# Lesson 4

Bjarne Poulsen

# Files, CLI, Exceptions and Operator Overloading.

## Running Programs from the Command Line

When you call your programs from the CLI, you can pass arguments to it. In the most basic form this looks like:

```
$ python your_program.py arg1 arg2
```

In this session, we will learn how to parse CLI arguments and options so that you can run your programs parametrized from the command-line.

**Parsing CLI Arguments**

Arguments are given -separated by spaces- after the name of your program on the CLI. Within your code, you can access them via the argv in the sys module, where argv argv[0] is the script pathname if known and all the following elements of that list are the arguments given to your program.

**Basic Logging Tutorial**

Logging is a means of tracking events that happen when some software runs. The software's developer adds logging calls to their code to indicate that certain events have occurred. An event is described by a descriptive message which can optionally contain variable data (i.e. data that is potentially different for each occurrence of the event). Events also have an importance which the developer ascribes to the event; the importance can also be called the level or severity.

https://docs.python.org/3/howto/logging.html

```
import sys
from urllib.parse import urlparse
import logging
log_fmt = '%(asctime)s - %(levelname)s - %(message)s'
logging.basicConfig(level=logging.DEBUG, format=log_fmt)

def check_args(arguments):
    # This is to be implemented in your programs...
    return True
```

```
def run(arguments):
    if check_args(arguments):
        for idx, argument in enumerate(arguments):
            logging.info('{} argument is {}'.format(idx, argument))
    else:
        print('Usage: python your_script.py arg_1 [arg_2 ...]')



if __name__ == '__main__':
    # Call me from the CLI for example with:
    # python your_script.py arg_1 [arg_2 ...]
    run(sys.argv[1:])
```

**Parsing CLI Options**

Options are given -separated by spaces- after the name of your program and after option names following dashes on the CLI. Within your code, you can parse them using getopt.getopt out of the argv in the sys module. See the following example:

https://docs.python.org/3/library/getopt.html

```
import sys
import getopt


def usage():
    return 'Usage : cli_opt_demo.py –n <name> or cli_opt_demo.py --name <name>'
```

```
def run(arguments):
    try:
        opts, args = getopt.getopt(arguments, "ho:v", ["help", "output="])
    except getopt.GetoptError as err:
        # print help information and exit:
        print(err)  # will print something like "option -a not recognized"
        usage()
        sys.exit(2)
```

```
output = None
    verbose = False
    for option, argument in opts:
        print(option)
        if option == "-v":
            verbose = True
        elif option in ("-h", "--help"):
            print(usage())
            sys.exit()
        elif option in ("-o", "--output"):
            output = argument
        else:
            assert False, "unhandled option"

    print(output)
```

Python's assert statement is a debugging aid that tests a condition.

But if the assert condition evaluates to false, it raises an AssertionError exception with an optional error message.

The proper use of assertions is to inform developers about unrecoverable errors in a program.

```
if __name__ == "__main__" :
    run(sys.argv[1:])
```

https://wiki.python.org/moin/Powerful%20Python%20One-Liners

http://docopt.org

**Piping Arguments to Your Program.**

If you want to allow your program to consume data that is piped in from other programs, then you have to let your program read input from stdin, which you can access from the module sys.

You can combine CLI arguments and piping support, as the following example illustrates. cli_replace.py replaces a match for a regular expression line by line with a replacement text. The regular expression and the replacement text are given as arguments to the program.

```
# Adapted from: https://wiki.python.org/moin/Powerful%20Python%20One-Liners

import re
import sys

if __name__ == '__main__':
    # Call me from the CLI for example with:
    # printf "uiuiui cat aiaia bumbum\naiaiai" | python cli_replace.py cat dog
    pattern = sys.argv[1]
    substitution = sys.argv[2]

    for line in sys.stdin:
        sys.stdout.write(re.sub(pattern, substitution, line))
```

PS C:\Users\bjpo1\OneDrive\Skrivebord\AdvaPro\CLI>
"uiuiui cat aiaia bumbum\naiaiai" | python xxxxx.py cat

uiuiui dog aiaia bumbum\naiaiai

CLI Arguments the Easy Way

There are quite a few libraries, which allow for parsing CLI arguments and options in a way that you do not have to write repititive code.

http://docopt.org

A CLI description language. After installing it via conda install docopt you can let the library create a parser for your CLI arguments out of a module's doc string.

## Writing to a File

One of the simplest ways to save data is to write it to a file. When you write text to a file, the output will still be available after you close the terminal containing your program's output. You can examine output after a program finishes running, and you can share the output files with others as well. You can also write programs that read the text back into memory and work with it again later.

**Writing to an Empty File**

To write text to a file, you need to call open() with a second argument telling Python that you want to write to the file. To see how this works, let's write a simple message and store it in a file instead of printing it to the screen.

The call to open() in the following example has two arguments. The first argument is still the name of the file we want to open. The second argument, 'w', tells the Python interpreter, that we want to open the file in write mode. You can open a file in read mode ('r'), write mode ('w'), append mode ('a'), or a mode that allows you to read and write to the file ('r+'). If you omit the mode argument, Python opens the file in read-only mode by default.

The open() function automatically creates the file you are writing to if it does not already exist. However, be careful opening a file in write mode ('w') because if the file does exist, Python will erase the file before returning the file object.

```
filename = '/xxxx/xx.txt'

with open(filename, 'w') as file_object:
    file_object.write(xxxxxxx)
```

Writing Multiple Lines

The write() function does not add any newlines to the text you write. So if you write more than one line without including newline characters, your file may not look the way you want it to.

```
import time
filename = '/xxx/xxx.txt'
with open(filename, 'w') as file_object:
    file_object.write(xxxxxxxx)
    time.sleep(60 * 2)
    file_object.write(yyyyyyy)
```

If you open /xxx/xxx.txt, you will see the two lines squished together. Including newlines in your write() statements makes each string appear on its own line. You can also use spaces, tab characters, and blank lines to format your output, just as you have been doing with terminal-based output.

**Appending to a File**

If you want to add content to a file instead of writing over existing content, you can open the file in append mode. When you open a file in append mode, Python does not erase the file before returning the file object. Any lines you write to the file will be added at the end of the file. If the file does not exist yet, Python will create an empty file for you.

```
filename = ‘/xxx/xxx.txt'

with open(filename, 'a') as file_object:
    file_object.write(xxxxxxx+ '\n')
    file_object.write(yyyyyyy+ '\n')
    file_object.write(zzzzzzzz + '\n')
```

## Reading from a File

An incredible amount of data is available in text files. Text files can contain weather data, traffic data, socioeconomic data, literary works, and more. Reading from a file is particularly useful in data analysis applications, but it's also applicable to any situation in which you want to analyze or modify information stored in a file. For example, you can write a program that reads in the contents of a text file and rewrites the file with formatting that allows a browser to display it. When you want to work with the information in a text file, the first step is to read the file into memory. You can read the entire contents of a file, or you can work through the file one line at a time.

```
import wget # wget.py must be created first and added to this folder.

# Download the file in case we do not have it already
url =
'https://raw.githubusercontent.com/ehmatthes/pcc/master/chapter_10/pi_30_digits.txt'
wget.download(url)

with open('pi_30_digits.txt') as file_object:
    contents = file_object.read()
    print(contents)

100% [..........................................................................] 39 / 393.1415926535
  8979323846
  2643383279
```

In this case, it closes the file once access to it is no longer needed. Notice how we call open() in this program but not close(). You could open and close the file by calling open() and close(), but if a bug in your program prevents the close() statement from being executed, the file may never close. This may seem trivial, but improperly closed files can cause data to be lost or corrupted. And if you call close() too early in your program, you'll find yourself trying to work with a closed file (a file you can't access), which leads to more errors. It is not always easy to know exactly when you should close a file, but with the structure shown here, Python will figure that out for you. All you have to do is open the file and work with it as desired, trusting that Python will close it automatically when the time is right.

The only difference between this output and the original file is the extra blank line at the end of the output. The blank line appears because read() returns an empty string when it reaches the end of the file; this empty string shows up as a blank line. If you want to remove the extra blank line, you can use rstrip().

**Storing Data with JSON files**

A simple way to persist and exchange machine readable data is using the json module.

The json module allows you to dump simple Python data structures into a file and load the data from that file the next time the program runs. You can also use json to share data between different Python programs. Even better, the JSON data format is not specific to Python, so you can share data you store in the JSON format with people who work in many other programming languages. It is a useful and portable format, and it is easy to learn.

**Using json.dump() and json.load()**

Let’s write a short program that stores a set of numbers and another program that reads these numbers back into memory. The first program will use json.dump() to store the set of numbers, and the second program will use json.load(). The json.dump() function takes two arguments: a piece of data to store and a file object it can use to store the data. Here’s how you can use json.dump() to store a list of numbers:

```
import json
numbers = {1: 'a', 2: [1,2,3]}
filename = ‘/xxxx/xxxx.json'
with open(filename, 'w') as f_obj:
    json.dump(numbers, f_obj)
```

```
{"1": "a", "2": [1, 2, 3]}
```

```
import json


filename = 'test.json'

with open(filename) as f_obj:
    de_numbers = json.load(f_obj)

print(de_numbers)
```

```
{"1": "a", "2": [1, 2, 3]}
```

```
import json

def marshal(path, data):
    with open(path, 'w') as f_obj:
        json.dump(data, f_obj)

def unmarshal(path):
    with open(path) as f_obj:
        content = json.load(f_obj)
    return content
```

```
# Some example data taken from:
# https://adobe.github.io/Spry/samples/data_region/JSONDataSetSample.html
example_data = {
    "id": "0001",
    "type": "donut",
    "name": "Cake",
    "image": {
        "url": "images/0001.jpg",
        "width": 200,
        "height": 200
    },
    "thumbnail": {
        "url": "images/thumbnails/0001.jpg",
        "width": 32,
        "height": 32
    }
}
```

```
filename = "test.json"
marshal(filename, example_data)
print(unmarshal(filename))
```

```
{'id': '0001',
 'image': {'height': 200, 'url': 'images/0001.jpg', 'width': 200},
 'name': 'Cake',
 'thumbnail': {'height': 32, 'url': 'images/thumbnails/0001.jpg', 'width': 32},
 'type': 'donut'}
```

**The CSV File Format**

One simple way to store data in a text file is to write the data as a series of values separated by commas, called comma-separated values. The resulting files are called CSV files.

CSV files are simple:

- Do not have types for their values—everything is a string
- Do not have settings for font size or color
- Do not have multiple worksheets
- Cannot specify cell widths and heights
- Cannot have merged cells
- Cannot have images or charts embedded in them

**Writing Data to CSV Files**

A Writer object lets you write data to a CSV file. To create a Writer object, you use the csv.writer() function. Enter the following into the interactive shell:

First, call open() and pass it 'w' to open a file in write mode. This will create the object you can then pass to csv.writer() to create a Writer object.

On Windows, you'll also need to pass a blank string for the open() function's newline keyword argument

The writerow() method for Writer objects takes a list argument. Each value in the list is placed in its own cell in the output CSV file. The return value of writerow() is the number of characters written to the file for that row (including newline characters). Notice how the Writer object automatically escapes the comma in the value '614,5' with double quotes in the CSV file. The csv module saves you from having to handle these special cases yourself.

```
import csv
import platform

if platform.system() == 'Windows':
    newline=''
else:
    newline=None

with open("output.csv", 'w', newline=newline) as output_file:
    output_writer = csv.writer(output_file)

    output_writer.writerow(['2015', '1', '0', '5100', '614,5'])
    output_writer.writerow(['2015', '1', '0', '5104', '2,3'])
    output_writer.writerow(['2015', '1', '0', '5106', '1'])
    output_writer.writerow(['2015', '1', '0', '5110', '1'])
```

2015,1,0,5100,"614,5"

2015,1,0,5104,"2,3"

2015,1,0,5106,1

2015,1,0,5110,1

```
import csv
with open("output.csv", 'w') as output_file:
    output_writer = csv.writer(output_file, delimiter='\t', quotechar='|')
    output_writer.writerow(['2015', '1', '0', '5100', '614\t5'])
    output_writer.writerow(['2015', '1', '0', '5104', '2,3'])
    output_writer.writerow(['2015', '1', '0', '5106', '1'])
    output_writer.writerow(['2015', '1', '0', '5110', '1’])
```

```
2015    1       0       5100    |614    5|

2015    1       0       5104    2,3

2015    1       0       5106    1

2015    1       0       5110    1
```

**Reading Data from Reader Objects in a for Loop.**

**name,department,birthday month**

**John Smith,Accounting,November**

**Erica Meyers,IT,March**

**Reading Data from Reader Objects in a for Loop**

```
import csv


with open('employee_file.txt', mode='r') as csv_file:
    csv_reader = csv.DictReader(csv_file)
    line_count = 0
    for row in csv_reader:
        if line_count == 0:
            print(f'Column names are {", ".join(row)}')
            line_count += 1
        print(f'\t{row["name"]} works in the {row["department"]} department, and
was born in {row["birthday month"]}.')
        line_count += 1
    print(f'Processed {line_count} lines.')
```

Column names are name, department, birthday month
John Smith works in the Accounting department, and was born in November.
Erica Meyers works in the IT department, and was born in March.
Processed 3 lines.

https://realpython.com/python-csv/

## Exceptions

Python uses special objects called exceptions to manage errors that arise during a program's execution. Whenever an error occurs that makes Python unsure what to do next, it creates an exception object. If you write code that handles the exception, the program will continue running. If you do not handle the exception, the program will halt and show a traceback, which includes a report of the exception that was raised.

Exceptions are handled with try-except blocks. A try-except block asks Python to do something, but it also tells Python what to do if an exception is raised. When you use try-except blocks, your programs will continue running even if things start to go wrong. Instead of tracebacks, which can be confusing for users to read, users will see friendly error messages that you write.

Let's look at a simple error that causes Python to raise an exception. You probably know that it's impossible to divide a number by zero, but let's ask Python to do it anyway:

**print(5 / 0)**

print(5/0)

ZeroDivisionError: division by zero

**Using try-except Blocks**

When you think an error may occur, you can write a try-except block to handle the exception that might be raised. You tell Python to try running some code, and you tell it what to do if the code results in a particular kind of exception.

```
try:
    print(5/0)
except ZeroDivisionError:
    print("You can't divide by zero!")
finally:
Print(“”)
```

You can't divide by zero!

**The else Block**

We can make this program more error resistant by wrapping the line that might produce errors in a try-except block. The error occurs on the line that performs the division, so that is where we put the try-except block. The following example also includes an else block***. Any code that depends on the try block executing successfully goes in the else block***.

```
import random


for i in range(0,20):
    try:
        result = random.randint(0,10) / random.randint(0,10)
    except ZeroDivisionError:
        print("Cannot divide by 0!")
    else:
        print(result)
```

0.2222222222222222
0.6666666666666666
0.0
1.75
0.0
1.6666666666666667
Cannot divide by 0!
Cannot divide by 0!
Cannot divide by 0!
2.0
7.0
0.2857142857142857
0.4444444444444444
0.4
0.0
0.8571428571428571
0.0
0.6666666666666666
7.0
Cannot divide by 0!

**Multiple Exceptions**

In case the code in your try block can throw different types of exceptions, you can catch them with multiple except blocks. In case you need access to the exception object, you can assign it to a variable via the as keyword.

```
try:
    with open('./Not_There.txt') as f_obj:
        contents = f_obj.read()
         print(5/0)
    except ZeroDivisionError as e:
         print(e)
    except FileNotFoundError as e:
         print(e)
     else:
        print("Everything went well...")
```

[Errno 2] No such file or directory: './Not_There.txt'

Alternatively, you can 'go up in the exception hierarchy' by catching all exceptions as in the following. However, you may no catch errors, that you would have liked to crash your program.

```
try:
    print(5/0)
    with open('./Not_There.txt') as f_obj:
    contents = f_obj.read()
except Exception as e:
    print(e)
else:
    print("Everything went well...")
```

division by zero

**Raising and Implementing Exceptions**

In case you need to raise your own exceptions, you can do so with the help of the raise keyword. To implement you own Exception you have to implement a subclass of the type of exception that is closest to your new type of error as illustrated in the following. ValueError. A ValueError is raised when a function receives a value that has the right type but an invalid value.

```
class NoOneValueError(ValueError):
def __init__(self, args):
        ValueError.__init__(self, args)
        value = 1
        some_data = [2, 7, 8, 10, 'aha’]
    if value in some_data:
         print('Alright!’)
    else:
         raise NoOneValueError('Oha, {} is not in {}!'.format(value, some_data))
 raise NoOneValueError('Oha, {} is not in {}!'.format(value, some_data))
__main__.NoOneValueError: Oha, 1 is not in [2, 7, 8, 10, 'aha']!
```

**Operator Overloading**

It is possible because + operator is overloaded by both int class and str class. The operators are actually methods defined in respective classes. Defining methods for operators is known as operator overloading. For e.g. To use + operator with custom objects you need to define a method called __add__.

**a = "Bjarne"**

**print(a*3)**

BjarneBjarneBjarne

**print(a / 3)**

TypeError: unsupported operand type(s) for /: 'str' and 'int'

**?**

**print(dir(a))**

['__add__', '__class__', '__contains__', '__delattr__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__', '__getitem__', '__getnewargs__', '__gt__', '__hash__', '__init__', '__init_subclass__', '__iter__', '__le__',

'__len__', '__lt__', '__mod__', '__mul__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__rmod__', '__rmul__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__', 'capitalize', 'casefold', 'center', 'count', 'encode', 'endswith', 'expandtabs', 'find', 'format', 'format_map', 'index', 'isalnum', 'isalpha', 'isascii', 'isdecimal', 'isdigit', 'isidentifier', 'islower', 'isnumeric', 'isprintable', 'isspace', 'istitle', 'isupper', 'join', 'ljust', 'lower',

'lstrip', 'maketrans', 'partition', 'replace', 'rfind', 'rindex', 'rjust', 'rpartition', 'rsplit', 'rstrip', 'split', 'splitlines', 'startswith', 'strip', 'swapcase', 'title', 'translate', 'upper', 'zfill’]

**print(a.__mul__(3))**

**BjarneBjarneBjarne**

If the behavior of a built-in function or operator is not defined in the class by the special method, then you will get a TypeError.

```
class Order:
    def __init__(self, item, customer):
      self.item = list(item)
      self.customer = customer


    def __len__(self):
      return len(self.item)
order = Order(['banana', 'apple', 'mango'], 'Real Python')
print(len(order))
```

3

But, when overloading len(), you should keep in mind that Python requires the function to return an integer.

```
def __add__(self, other):
        new_item = self.item.copy()
        new_item.append(other)
        return Order(new_item, self.customer)

order =(order + 'orange').item
print(len(order))
```
4

**Python Magic Methods Or Special Functions**

As we already there are tons of Special functions or magic methods in Python associated with each operator. Here is the tabular list of Python magic methods.

List of Assignment operators and associated magic methods.

| Binary Operators | Magic Method or Special Function |
|---|---|
| **–** | object.__sub__(self, other) |
| **+** | object.__add__(self, other) |
| ***** | object.__mul__(self, other) |
| **/** | object.__truediv__(self, other) |
| **//** | object.__floordiv__(self, other) |
| **%** | object.__mod__(self, other) |
| ****** | object.__pow__(self, other) |
| **>>** | object.__rshift__(self, other) |
| **<<** | object.__lshift__(self, other) |
| **&** | object.__and__(self, other) |
| **\|** | object.__or__(self, other) |
| **^** | object.__xor__(self, other) |

| Assignment Operators | Magic Method or Special Function |
| --- | --- |
| **-=** | object.__isub__(self, other) |
| **+=** | object.__iadd__(self, other) |
| ***=** | object.__imul__(self, other) |
| **/=** | object.__idiv__(self, other) |
| **//=** | object.__ifloordiv__(self, other) |
| **%=** | object.__imod__(self, other) |
| ****=** | object.__ipow__(self, other) |
| **>>=** | object.__irshift__(self, other) |
| **<<=** | object.__ilshift__(self, other) |
| **&=** | object.__iand__(self, other) |
| **\|=** | object.__ior__(self, other) |
| **^=** | object.__ixor__(self, other) |

| Comparison Operators | Magic Method or Special Function |
| --- | --- |
| **<** | object.__lt__(self, other) |
| **>** | object.__gt__(self, other) |
| **<=** | object.__le__(self, other) |
| **>=** | object.__ge__(self, other) |
| **==** | object.__eq__(self, other) |
| **!=** | object.__ne__(self, other) |

| Unary Operators | Magic Method or Special Function |
| --- | --- |
| **–** | object.__neg__(self) |
| **+** | object.__pos__(self) |
| **~** | object.__invert__(self) |

```
class Example:
    def __init__(self,a,b):
        self.a = a
        self.b = b

    def __str__(self):
        return "({0},{1})".format(self.a,self.b)

    def __iadd__(self,other):
        self.a += other.a
        self.b += other.b
        return self
```

**obj1 = Example(1,2)**

**obj2 = Example(2,3)**

**obj2 += obj1**

**print("obj1 =",obj1)**

**print ("obj2 =", obj2)**

obj1 = (1,2)

obj2 = (3,5)

https://docs.python.org/3/reference/datamodel.html

https://realpython.com/operator-function-overloading/

**Exercise .**

1. Select a format to write to the file.
2. A maze generation algorithm must be tested by writing to a file.
3. A maze solution algorithm must be tested by reading from the file.
4. The exceptions that you want to have in your program must
   prepared and tested.
5. Try to find which operator overloading you can use in your program.
6. This should be documented and reviewed with your review group.
7. The final solution needs to be reviewed with me.

You have 2 weeks.