except NameError:  # <- Only catch NameError exceptions
    # print keys to screen when error occurs.
    print("Key not found in dictionary. Available keys: ",
d.keys())
```

output

```
Traceback (most recent call last):
  File "errors.py", line 2, in <module>
    d['Wed']
KeyError: 'Wed'       # <- NOTE: our bug produces a
KeyError
```

Exceptions – Class hierarchy

Pythons built-in exceptions are structured in a class hierarchy.



Catching specific exceptions (continued 2)

The class hierarchy makes it possible to be more flexible in specifying which exceptions to catch
The exception type that you specify ***and all derived classes from this class*** will be caught

errors.py

```
d = {'Mon':'Monday', 'Tue':'Tuesday'}
try:
    d['Wed']
except Exception:  # Exception covers both NameError and
KeyError
    print("Key not found in dictionary. Available keys: ",
d.keys())
```

Catching exceptions – accessing details

You can access the exception object itself to gain extra information regarding the error.
This this done using the *as* keyword

errors.py
```
d = {'Mon':'Monday', 'Tue':'Tuesday'}
try:
    d['Wed']
except Exception as error:   # Save error object
    # error contains information about key that failed
    print("Information about error: " + str(error))
```

output
```
Information about error: 'Wed'
```

Exercise

Write the most specific try/except statement for each of the following situations:

1.

```
print(I_love_python)
```

2.

```
l=[1,2,3,4]
print(l[4])
```

3.

```
print(1/0)
```

Exercise - solution 1

1.

```
exceptions.py
print(I_love_python)
```

```
output
Traceback (most recent call last):
  File "exceptions.py", line 1, in
    print(I_love_python)
NameError: name 'I_love_python' is not defined
```

We note that our mistake raises a NameError. Solution:

```
try:
    print(I_love_python)
except NameError as error:
    print("Something is wrong here: %s" % error)
```

Exercise - solution 2

2.

```
exceptions.py
l=[1,2,3,4]
print(l[4])
```

```
output
Traceback (most recent call last):
  File "exceptions.py", line 2, in
    print(l[4])
IndexError: list index out of range
```

We note that our mistake raises an IndexError. Solution:

```
l=[1,2,3,4]
try:
    print(l[4])
except IndexError as error:
    print("Something is wrong here: %s" % error)
```

Exercise - solution 3

3.

```
exceptions.py
print(1/0)
```

```
output
Traceback (most recent call last):
  File "exceptions.py", line 1, in
    print(1/0)
ZeroDivisionError: integer division or modulo by zero
```

We note that our mistake raises a ZeroDivisionError.

Solution:

```
try:
    print(1/0)
except ZeroDivisionError as error:
    print("Something is wrong here: %s" % error)
```

Raising exceptions

You can also raise the built-in exceptions yourself:

```
exceptions.py
raise KeyError
```

```
output
Traceback (most recent call last):
  File "exceptions.py", line 1, in
    raise KeyError
KeyError
```

The raise statement can be used at any place in your code to interrupt the program

Exceptions – the assert statement

It is common to insert sanity checks in a program, testing that a particular condition is always true

```
if not condition:
    raise exception
```

There is a statement called assert that does this automatically

```
assert condition    # raises an AssertionError if
condition is not True
```

Example:

exceptions.py

```
x = 2
assert x>0 and
x<10
assert x<0 and
x>10
```

output

```
Traceback (most recent call last):
  File "exceptions.py", line 3, in
    assert x<0 and x>10
AssertionError
```

Exercise

Write a function that takes a single number as argument

1. The function should check that this number is an even number (0,2,4,6,...) and raise an exception otherwise.
2. Call the function with an uneven number:
    - Without catching the exception
    - Catching the exception and printing a warning.

Exercise - solution

errors.py

```
def my_function(value):
    '''Function without any functionality. But assumes
that value is even.'''
    assert value%2 == 0    # Raise exception if value is
not even

# Call with uneven number
my_function(1)
```

output

```
Traceback (most recent call last):
  File "errors.py", line 22, in <module>
    my_function(1)
  File "errors.py", line 13, in my_function
    assert value%2 == 0
AssertionError
```

errors.py

```
# Call with uneven number - catch AssertionError
try:
    my_function(1)
except AssertionError:
    print("Assertion failed in function")
```

output

```
Assertion failed in function
```

Defining your own exceptions

Rather than using the built-in exception types, you can define your own exceptions, to cover specific problems relevant to your program
This is done by creating a class that inherits from Exception

```
class MyException(Exception):   # Defining a very simple exception class
    pass

raise MyException               # Raising an exception of this type
```

```
Traceback (most recent call last):
  File "errors.py", line 8, in <module>
    raise MyException
__main__.MyException
```

For more information, see **the python docs**.

try...except...finally

Consider this code

```
try:
    data_file = open("british-english")

    words = {}
    for i, line in enumerate(data_file):
        words[line.strip()] = i

    print(words['bokeh'])

    data_file.close()
except KeyError as error:
    print("Word %s not found" % str(error))
```

Q: What's wrong here? A: We don't close the file when an exception occurs

try...except...finally(2)

You can fix the previous example by adding a finally clause to your try-except statement

```
try:
    data_file = open("british-english")

    words = {}
    for i, line in enumerate(data_file):
        words[line.strip()] = i

    print(words['bokeh'])

except KeyError as error:
    print("Word %s not found" % str(error))
finally:
    data_file.close()
```

This code will be executed no matter whether an exception was raised or not

The `with` statement

This pattern is very common

```
thing = initialize_thing()   # (e.g. open a file)
try:
    # do something with thing
finally:
    finalize_thing()         # (e.g. close a file)
```

For this reason, the following shortcut was introduced

```
with initialize_thing() as thing:
    # do something with thing
```

Example from before:

```
with open("british-english") as data_file:
    words = {}
    for i, line in enumerate(data_file):
        words[line.strip()] = i
    print(words['bokeh'])
```

The `with` statement (2)

```
with open("british-english") as data_file:
    words = {}
    for i, line in enumerate(data_file):
        words[line.strip()] = i
    print(words['bokeh'])
```

Note how the file is now automatically closed when the file exits the with statement

This only works for classes that have implemented the following two methods:

- __enter__: Initializes object
- __exit__: Finalizes object

The file class is one such example

Speaker notes