Programming I

Course 7

Introduction to programming

What we talked about?

- Testing
    - User perspective
    - Programmer perspective
        - Simplest way – use assert

- Debugging
    - Simplest way – use print

What we talk today?

- `Modules`
- List Comprehension
- Generators
- Recursive Functions
- Files

Modules

- Functions
  - Use for …
- Modules
  - User for …
    - Group data & functions
    - Group into a file
  - Types
    - Standard
    - Custom modules (your own)

Modules. Import

- Default way
  - `import myModule`
  - To call a function use dot(.) to prefix function with module name
    - `var = myModule.foo()`
- from <module> import <function>
  - Not prefix each time the function with module name

  -
    ```
    from myModule import foo
    var = foo()
    ```

    Import ONLY foo() function from myModule

  -
    ```
    from myModule import *
    var = foo()
    ```

    Import ALL functions from myModule

Modules. Import

- Default way

- from <module> import <function>

- import <module> as <name>
  - If a module has a long name can be replaced with a shortest name
  - `import myModule as mm`

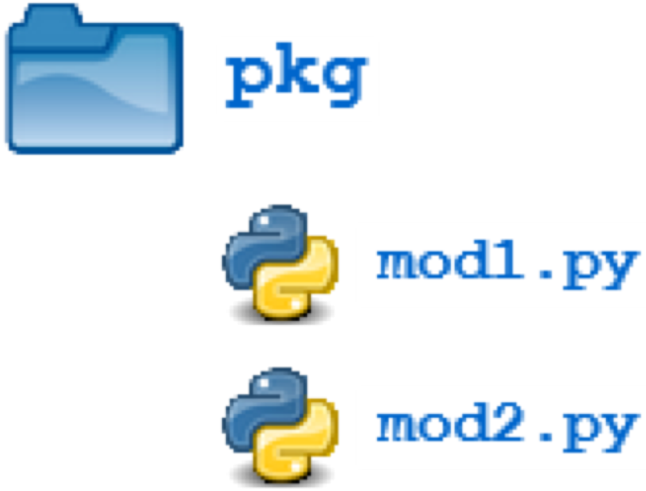
    ```
    var = mm.foo()
    ```

Module Identification

- Where does Python looks for modules?
  - Standard path
    - A list of paths where all standards modules are placed
    - PYTHONPATH
    - sys.path
  - Working directory
    - The project root directory
    - Our case
      - The directory where the norebook is placed

Modules. Packages

- A way of structuring Python's module namespace by using "dotted module names"



Usage example

```
import pkg.mod1
import pkg.mod2
```

Find Information about Modules

- What functions does it offer?
  - `dir()` – function

- Example

  -
    ```
    import math
     dir(math)
    ```
    =>
    ```
    ['__doc__', '__name__', 'acos', 'asin', 'atan', 'atan2', 'ceil',
    'cos', 'cosh', 'degrees', 'e', 'exp', 'fabs', 'floor', 'fmod',
    'frexp', 'hypot', 'ldexp', 'log', 'log10', 'modf', 'pi', 'pow',
    'radians', 'sin', 'sinh', 'sqrt', 'tan', 'tanh']
    ```

  -
    ```
    print (math.__doc__)
    print(math.sin.__doc__)
    ```

Specific details about module/functions

__doc__ atribute: gives the documentation string or comment that the programmer put (as you should put) at the head of his module

Modules – pyc Files

- Python compile module in order to speed up execution
  - Result a `pyc` file
  - "compile" - conversion to ‘byte code’.
  - If the source code of the module is changed Python notice and will recompile next time when the module is used

- The point is that the byte code in the .pyc module will run much faster than if the module is interpreted every time.

- You don’t have to worry about any of this. Just ignore the .pyc files. If you delete them, Python will re-create them when it needs to

What we talk today?

- Modules
- `List Comprehension`
- Generators
- Recursive Functions
- Files

List Comprehension

- How we create a matrix with *n* lines and *n* columns filled with *0*?

- Solution 1

```
mat =[]
for i in range(n):
    line = []
    for j in range(n):
        line.append(0)
    mat.append(line)
```

- Solution 2

```
mat = [ [0]**n for i in range(n)]
```

List comprehension

List Comprehension

- Allow you to create lists with a for loop with less code
- Part of functional programming in Python
- Examples
  - comp_list = [x ** 2 for x in range(7) if x % 2 == 0] -> [4, 16, 36]

  -
    ```
    nums = [1, 2, 3, 4, 5]
    letters = ['A', 'B', 'C']
    nums_letters = [[n, l] for n in nums for l in letters]
    ```

  -> [[1, 'A'], [1, 'B'], [1, 'C'], [2, 'A'], [2, 'B'], [2, 'C'], [3, 'A'], [3, 'B'], [3, 'C'], [4, 'A'], [4, 'B'], [4, 'C'], [5, 'A'], [5, 'B'], [5, 'C']]

List Comprehension

- Works with other data structures
  - `dict_comp = {x:chr(65+x) for x in range(1, 11)} ->` {1: 'B', 2: 'C', 3: 'D', 4: 'E', 5: 'F', 6: 'G', 7: 'H', 8: 'I', 9: 'J', 10: 'K'}
  - `set_comp = {x**3 for x in range(10) if x%2 == 0} ->` {0, 8, 64, 512, 216}

What we talk today?

- Modules
- List Comprehension
- `Generators`
- Recursive Functions
- Files

Generators

- Lets consider the following problem with the following implementation
    - Define a function that calculates the sum of first *n* numbers where *n* is a very large number

```
def func(n):
    S = 0
    for i in range(n):
        S += i
    return S
```

- What does range function?

Generates a list of *n* numbers in memory
n a large number
=>
A LARGE PART OF MEMORY IS OCUPATED

Generators

- Usually when a function is called the program control is passed to the function until the function terminates (it reaches the final statement, it encounters return instruction or an exception is generated)

- Generators
  - Allows creation of a function that behaves like an iterator on a sequence (list, set, map, tuple)

- Iterator is a way to walk through a sequence using *next()* function in order to obtain the next element from the sequence
  - *It is the way that is used by for loop in order to obtain the elements of a sequence*

Generators

- Are functions that use yield keyword instead of return

- Yield allows the preservation of function variables state until a next call on it is done
  - If *return* instruction is used the values of the variables are destroyed and at the next call are initialized again

- Syntax

```
yield <variable>
```

If <variable> is missing it the returned value is `None`

Generators

Initial function

```
def func(n):
    S = 0
    for i in range(n):
        S += i
    return S
```

Constructs the hole list of elements in memory

Using generators

```
def generator(n):
    i=0
    while i <= n:
        yield i
        i += 1


def func(n):
    S = 0
    for i in generator(n):
        S += i
    return S
```

Obtain a new value each time it is needed

Generators - Internals

```
def myGenerator(l):
     total = 0
     for n in l:
       yield total
       total += n
```

```
newGenerator = myGenerator([10,20,30])

print(next(newGenerator))
print(next(newGenerator))
print(next(newGenerator))

# result

# 0
# 10
# 30
```

Generators - expressions

Using function

```
def generator():
        for item in collection:
                yield expression
```

Using expressions

genexpr = (expression for item in collection)

```
even_squares = (x * x for x
in range(10) if x % 2 == 0)
```

Generators

- Generators are used to *generate* a series of values
- yield is like the return of generator functions
- The only other thing yield does is save the "state" of a generator function
- A generator is just a special type of iterator
- Like iterators, we can get the next value from a generator using next()
    - for gets values by calling next() implicitly

Generators

- Less memory consumption
  - Generators help to minimize memory consumption, especially when dealing with large data sets, because a generator will only return one item at a time.
- Better performance and optimisation
  - Generators are lazy in nature
    - Only generate values when required to do so; unlike a normal iterator, where all values are generated regardless of whether they will be used or not, generators only generate the values needed.
    - Program performing faster

What we talk today?

- Modules
- List Comprehension
- Generators
- Recursive Functions
- Files

Recursive Functions

- What is recursion?
    - Process of repeating items in a self-similar way.
- Algorithmically: a way to design solutions to problems by divide-and-conquer or decrease-and-conquer
    - reduce a problem to simpler versions of the same problem
- Semantically: a programming technique where a function calls itself
    - in programming, goal is to NOT have infinite recursion
    - must have 1 or more base cases that are easy to solve
    - must solve the same problem on some other input with the goal of simplifying the larger problem input

Recursion

- Consider the following problem
  - Calculate factorial for a number n

  - How we can write n!

    - n! = 1*2*…*n

    - n! = n * (n-1)!

Recursion

ITERARIVE SOLUTION

```
def fact1(n):
    f = 1
    i = 1
    while i <= n:
        f = f * i
        i = i + 1
    return f
```

Using loops:
for or while

RECURSIVE SOLUTION

```
def fact2(n):
    if n == 1:
        return 1
    else:
        return n* fact2(n-1)
```

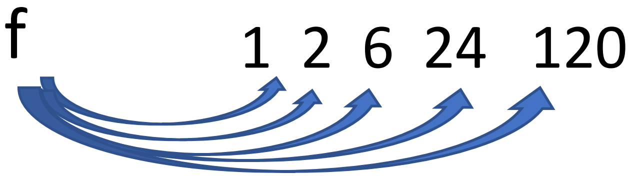
Base case

Recursive call

Recursion – What happens for n=5?

ITERARIVE SOLUTION

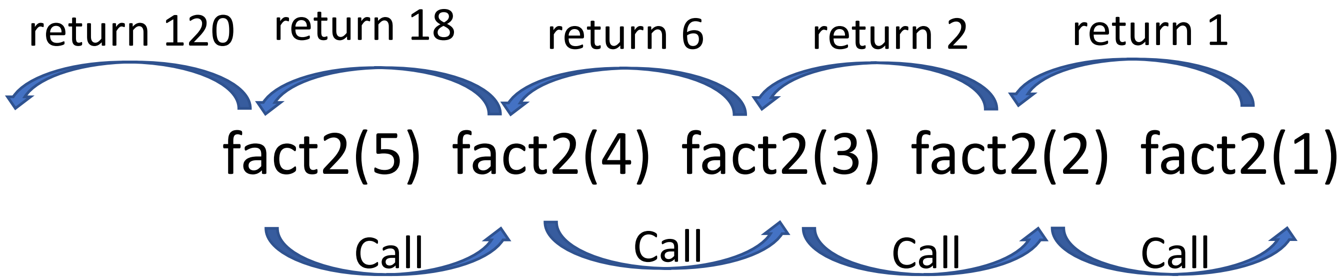
```
def fact1(n):
    f = 1
    i = 1
    while i <= n:
        f = f * i
        i = i + 1
    return f
```



f 1 2 6 24 120

RECURSIVE SOLUTION

```
def fact2(n):
    if n == 1:
        return 1
    else:
        return n* fact2(n-1)
```



return 120 return 18 return 6 return 2 return 1

fact2(5) fact2(4) fact2(3) fact2(2) fact2(1)

Call Call Call Call

Recursivon – What happens for n=5?

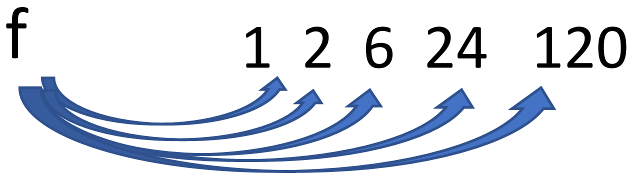
ITERARIVE SOLUTION

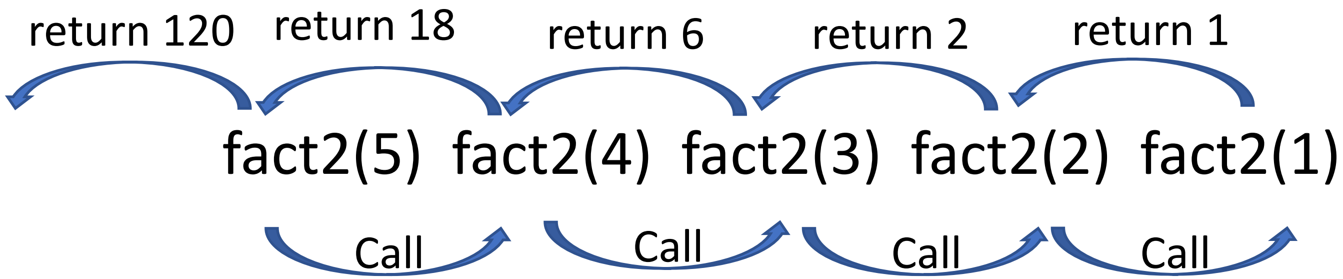
- More efficient from computer POV

RECURSIVE SOLUTION

- Easiest to implement for programmers
- Not efficient
  - Because – function call stack

↑ Function local variables
↑ Function parameters
↑Function call





Function call stack

- each recursive call to a function creates its own scope/environment
- bindings of variables in a scope are not changed by recursive call
- flow of control passes back to previous scope once function call returns value

Story … Fibonaccy numbers

- Leonardo of Pisa (aka Fibonacci) modeled the following challenge
  - Newborn pair of rabbits (one female, one male) are put in a pen
  - Rabbits mate at age of one month
  - Rabbits have a one month gestaSon period
  - Assume rabbits never die, that female always produces one new pair (one male, one female) every month from its second month on.
  - How many female rabbits are there at the end of one year?

Recursion – Recursive Solution

- Recursive step
  - think how to reduce problem to a simpler/ smaller version of same problem
- Base case
  - keep reducing problem until reach a simple case that can be solved directly

- Fibonaccy numbers

1 1 2 3 5 8 13 …

  - Recursive step
    - $F_n = F_{n-1} + F_{n-2}$
  - Base case
    - $F_1 = 1$
    - $F_2 = 1$

What we talk today?

- Modules
- List Comprehension
- Generators
- Recursive Functions
- `Files`

Files

- A file is a sequence of data stored in secondary memory (usually on a physical environment: magnetic disk, SSD, etc)

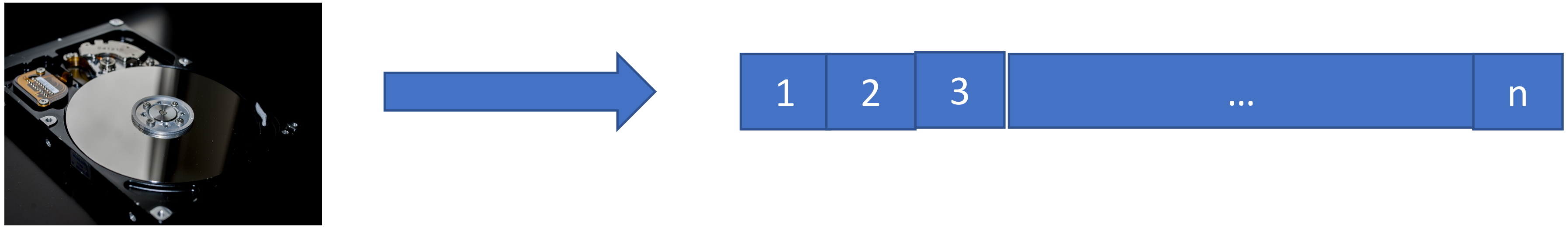
- Can contain any data type
  - Easy to read: text
  - Hard to read: binary (e.g. open an image file into a text editor)

- Files assures persistence (as long the physical support allows this)

- Files allows the possibility to work with big data
  - Not dependent of principal memory dimmension

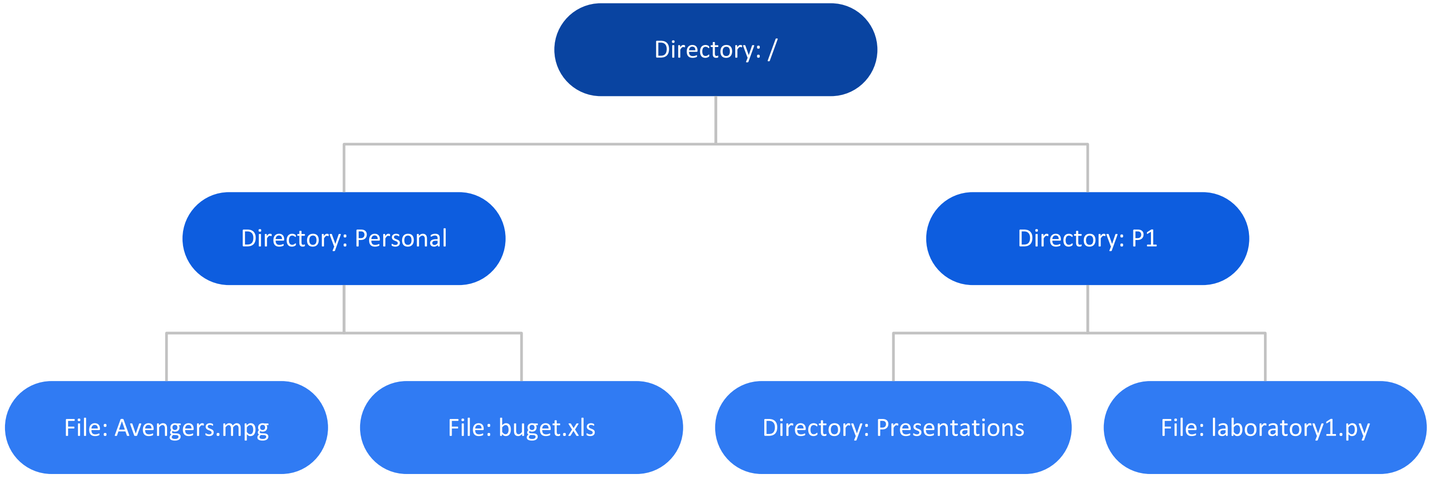
What is a file system?

- A hierarchical structure that organize and allows the file access (a logical data grouping)
- Managed by operating system
- Physical support, offers through operating system an linear abstraction of it, in shape of blocks
  - Blocks – a sequence of octets without any other "exposed" organization form
- The operating system through file system offers a view of the octets sequence



What is a file system?

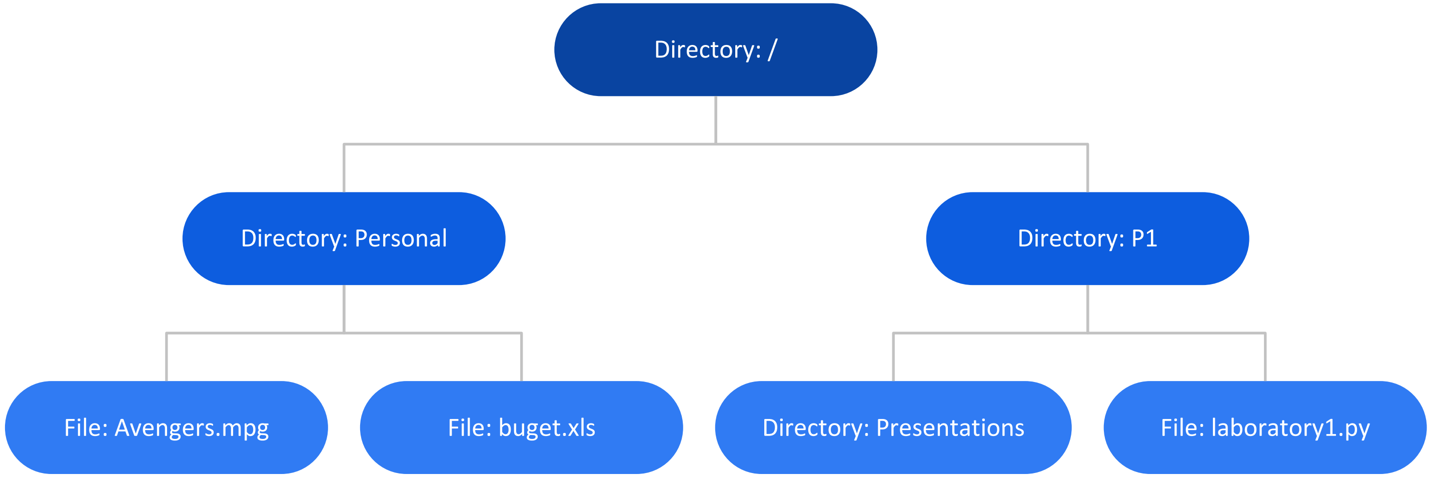
- A hierarchical structure that organize and allows the file access (a logical data grouping)
- Manage the data distribution on physical device (the data are not necessary sequential but the file system offers a "sequential view" of them



What is a file system?

More details at Operating Sytems course

- This structure is exposed to the applications by operating system through VFS(Virtual File System)
- Hides the implementation details and allows to concentrate on the file content



Directory: /

Directory: Personal

Directory: P1

File: Avengers.mpg

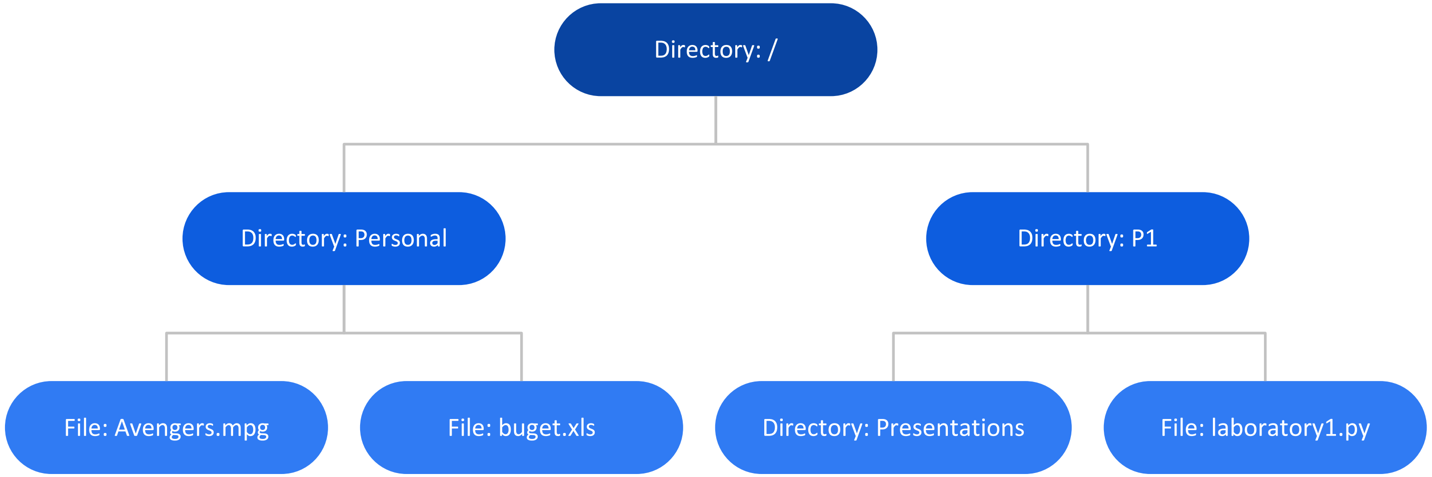
File: buget.xls

Directory: Presentations

File: laboratory1.py

What is a file system?

- Base elements
  - Files – usually an atomically unit (does not have divisions from operating system POV)
  - Directories – a collection of files and directories
- Offers operation for files and directory (paths) management



Operations Offered by File System

- File system (files & directories) management
  - Creation: `create` (create file), `mkdir` (create directory)
  - Removal: `remove/unlink` (delete object)
  - Rename/move: `rename` (object rename)
- Objects are identified by a
  - Name
  - Path inside the file system
- Example

  Path Name

  - `/Users/fmicota/Documents/note-p1.csv`
- The character '/' is used to separate the path.
  - Windows uses '\' but also accept '/'

File operations

- Create
  - The operation of file creation, allocates necessary resources in file system
- Open
  - The operation of file opening and associate a logic identifier to the file ("file handler")
- Read
  - The operation of transfer of the data from the file (storage device) in principal memory of processing unit
- Write
  - The operation of transfer of data from principal memory to the file (storage device)
- Seek
  - Positioning the file pointer to the place where data are read/write
- Close
  - The synchronization of all data in file and resource release

File Operations in Python - open

- Creation and opening a file
    - `fisier = open("/tmp/file.txt", "r")`

The path were the file is located

The type of operations (read/write/append) that can be done on the file

- Syntax
    - Open (path, open_type)

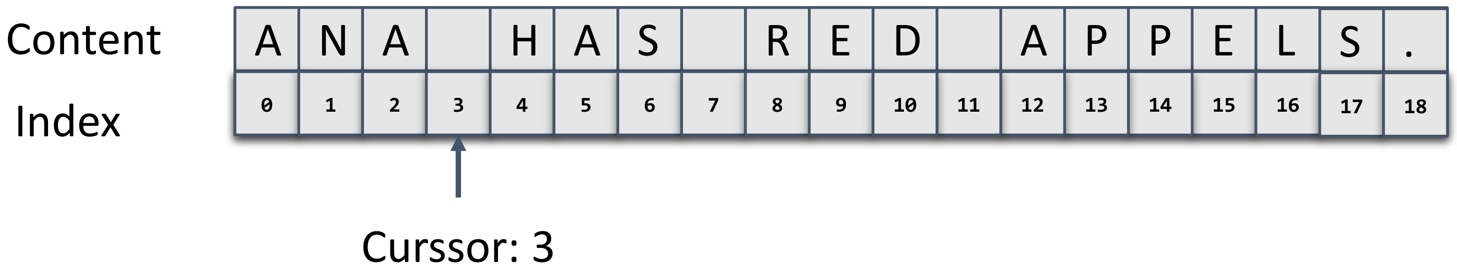
File Operations in Python

- Syntax
  - open (path, open_type)

r	Opens a file for read operation. Does not create the file. It is the default value.	**w**	Opens a file for write operation. If the file exist it is truncated (deletes its content), otherwise it creates a new empty file.
r+	Opens a file for read/write operations. Does not create the file. The file curssor is placed at the strat of the file.	**w+**	Opens a file for read/write operation. If the file exist it is truncated (deletes its content), otherwise it creates a new empty file.
a	Opens a file for write operation. It creates the file, if the file does not exist. The file curssor is placed at the end of the file.	**a+**	Opens a file for write operation. If the file does not exist a new empty file is created. The file curssor is placed at the end of the file.

File Operations in Python

- What is a file curssor?
  - A identifier that tells the position in the file where the operations of read/write start



Content	A	N	A		H	A	S		R	E	D		A	P	P	E	L	S	.
Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18

Curssor: 3

- In the example the operation starts at index 3.
- The file index starts with 0
- The current value of the cursor can be found with *tell()* function

File Operations in Python - close

- *close()* method is used to close a file object obtained by using *open()* function

- Example

```
file = open("example.txt, "w+")
file.close()
```

- Close operation assures that all data are written on the disk
  - Sometimes the written data are stored into a buffer zone that is not flushed until the file is closed (some modification in file are not visible)

- If the file is closed no operation is possible on it

File Operations in Python - read

- *read()* method is used to 'read' data from a file object obtained by using *open()* function

- Syntax
  - `read(size=-1)`

- *read()* – reads and returns *'size'* characters (if *'size'* < file length then it return as much characters it could read)
- If *'size'*==-1 then it reads the hole file
- The function returns the read characters like s string
- The read operation is relative to file cursor

File Operations in Python - read

- Text files
  - `readline(size=-1)`
    - Reads a characters until it reaches new line character into a file, the number of characters read from a line can be limited by the *'size'* argument
  - `readlines(hint=-1)`
    - Reads multiple lines from a file, the number of lines is limited by *'hint'* argument

File Operations in Python – write

- *write()* method is used to 'write' data to a file object obtained by using *open()* function

- Syntax
  - `write(text)`

- The function writes the string *'text'* into a file
- It returns the number of written characters in the file
- The write operation overwrites the file content
- If the end of the file is reached the file is resized in order to store all data

File Operations in Python - seek

- *seek()* method is used to seek data in a file object obtained by using *open()* function

- Syntax
  - `seek(cookie, whence=0)`

- The function moves the cursor at the position specified by argument *'cookie'* (also known like offset)
- The value of argument *'whence'* is
  - 0 – start of the file, *'cookie'* can have a positive value (move forward)
  - 1 – current position of cursor in file, *'cookie'* can have a positive value (move forward) or negative value (move backward)
  - 2 – end of the file , *'cookie'* can have a negative value (move backward).

File Operations in Python - truncate

- *truncate()* method is used to 'truncate' the data from a file object obtained by using *open()* function

- Syntax
  - `truncate(pos=None)`

- The function truncates the file until *'pos'* position in file (remove the data after *'pos'* position)
- The *'pos'* argument is relative to file beginning, if it is missing the current cursor position is used to truncate the file

File Operations in Python - Example

```
try:
    f = open("fisier.txt", "w+")
    sir = f.read()
    sir = sir.upper()
    f.seek()
    f.write(sir)
finally:
    f.close()
```

Opens a file for read/write operations

Read the data from the disk and load into the memory

Goes the the beginning of the file and replace the content of the file with the uppercase text.

Close the file -> the modifications are flushed on disk

File Operations in Python - with

- 'try:…finally:…' block can be replaced 'with' with instruction, that assures that the close operation is executed each time

```
with open("fisier.txt", "w+") as f:
   sir = f.read()
   sir = sir.upper()
   f.seek()
   f.write(sir)
```

Binary Files

- Python offers support for binary files ("non-text file")

- Opening
  - Similarly with text files, adding **b** at "open_type" argument
  - `open ( "a.dat", "rb")`

- All operation except *readline()* and *readlines()* are available for binary files

- The functions read/write return/receive objects of type 'bytes' not strings

bytes and bytearray Datatype

- `bytes` datatype is used to represent a immutable sequence of octets
- `bytearray` is used to represent a mutable sequence of octets

- Conversion from string to byte
  - `b=bytes("a text", "utf8")`

Utf8 is a encoding convention for characters, other conventions are ASCII, UTF16, ISO-8859-1

- Conversion from byte to string
  - `decode(encoding)`

'Encoding' is a codification, if the conversion is not possible a `UnicodeDecodeError` is raised

Semistructured files

- Again text file ..
- Semitructured fies
  - Files that follow a structure
- Example
  - **C**omma **S**eparated **V**alues (CSV)
  - **J**ava**S**cript **O**bject **N**otation (JSON)
  - e**X**tensible **M**arkup **L**anguage (XML)

```
Name; mark; year
Poescu Ion; 10; 2
Vasilescu Vasile; 9; 1
```

```
[{'name':'Popescu Ion','mark':10,'year':2},
{'name':Vasilescu Vasile','mark':9,'year':1}]
```

```
<students>
<student name="Popescu Ion"><mark>10</mark><year>2</year></student>
<student name="Vasilescu Vasile"><mark>9</mark><year>1</year></student>
</students>
```

CSV

- A CSV file (Comma Separated Values file) is a type of plain text file that uses specific structuring to arrange tabular data.

- Because it's a plain text file, it can contain only actual text data—in other words, printable ASCII or Unicode characters.

- Example

```
Name; mark; year
Poescu Ion; 10; 2
Vasilescu Vasile; 9; 1
```

- Python supports CSV natively
  - `import csv`

CSV

- Load data from CSV – file

```
with open('student.csv') as csv_file:
    csv_reader = csv.reader(csv_file, delimiter=';')
    line_count = 0
    for row in csv_reader:
        if line_count == 0:
            print('Column names are {}'.format(", ".join(row)))
            line_count += 1
        else:
            print('\tStudent {} in year {} has the mark {}.'.format(row[0],row[2],row[1]))
            line_count += 1
    print(f'Processed {line_count} lines.')
```

CSV

- Store data from CSV – file

```
import csv
lst = [['Name', 'Age', "Passion"], \
       ['Maria', 20, 'drawing'], \
       ['Jon', 19, 'computers']]

with open('student_out.csv', 'w') as csv_file:
    csv_writer = csv.writer(csv_file, delimiter=';')

    for row in lst:
        csv_writer.writerow(row)
```

JSON

- Format for
    - exchanging information (WEB),
    - Information storage (database)
- Readable by anyone
- JSON supports primitive types, like strings and numbers, as well as nested lists and objects.

```
{ "firstName": "Jane",
  "lastName": "Doe",
  "hobbies": ["running", "sky diving", "singing"],
   "age": 35,
   "children": [
        { "firstName": "Alice", "age": 6 },
        { "firstName": "Bob", "age": 8 } ]
}
```

JSON

- Python supports JSON natively
  - `import json`

- json library
  - LOAD JSON in a python dictionary
    - `json_data= '{ "firstName": "Jane", "lastName": "Doe", "hobbies": ["running", "sky diving", "singing"]}'`
    - `my_dict = json.loads(json_data)`

  - STORE python dictionary into a JSON format
    - `json.dumps(my_dictionary)`
    - `json.dumps(my_dictionary, indent=4)`
    - `json.dumps(my_dictionary, indent=4, sort_keys=True)`

JSON

- Loading/Storing from/into a file

- Loading

```
with open("data_file.json", "r") as read_file:
    data = json.load(read_file)
```

- Storing

```
with open('data_file.json', 'w') as outfile:
    json.dump(data, outfile)
```

Bibligrapy

- https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-0001-introduction-to-computer-science-and-programming-in-python-fall-2016/lecture-slides-code/