Lecture 2. Writing scripts in Python

Alexander Zhigalov / Dept. of CS, University of Helsinki and Dept. of NBE, Aalto University

Outline / **overview**

- **Section 1.** Data types and basic operations
- **Section 2.** IPython console
- **Section 3.** Packages
- **Section 4.** Debugging

Motivation

- Do **not afraid** to do programming, because everyone can do it. Often, you do not need to have advance level to accomplish your tasks.
- **Improve** your skills to make programming quick and efficient.
- **Use** programming on a daily basis in your projects and tasks.
- Remember that lectures are for **YOU**, so do not hesitate to ask questions.

Section 1. Data types and basic operations

I. Data types / number, string, list

Purpose

- Use of different type of information. For instance,

```
# types
number = 10.25                      # number
string = 'lecture_2'                # string
data = [1, 0.25, [1, 2], 'types'] # list

# print
print("number: %1.2f" % (number))
print("string: %s" % (string))
print("list item [0]: %d" % (data[0]))
print("list item [1]: %f" % (data[1]))
print("list item [2]: %s" % (data[2]))
print("list item [3]: %s" % (data[3]))
```

See, “L02_variable_types.py”

- Optimization of execution time and memory allocation. For instance, np.int8 (1 byte) vs. np.float64 (8 bytes)

II. Multiple executions / for, while

Purpose

- Repeat the same piece of code many times. For instance, n = n + 1 (10 times)
- Automation, i.e., automatic handling of indexed arrays. For instance, EEG[session][trial][sample]
- Data acquisition or time-dependent experiment. For instance, acquire data each 1 ms or show stimulus each 1 second

Pitfalls

- Array indexing: loops in Python are slow. For instance, y[n] = np.log(x[n] ** 2) vs. y = np.log(x[0:100000] ** 2); second command is almost twice faster
- Infinite loop: when the stop condition is always false

II. Multiple executions / for, while

Example

```
# parameters
N = 10

# loop “for”
for n in range(0, N):
  print(‘%d’ % (n))

# loop “while”
n = 0
while n < N + 1: # do while condition is true
  print(‘%d’ % (n))
  n = n + 1

# nested loops
conditions = (‘C1’, ‘C2’, ‘C3’)
epochs = np.arange(0, 10)
channels = [1, 2, 3, 4, 5]
for condition in conditions:
  for epoch in epochs:
    for channel in channels:
```

See, “L02_loops.py”

III. Conditional execution / if, elif, else

Purpose

- Conditional handling of execution steps. For instance, special cases, exceptions, etc.
- Comparison operators: == (equal), != (not equal), > (more), >= (more or equal), …
- Logical operators: and, or, not

Pitfalls

- Self-excluding conditions. For instance, a < 0 and b > 0 and a > b

III. Conditional execution / if, elif, else

Example

```
# if, else
N = 10
for n in range(0, N):
  print("n = %d" % (n))
  if n >= 7:
    break # keyword
  elif n < 3:
    continue # keyword
  else:
    m = n * n;
  print("m = %d" % (m))
```

See, “L02_if_else.py”

IV. Functions / def

Purpose

- Compact code and better readability
- Reuse of standard functions (e.g., load data, specific preprocessing, etc.)
- Modularity of the code: easy to add new functionality and easy to test

Pitfalls

- Number of functions: there should be a balance between length of the code and the number of functions. Functions execute slower than the linear code.
- Order and types of input and output parameters: always check the usage of function. For instance, `function?`
- Naming: function should have a meaningful name. For instance, “func_1” vs. “remove_mean”, “removeMean”, “RemoveMean”

IV. Functions / def

Example

```
def add(nStudents, n):

    nStudents = nStudents + n
    return nStudents

def L02_main():

    nStudents = 15

    # print message
    print('There are %d students in the class before add()' % (nStudents))

    # call sub-function
    nStudents = add(nStudents, 5)

    # print message
    print('There are %d students in the class after add()' % (nStudents))

if __name__ == '__main__':
    L02_main()
```

See, "L02_functions.py", "L02_sub_functions.py"

V. Classes and objects / class

Purpose

- Encapsulation, i.e., properties and methods (functions) are accessible only by an object. For instance, object **Obj** from class A has access only to properties of class A, not class B. Obj = np.array((1, 2, 3)), Obj.max(), not Obj.fft()
- Inheritance, i.e., build specific classes from a class with common properties. For instance, classes **cat** and **dog** have many similar properties (e.g., name, color, etc.), and these properties can be implemented in a parent class **animal**
- Polymorphism, i.e., method has same name in all subclasses but class specific implementation. For instance, **animal** is a superclass (parent class) of subclasses **cat** and **dog**. The **animal** class has shared animal attributes (e.g., name, color, etc.) but abstract method **talk**() is implemented in the classes **cat** and **dog** differently. We can use cat.color() and cat.talk() in the same way as dog.color() and dog.talk()

Pitfalls

- Too many classes and subclasses to implement a simple data analysis approach

V. Classes and objects / class

Example (1/2)

```
class Student():

    # init
    def __init__(self, firstname, surname, ID):
        self.firstname = firstname
        self.surname = surname
        self.ID = ID
        self.grade = 0

    # print_info
    def print_info(self):
        print("%s %s, ID: %d, grade: %d" % (self.firstname, self.surname, \
                                            self.ID, self.grade)
    # get_ID
    def get_ID(self):
        return self.ID

    # set_grade
    def set_grade(self, grade):
        self.grade = grade
```

See, “L02_classes_and_objects.py”

V. Classes and objects / class

Example (2/2)

```
# create object
tStudentA = Student('Muhammad', 'Lee', 123456)
tStudentB = Student(firstname='Lena', surname='Wu', ID=234567)
tStudentC = Student(ID=345678, surname='Chang', firstname='Max')

# list of objects
tStudentList = [tStudentA, tStudentB, tStudentC]

# print info
tStudentList[1].print_info()

# set grade
tStudentList[1].set_grade(5)

# print info
tStudentList[1].print_info()
```

See, “L02_classes_and_objects.py”

VI. File operations

Purpose

- Write and read data

Pitfalls

- Unspecified file format
- Multiple access to files

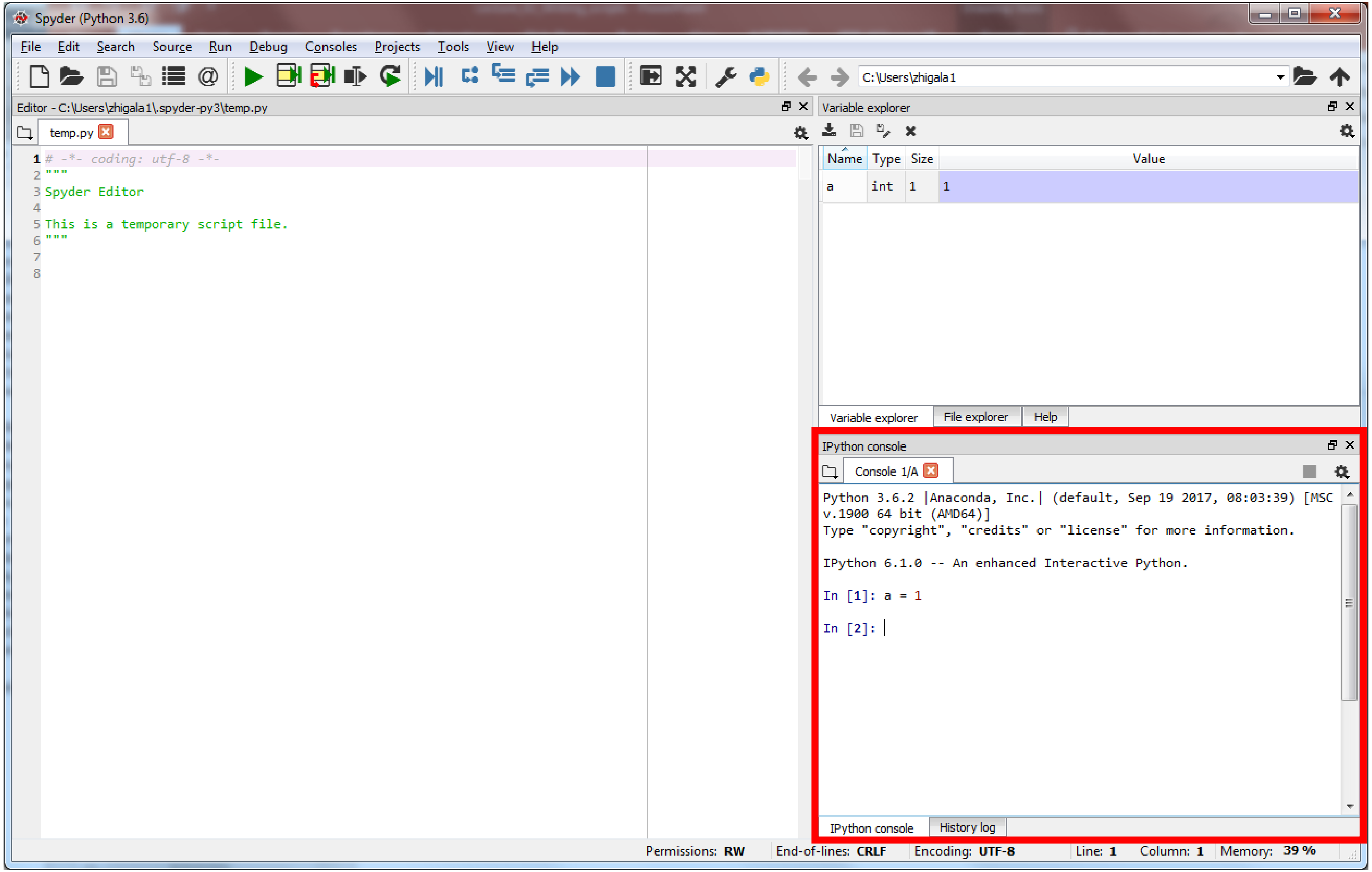
VI. File operations

Example

```
# text to write
aLine1 = 'Line 1\n'
aLine2 = 'Line 2\n'

# write text to file
hFile = open('output.txt', 'w')
hFile.write(aLine1) # write text to file
hFile.write(aLine2) # add second line
hFile.close() # close file

# read text from file
hFile = open('output.txt', 'r')
aLines = hFile.read() # read text from file
print(aLines) # print file content
hFile.close() # close file
```

See, “L02_text_files.py”

Section 2. IPython console



IPython Console

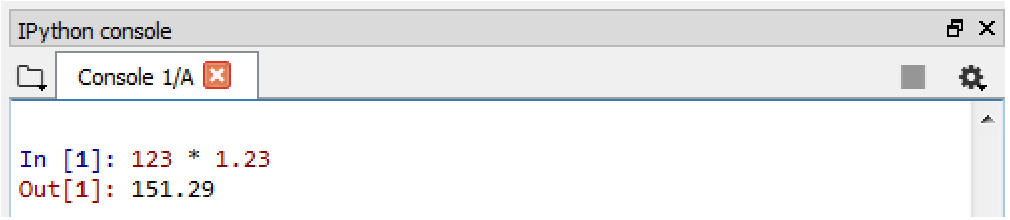
Console

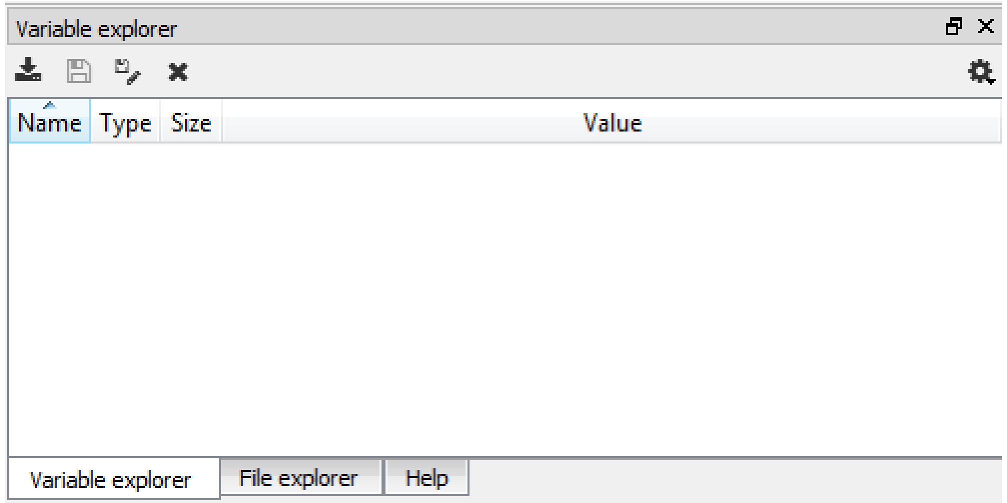
Quick and simple calculations.

Useful commands

- Restart kernel, “Ctrl + .”
- Clear variable in workspace, `del variable`
- Clear console, `clear`

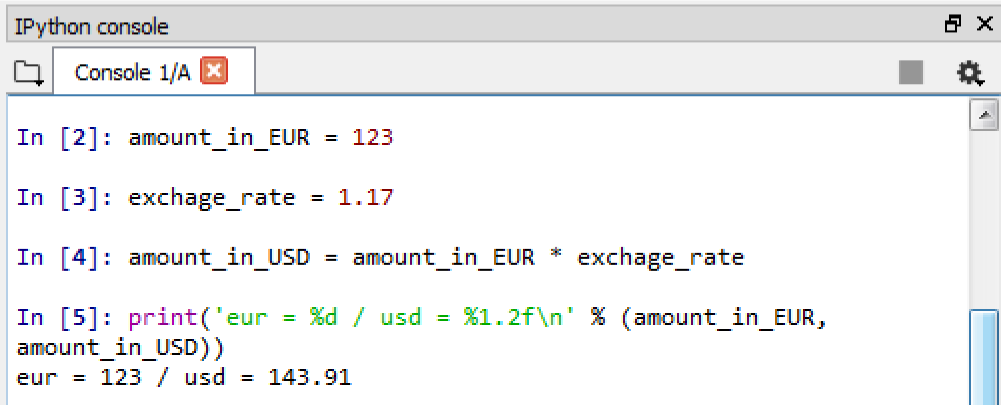
Example 1

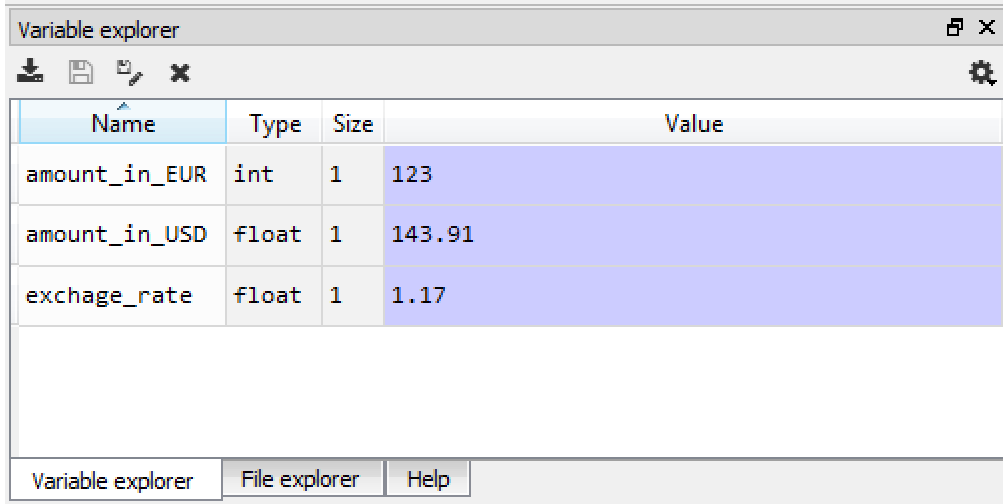




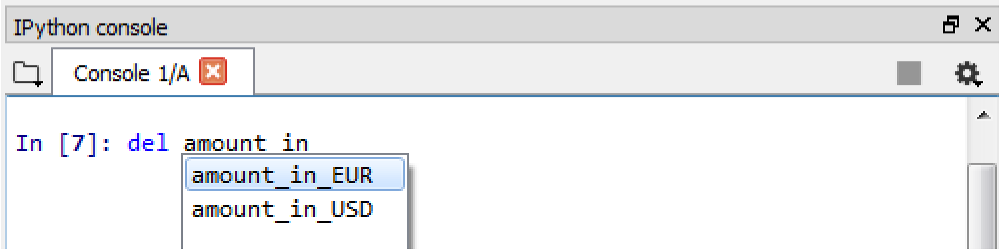
Example 2

- variable name is case sensitive, i.e., amountInEur is not the same as amountineur
- variables should be meaningful, e.g., amount_in_EUR instead of j





- button “Tab” does autocompletion



Example 3

- “import numpy as np” and “import numpy” is the same as “np.zeros(10)” and “numpy.zeros(10)”
- import matplotlib.pyplot as plt

IPython console

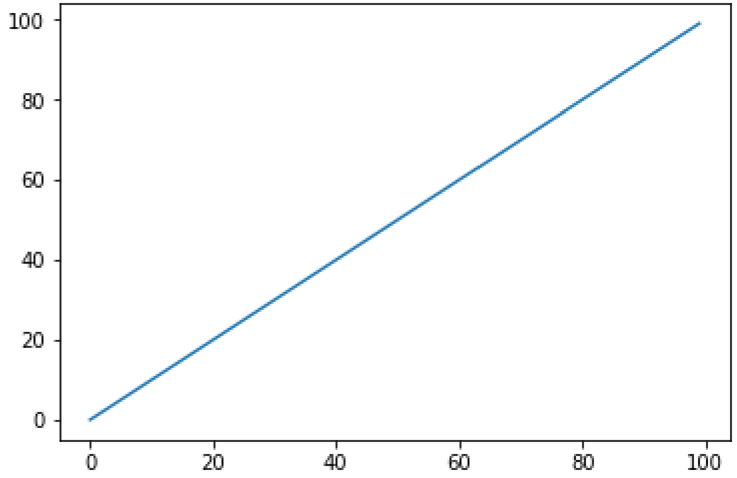
Console 1/A

```
In [12]: import numpy as np

In [13]: x = np.zeros(100)

In [14]: for i in range(0, 100):
    ...:     x[i] = i
    ...:
```

IPython console

Console 1/A

```
In [21]: import matplotlib.pyplot as plt

In [22]: plt.plot(x)
Out[22]: [<matplotlib.lines.Line2D at 0xd5d5b00>]
```



- how to get a function parameters? For instance, `np.linspace`?

Section 3. Packages

numpy

https://docs.scipy.org/doc/numpy-dev/user/quickstart.html

```
import numpy as np

# vector
a = np.array([1,2,3,4])
a = np.array((1,2,3,4))

# matrix
b = np.zeros([3,4])

# arange
b = np.arange(10,30,5) # (start, stop, step)

# random
b = np.random.random((2,3))

# reshape
c = np.arange(12).reshape(3,4)

# min, max, sum, abs
np.min(a), np.max(a), np.sum(a), np.abs()
```

```
# indexing
a = np.array([1,2,3,4])

# first item of array
a[0] = 1

# last item of array
a[3] = a[-1] = 4

# first two items: [0, 1, 2), 2 is not included
a[0:2] = a[:2]

# last two items
a[2:4] = a[-2:]

# first and third items two items
a[0:3:2] = a[::2] = a[start, stop, step]

# items in reverse order
a[-1::-1] = a[::-1]
```

scipy

https://docs.scipy.org/doc/scipy-0.18.1/reference/tutorial/index.html

```
# interpolation
from scipy.interpolate import interp1d
f = interp1d(x, y, kind='cubic') # interpolation
u = f(xnew) # interpolated data
```

See, “L02_scipy_interpolate.py”

```
# Fourier transform
from scipy.fftpack import fft
x = np.linspace(0.0, N*T, N)
y = np.sin(50.0 * 2.0 * np.pi * x) + 0.5 * np.sin(80.0 * 2.0 * np.pi * x)
u = fft(y)
```

See, “L02_scipy_interpolate.py”

```
# filter
from scipy import signal
b, a = signal.butter(4, [30.0 / (fs / 2), 60.0 / (fs / 2)], 'bandpass')
u = signal.filtfilt(b, a, y)
```

See, “L02_scipy_filter.py”

matplotlib

https://matplotlib.org/examples/

https://matplotlib.org/api/pyplot_summary.html

https://matplotlib.org/api/_as_gen/matplotlib.pyplot.plot.html#matplotlib.pyplot.plot

https://matplotlib.org/users/pyplot_tutorial.html

```
from matplotlib.pyplot import plot

plot(x, y)        # plot x and y using default line style and color
plot(x, y, 'bo')  # plot x and y using blue circle markers
plot(y)           # plot y using x as index array 0..N-1
plot(y, 'r+')     # ditto, but with red plusses
```

scikit-learn

http://scikit-learn.org/stable/

http://scikit-learn.org/stable/supervised_learning.html#supervised-learning / classification

http://scikit-learn.org/stable/modules/clustering.html#clustering / clustering

http://scikit-learn.org/stable/modules/decomposition.html#decompositions / dimensionality reduction (PCA, ICA)

```
from sklearn.svm import SVC

# create and fit model
model = SVC(kernel=‘linear’)
model.fit(X, y)

# predict
u = model.predict(Z)
u = u > 0.5

# accuracy
a = np.mean(y == u)
```

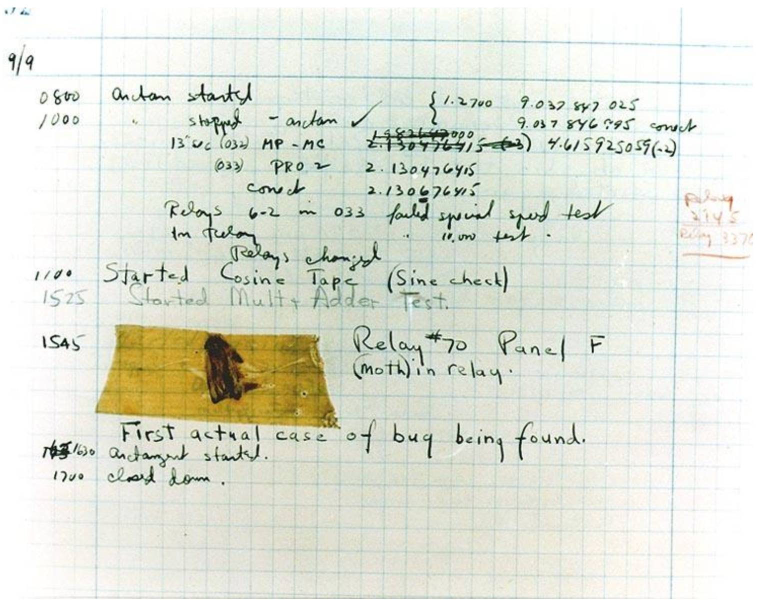
See, “L02_scipy_svm.py”

Section 4. **Debugging**

Debugging

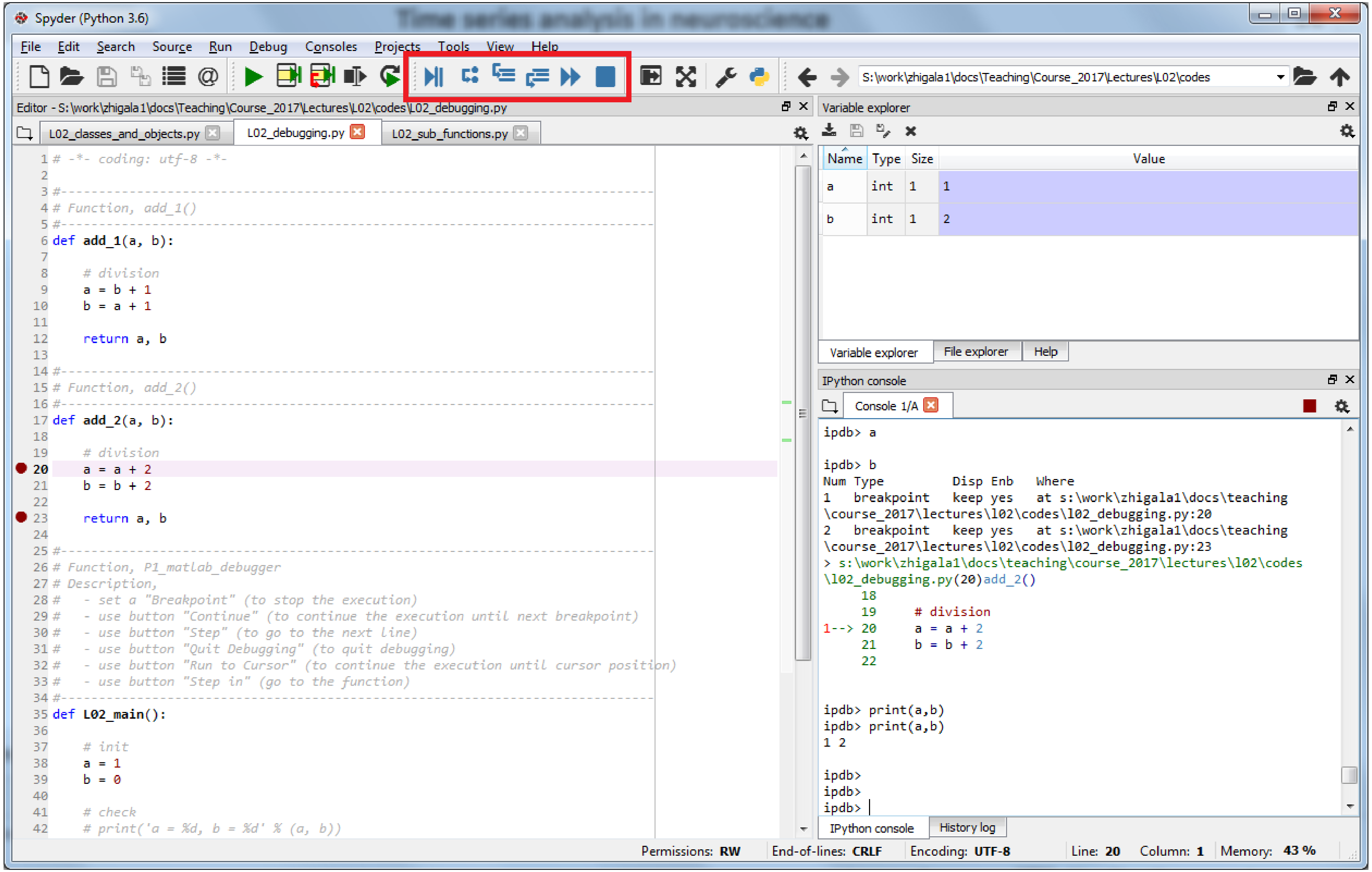
What is the debugging process?

The name **debug** comes from a real life story, a bug (moth) stuck in a relay and thereby impeding operation.



Why is it important?

Program should handle errors / report all possible problems (warnings), especially, at the development stage.


Spyder (Python 3.6)
File Edit Search Source Run Debug Consoles Projects Tools View Help
S:\work\zhigala1\docs\Teaching\Course_2017\Lectures\L02\codes
Editor - S:\work\zhigala1\docs\Teaching\Course_2017\Lectures\L02\codes\L02_debugging.py
L02_classes_and_objects.py
L02_debugging.py
L02_sub_functions.py
-*- coding: utf-8 -*-
#---
Function, add_1()
#---
def add_1(a, b):
# division
a = b + 1
b = a + 1
return a, b
#---
Function, add_2()
#---
def add_2(a, b):
# division
a = a + 2
b = b + 2
return a, b
#---
Function, P1_matlab_debugger
Description,
- set a "Breakpoint" (to stop the execution)
- use button "Continue" (to continue the execution until next breakpoint)
- use button "Step" (to go to the next line)
- use button "Quit Debugging" (to quit debugging)
- use button "Run to Cursor" (to continue the execution until cursor position)
- use button "Step in" (go to the function)
#---
def L02_main():
# init
a = 1
b = 0
# check
# print('a = %d, b = %d' % (a, b))
Variable explorer
Name Type Size Value
a int 1 1
b int 1 2
Variable explorer File explorer Help
IPython console
Console 1/A
ipdb> a
ipdb> b
Num Type Disp Enb Where
1 breakpoint keep yes at s:\work\zhigala1\docs\teaching\course_2017\lectures\l02\codes\l02_debugging.py:20
2 breakpoint keep yes at s:\work\zhigala1\docs\teaching\course_2017\lectures\l02\codes\l02_debugging.py:23
> s:\work\zhigala1\docs\teaching\course_2017\lectures\l02\codes\l02_debugging.py(20)add_2()
18
19 # division
1--> 20 a = a + 2
21 b = b + 2
22
ipdb> print(a,b)
ipdb> print(a,b)
1 2
ipdb>
ipdb>
ipdb>
IPython console History log
Permissions: RW End-of-lines: CRLF Encoding: UTF-8 Line: 20 Column: 1 Memory: 43 %

Literature

- **Python programming language**

- Downey A., “**Think Python**” (http://greenteapress.com/wp/think-python-2e/), see “materials/L02_thinkpython2.pdf”
- http://www.scipy-lectures.org/, see “materials/L02_ScipyLectures.pdf”
- Beazley and Jones, “**Python Cookbook**”
- Bressert E., “**SciPy and NumPy: An Overview for Developers**”
- http://science-it.aalto.fi/scip/pysc-fall-2016/, “**Python for scientific computing**” (slides)