

# Dictionaryes

Thomas Schwarz, SJ

# Opening and Reading Text Files

- Python follows the posix conventions:
  - You can open a file
  - You can interact with a file
  - You then close the file
- Easiest done with a **Python context**
  - The context automatically closes the file after use

```
with open(filename) as infile:  
    |- statement block -|
```

# Opening and Reading Text Files

- To read from a file, we can use a for loop

```
with open(filename) as infile:
    for line in infile:
        |- do something with each line
```

- Within the for loop, we can use `strip()` in order to break the line apart at white spaces

```
with open(filename) as infile:
    for line in infile:
        for words in line.split( )
```

# Dictionaries

- Python has a efficient association data structure — the dictionary
  - Dictionary pairs keys with values
    - Useful for: indices
    - Useful for: translations
    - Useful for: quick lookups
      - E.g.: first letters —> full email address
      - E.g.: human-readable URL —> IP address
      - ...

# Dictionaries

- Dictionaries are key-value stores
  - Keys — anything, but needs to be immutable
    - Remember: Lists are mutable, strings are immutable
  - Value — anything

# Dictionaries

- Dictionaries are created by using curly brackets

- Can use lists

```
dicc = {1: 'uno', 2: 'dos', 3: 'tres' }
```

- Or can use assignment

```
dicc = { }
```

```
dicc[1] = "uno"
```

```
dicc[2] = "dos"
```

```
dicc[3] = "tres"
```

- Values are assigned / retrieved using the bracket notation

# Dictionary

- Dictionary `dicc={ }`

- Accessing values:

```
dicc['key']
```

```
>>> dicc = {1: "uno", 2: "dos", 3: "tres"}
>>> dicc[1]
'uno'
>>> dicc[1] = "one"
>>> dicc[1]
'one'
```

- With default value

```
dicc.get(key, default_value)
```

- Or with if - else

```
if key in dicc:
```

- Creating / changing values

```
dicc['key'] = value
```

# Dictionary

- Deleting from a dictionary

```
dicc = {}
```

- Use the `del` keyword

- Raises a key error if the key is not in the dictionary

```
if key in dicc:  
    del dicc[key]
```

- Use the `pop` method, which returns the value

```
value = dicc.pop(key)
```

```
value = dicc.pop(key, default)
```



# Dictionary

- Checking for existence
  - Use the “in” keyword

```
>>> dicc = {1: "uno", 2: "dos", 3: "tres"}
>>> 1 in dicc
True
>>> 4 not in dicc
True
```

# Dictionaries

- A simple program that “learns” Spanish words

```
def test():
    dicc = {}
    while True:
        astr = input("Enter an English word: ")
        if astr == "Stop it":
            return
        elif astr in dicc:
            print(dicc[astr])
        else:
            print("I have not yet learned this word")
            val = input("Please enter the Spanish word: ")
            dicc[astr] = val
```

# Dictionaries

- Dictionaries have an internal structure
  - You will learn in Data Structures how to build dictionaries yourselves
  - For the moment, enjoy their power
- You can print dictionaries
  - You will notice that they change structure after inserts and not reflect the order in which you inserted elements
  - This is because they optimize access

# Dictionaries

- Deleting all entries in a dictionary
  - use the `clear()` method
- Deleting an entry without fear of creating a key error
  - Use an if statement
  - Use `pop` with a second argument `None`
    - `dicc.pop(1, None)`

# Dictionaries

- Looping over keys
  - Simplest:
    - `for number in dicc:`
  - `iterkeys()` or `iter` works the same way
    - `for number in dicc.iterkeys():`
    - `for number in iter(dicc):`

# Some Uses of Dictionaries

- Dictionaries can be used to count things.
  - Example: Count the number of letters in a file.
    - We open the file with encoding latin-1 so that there are no encoding errors

```
alphabet = "abcdefghijklmnopqrstuvwxyz"
```

```
with open("alice.txt", encoding = "latin-1") as infile:  
    dicc = {}  
    for letter in alphabet:  
        dicc[letter]=0
```

# Some Uses of Dictionaries

- Create and initialize a dictionary
  - We are only interested in letters

```
alphabet = "abcdefghijklmnopqrstuvwxyz"
```

```
with open("alice.txt", encoding = "latin-1") as infile:  
    dicc = {}  
    for letter in alphabet:  
        dicc[letter]=0
```

# Some Uses of Dictionaries

- Read the file line by line.
  - Read each letter in the line
    - After changing to lower case, update dictionary

```
alphabet = "abcdefghijklmnopqrstuvwxyz"
```

```
with open("alice.txt", encoding = "latin-1") as infile:  
    dicc = {}  
    for letter in alphabet:  
        dicc[letter]=0  
    for line in infile:  
        for letter in line:  
            letter=letter.lower()  
            if letter in alphabet:  
                dicc[letter]+=1
```



# Some Uses of Dictionaries

- Now process the dictionary
  - Calculate the sum of values (i.e. the counts)
  - Pretty-print the results

```
for letter in alphabet:
    cum += dicc[letter]
for letter in alphabet:
    print("{:1s} {:5d} {:5.2f}%".format(
        letter, dicc[letter], dicc[letter]/cum*100))
```

# Some Uses of Dictionaries

- Using lists as dictionary values
  - in order to create an index of words in a file

# Some Uses of Dictionaries

- Open file with encoding “latin-1”
  - Read file line by line
    - Break line into words
    - Normalize words by stripping and lowering

```
with open("alice.txt", encoding = "latin-1") as infile:  
    index = {}  
    word_count = 0  
    for line in infile:  
        for word in line.split():  
            word_count += 1  
            word = word.lower().strip(",. ; : ? ! [ ] - ' \")
```

# Some Uses of Dictionaries

- Add word to dictionary if long enough

```
with open("alice.txt", encoding = "latin-1") as infile:
    index = {}
    word_count = 0
    for line in infile:
        for word in line.split():
            word_count += 1
            word = word.lower().strip(",.;;?![]-'\")
            if len(word)>7:
                if word in index:
                    index[word].append(word_count)
                else:
                    index[word] = [word_count]
```

# Some Uses of Dictionaries

- Print out results if word is frequent enough

```
for word in index:  
    if len(index[word])>2:  
        print(word, index[word])
```

# A Teaser on Iterators

- Iterators are the hidden engine of many Python features
  - Iterators are almost like lists
    - You always can get the next element
      - Unless you are at the end of a list
  - But they are not lists:
    - All the elements in the list have to be there before the list can be used
      - They need to be stored in memory
      - Which uses up space
      - And can be disastrous if there are just too many

# A Teaser on Iterators

- Iterators are only created when there is a need
- Iterators are often hidden from view
- But we will have to use them
  - For our purposes:
    - We can make them explicitly into lists because we are just not working with millions of data items
    - But hopefully, once we get to play with the grown-ups ...
- Seriously, we get back to iterators

# Multi-Dictionaries

- Problem:
  - Instead of associating one value with a key, we want to associate several values:
    - a “multi-dictionary”
- Solution:
  - The values of the dictionaries should be lists (or sets — coming week)



# Multi-Dictionaries

- Example:
  - We want to pass through a file and create an index of important words with their occurrences

```
with open("alice.txt", encoding = "latin-1") as infile:
    dicc = {}
    word_number = 0
    for line in infile:
        for word in line.split():
            word = word.strip(" : , . ? ! [ ] ' ")
            word = word.lower()
            word_number += 1
            if len(word) > 8:
                if word in dicc:
                    dicc[word].append(word_number)
                else:
                    dicc[word] = [word_number]
```

# Calculating on Values

- Assume you have a dictionary with numerical values
  - For example: a dictionary with the prices of stocks on September 15, 2018
- You want the average, the maximum, the minimum ... price

```
dstocks = {"tata": 2063.30,  
           "hdfc": 2029.20,  
           "hiul": 1630.15,  
           ...  
           }
```

# Solution

- You can access the values of a dictionary through the values method.
- `values()` returns an iterator of all the values in the dictionary

```
>>> dst = {"apple": 256.34, "fb": 145.23, "ibm": 98.34, "ms": 198.75}
>>> dst.values()
dict_values([256.34, 145.23, 98.34, 198.75])
>>> max(dst.values())
256.34
>>> sum(dst.values())/len(dst.values())
174.665
```

# Calculating with keys

- Problem:
  - You want to calculate on the keys of a dictionary
- Solution:
  - The `keys()` method returns an iterator of the keys of a dictionary

# Finding the most common item in a list

- We use a dictionary as a counter.
  - First way: We can do so by ourselves.
    - Create a dictionary
      - Pass through the list

```
def most_frequent(lista):  
    counter = {}  
    for x in lista:  
        counter[x]=counter.get(x, 0)+1
```

get specifies a default value, it is otherwise equivalent to counter[x]

# Finding the most common item in a list

- If we do not want to use get, we can just check whether the list-item is already in the dictionary

```
def most_frequent(lista):  
    counter = {}  
    for x in lista:  
        if x in counter:  
            counter[x]+=1  
        else:  
            counter[x]=1
```

# Finding the most common item in a list

- After counting, we pass through the dictionary to find the maximum element.
- Notice that we are interested in the key, not the value

```
def most_frequent(lista):  
    counter = {}  
    for x in lista:  
        counter[x]=counter.get(x, 0)+1  
highest_seen = 0  
for x in counter:  
    if counter[x]>highest_seen:  
        best_key = x  
        highest_seen = counter[x]  
return best_key
```

highest\_seen contains the highest encountered value

# Finding the most common item in a list

- After counting, we pass through the dictionary to find the maximum element.
- Notice that we are interested in the key, not the value

```
def most_frequent(lista):  
    counter = {}  
    for x in lista:  
        counter[x]=counter.get(x, 0)+1  
highest_seen = 0  
for x in counter:  
    if counter[x]>highest_seen:  
        best_key = x  
        highest_seen = counter[x]  
return best_key
```

highest\_seen is adjusted  
whenever we see a higher  
value in the counter



# Finding the most common item in a list

- After counting, we pass through the dictionary to find the maximum element.
- Notice that we are interested in the key, not the value

```
def most_frequent(lista):  
    counter = {}  
    for x in lista:  
        counter[x]=counter.get(x, 0)+1  
    highest_seen = 0  
    for x in counter:  
        if counter[x]>highest_seen:  
            best_key = x  
            highest_seen = counter[x]  
    return best_key
```

but we also need to remember the key, which we record in best\_key

# Finding the most common item in a list

- After counting, we pass through the dictionary to find the maximum element.
- Notice that we are interested in the key, not the value

```
def most_frequent(lista):  
    counter = {}  
    for x in lista:  
        counter[x]=counter.get(x, 0)+1  
    highest_seen = 0  
    for x in counter:  
        if counter[x]>highest_seen:  
            best_key = x  
            highest_seen = counter[x]  
    return best_key
```

because the key with the highest counter value is the result that we return

# Finding the most common item in a list

- But we can also use the work of others
  - The `Counter` in the `collections` module
    - You create a *new object* of type `Counter`

```
from collections import Counter

def most_frequent(lista):
    ctr = Counter()
```

**Defines a new object called `ctr`  
`ctr` is an object of type `Counter`**

# Finding the most common item in a list

- Counters are (updated) like dictionaries
  - But they have a default value of 0

```
from collections import Counter

def most_frequent(lista):
    ctr = Counter()
    for item in lista:
        ctr[item] += 1
```

**Here we add 1 to  
the value of  
ctr[item]**

**No need to initialize!**

# Finding the most common item in a list

- Counters have a method called `most_common`
  - Argument is the number of most common items
  - Returns a list of pairs

```
from collections import Counter

def most_frequent(lista):
    ctr = Counter()
    for item in lista:
        ctr[item] += 1
    return ctr.most_common(1)[0][0]
```

- Get a list of one elements.
- Get the first (and only) element of the list
- Get the first coordinate of that element

# Memoization

- (Some) Computer Scientists love recursion
  - A function calls itself
    - This is super-elegant and the more mathematically inclined pine for this elegance
  - But it is not necessarily very fast
    - The more engineeringly inclined think its a waste

# Recursion

- When it works
  - Factorials
    - The factorial of  $n$  is  $n (n-1) (n-2) (n-3) \dots (4) (3) (2) (1)$
    - Define it to be one for negative or zero  $n$

# Recursion

- This implementation has the function factorial call itself

```
def factorial(number):  
    if number < 1:  
        return 1  
    else:  
        return number * factorial(number - 1)
```

- Here we are calling on the function itself
- Will call factorial(number-1), which will call factorial(number-2), which will call factorial(number-3) ... until we call factorial on 1, in which case the recursion stops.



# Recursion

- This implementation has the function factorial call itself

```
def factorial(number):  
    if number < 1:  
        return 1  
    else:  
        return number * factorial(number - 1)
```

- **The base case:**
  - We cannot call recursion infinitely often, so we need one.

# Recursion

- The Fibonacci numbers
  - The Fibonacci numbers are defined recursively
  - $f_0 = 0, \quad f_1 = 1, \quad f_n = f_{n-1} + f_{n-2}$

```
def fibonacci(number):  
    if number <= 0:  
        return 0  
    if number == 1:  
        return 1  
    return fibonacci(number-1)+fibonacci(number-2)
```

# Recursion

- But this implementation is inane!
  - Takes too long even for small numbers.
    - We can use the time-module in order to obtain the cpu-time
      - We do so once before and after execution of the function
  - This yields approximately the time it takes to execute the function

# Recursion

- We just write a function that measures the time

```
def measure(function, number):  
    start = time.time()  
    function(number)  
    print(number, time.time()-start)
```

# Recursion

- Now we try it out with factorial and fibonacci
  - Not a problem with factorial

```
27 1.52587890625e-05
28 1.5974044799804688e-05
29 1.52587890625e-05
30 1.5735626220703125e-05
31 1.811981201171875e-05
32 1.71661376953125e-05
33 1.7881393432617188e-05
34 1.7881393432617188e-05
35 1.9073486328125e-05
36 1.9788742065429688e-05
37 1.8835067749023438e-05
38 2.09808349609375e-05
39 2.193450927734375e-05
```

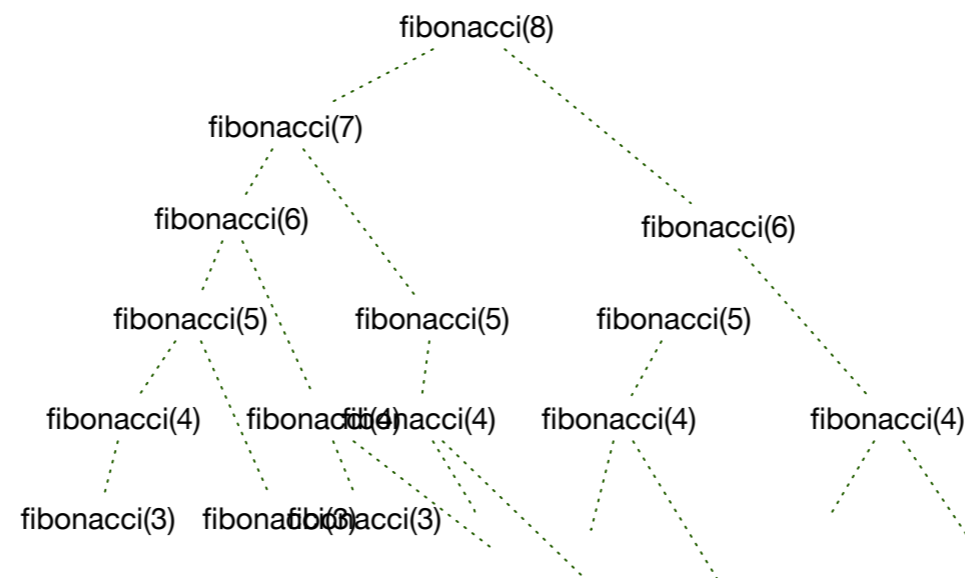
# Recursion

- But disastrous for Fibonacci
- It takes 34 seconds in order to calculate fibonacci(39).

```
28 0.17530512809753418
29 0.27112603187561035
30 0.43769311904907227
31 0.7113552093505859
32 1.1374599933624268
33 1.846013069152832
34 2.9945621490478516
35 4.856478929519653
36 7.85633397102356
37 12.681456804275513
38 20.59703803062439
39 33.98105502128601
```

# Recursion

- What is the problem?
  - Look at what happens if we calculate fibonacci(9).
  - We calculate fibonacci(8) and fibonacci(7)
    - Since the first one also calculates fibonacci(7), we calculate fibonacci(7) twice.
    - And it gets worse for fibonacci(6), fibonacci(5), ...



# Memoization

- A simple trick to speed up recursive functions is to remember values that we have already calculated.
- Create a dictionary (possibly global) that stores values already calculated
  - Before any calculation check whether the desired value is in the dictionary
  - If we calculate something, we put the value into the dictionary



# Memoization

```
fdic={0: 0, 1:1}

def fibonacci2(number):
    if number in fdic:
        return fdic[number]
    else:
        retval = fibonacci2(number-1)+fibonacci2(number-2)
        fdic[number] = retval
        return retval

for i in range(41):
    measure(fibonacci2, i*50)
```

# Memoization

```
fdic={0: 0, 1:1}
```

```
def fibonacci2(number):  
    if number in fdic:  
        return fdic[number]  
    else:  
        retval = fibonacci2(number-1)+fibonacci2(number-2)  
        fdic[number] = retval  
        return retval  
  
for i in range(41):  
    measure(fibonacci2, i*50)
```

- Defining the dictionary

# Memoization

```
fdic={0: 0, 1:1}

def fibonacci2(number):
    if number in fdic:
        return fdic[number]
    else:
        retval = fibonacci2(number-1)+fibonacci2(number-2)
        fdic[number] = retval
        return retval

for i in range(41):
    measure(fibonacci2, i*50)
```

- Check whether value is in the dictionary

# Memoization

```
fdic={0: 0, 1:1}

def fibonacci2(number):
    if number in fdic:
        return fdic[number]
    else:
        retval = fibonacci2(number-1)+fibonacci2(number-2)
        fdic[number] = retval
        return retval

for i in range(41):
    measure(fibonacci2, i*50)
```

- Calculation is necessary

# Memoization

```
fdic={0: 0, 1:1}

def fibonacci2(number):
    if number in fdic:
        return fdic[number]
    else:
        retval = fibonacci2(number-1)+fibonacci2(number-2)
        fdic[number] = retval
        return retval

for i in range(41):
    measure(fibonacci2, i*50)
```

- But we store the result in the dictionary in case we use it in the future

# Memoization

```
fdic={0: 0, 1:1}

def fibonacci2(number):
    if number in fdic:
        return fdic[number]
    else:
        retval = fibonacci2(number-1)+fibonacci2(number-2)
        fdic[number] = retval
        return retval

for i in range(41):
    measure(fibonacci2, i*50)
```

- And now we measure

# Decorators

- Python uses decorators to allow changing functions
- A decorator is implemented by:
  - Creating a function of a function that returns the amended function

# Decorators

```
def timeit(function):
    def clocked(*args):
        start_time = time.perf_counter()
        result = function(*args)
        duration = (time.perf_counter() - start_time)
        name = function.__name__
        arg_string = ', '.join(repr(arg) for arg in args)
        print('Function {} with arguments {} ran
              in {} seconds'.format(
                name, arg_string, duration))
        return result
    return clocked
```



# Decorators

- Decorator takes a function with positional arguments as function
- Decorator defines a new version of the argument function
- And returns it.

# Decorators

```
def timeit(function):
    def clocked(*args):
        start_time = time.perf_counter()
        result = function(*args)
        duration = (time.perf_counter() - start_time)
        name = function.__name__
        arg_string = ', '.join(repr(arg) for arg in args)
        print('Function {} with arguments {} ran
              in {} seconds'.format(
                name, arg_string, duration))
        return result
    return clocked
```

# Decorators

- To use a decorator, just put its name on top of the function definition
  - Decorator generator is executed when module is imported (or generator is defined)
  - When decorated function is defined, the modified version is created

# Decorators

```
@timeit
def fibonacci(n):
    if n == 0:
        return 0
    if n == 1:
        return 1
    else:
        return fibonacci(n-1)+fibonacci(n-2)
```

# Decorators

- If we execute this function, we get to see how often fibonacci is called on arguments already executed

```
>>> fibonacci(10)
Function fibonacci with arguments 1 ran in 5.140000070014139e-07 seconds
Function fibonacci with arguments 0 ran in 1.0870000011209413e-06 seconds
Function fibonacci with arguments 2 ran in 0.16927908399999958 seconds
Function fibonacci with arguments 1 ran in 1.2330000060956081e-06 seconds
Function fibonacci with arguments 3 ran in 0.2676633440000131 seconds
Function fibonacci with arguments 1 ran in 9.8000001003129e-07 seconds
Function fibonacci with arguments 0 ran in 1.0470000120221812e-06 seconds
Function fibonacci with arguments 2 ran in 0.09880945999999824 seconds
Function fibonacci with arguments 4 ran in 0.4692909440000079 seconds
Function fibonacci with arguments 1 ran in 6.51999997103303e-07 seconds
Function fibonacci with arguments 0 ran in 1.0500000087176886e-06 seconds
Function fibonacci with arguments 2 ran in 0.11281222700000626 seconds
Function fibonacci with arguments 1 ran in 1.958000012791672e-06 seconds
Function fibonacci with arguments 3 ran in 0.21685028000000273 seconds
Function fibonacci with arguments 5 ran in 0.7868284680000102 seconds
Function fibonacci with arguments 1 ran in 5.6999999742402e-07 seconds
Function fibonacci with arguments 0 ran in 1.0729999928571488e-06 seconds
Function fibonacci with arguments 2 ran in 0.11366798399998856 seconds
Function fibonacci with arguments 1 ran in 1.2930000110600304e-06 seconds
Function fibonacci with arguments 3 ran in 0.2176230820000029 seconds
Function fibonacci with arguments 1 ran in 5.839999914769578e-07 seconds
```

# Memoization with lru\_cache

- We can define our own memoization decorator
  - But Python has one that uses an LRU cache
    - Memoization is LRU cache with an infinite cache size
    - Import from functools lru\_cache

```
@functools.lru_cache
def fib(n):
    if n <= 1:
        return n
    else:
        return fib(n-1)+fib(n-2)
```