

# CLASS 13: PYTHON SCRIPTS

ENGR 102 – Introduction to Engineering

2

# Python Scripts - Modules

# Spyder Console

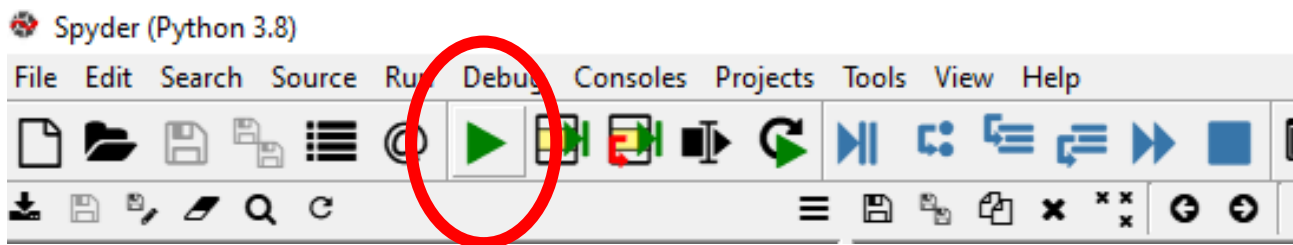
3

- As we've seen, we can execute Python commands through the **console**
  - ▣ Useful for quick calculations, debugging, etc.
  - ▣ Enter one expression at a time
  - ▣ To execute a sequence of commands repeatedly, must re-enter all commands each time
  - ▣ Command history is only record of executed commands
- Better practice is to write all commands to be executed in a single file, **script**, or **module**

# Python Scripts

4

- ***Scripts*** or ***modules*** or ***programs*** are files containing a series of Python commands
  - .py filename extension
  - Quickly and easily re-run at any time – no need to re-type all commands in the command window
  - Execute in Spyder by clicking the ***Run*** button (or ***F5***)



- Our primary mode of executing Python code

# Scripts vs. Programs vs. Modules

5

- We'll use the terms *scripts* or *programs* interchangeably when referring to Python files
- Technically, they are scripts, but this distinction is not important for our purposes.
- **Programs**
  - Written (possibly) in a high-level language – *source code*
  - ***Compiled*** (once) by a ***compiler*** into a ***machine language*** executable file – ***object code***
  - Fast, because compilation performed once, ahead of runtime
- **Scripts**
  - High-level source code is ***interpreted*** and executed line-by-line by an ***interpreter*** at runtime
  - Slower than compiled programs
- **Modules**
  - Python ***scripts*** that are intended to be ***imported*** into other scripts or modules

# Python Scripts – Best Practices

6

Start scripts with a comment listing the file name.

Additional comments with a brief overall script description and other details is useful.

Define variables to be used in equations. Parameters can be changed in a single place.

- Keep your code **DRY**:  
**Don't Repeat Yourself**

```
1  # rc_resp_ex.py
2  # kwebb 07/06/21
3  # plot the transient response of an RC circuit
4
5  import matplotlib.pyplot as plt
6  import numpy as np
7
8  # define circuit parameters
9  R = 1e3      # resistor [ohms]
10 C = 10e-9   # capacitor [F]
11 tau = R*C   # time constant
12 Vi = -1     # initial voltage
13 Vf = 2      # final voltage
14
15 # create time vector spanning 10 time constants
16 t = np.linspace(0, 10*tau, 1000)
17
18 # calculate response
19 vo = Vf + (Vi - Vf)*np.exp(-t/tau)
20
21 # plot the output voltage
22 plt.figure(1)
23 plt.plot(t/1e-6, vo, label='$v_o(t)$')
24 plt.xlabel('time    [$\mu$ sec]')
25 plt.ylabel('$v_o(t)$    [V]')
26 plt.grid()
27 plt.xlim((0, 100))
28 plt.title('RC Circuit Response')
29
```

Thoroughly comment your code.

# Comments

7

- Comments are explanatory or descriptive text added to your code
  - ▣ Not executed commands
- In Python, comments are preceded by the hash mark: #
- Comments may occupy an entire line
- Or, may be inserted at the end of a line, after uncommented expressions
- Ctrl+1 comments and uncomments a line of text in the Spyder editor
- Commenting is useful for temporarily removing instructions from a script

```
7
8 # define circuit parameters
9 R = 1e3 # resistor [ohms]
10 C = 10e-9 # capacitor [F]
11 tau = R*C # time constant
12 Vi = -1 # initial voltage
13 Vf = 2 # final voltage
14
15 # create time vector spanning 10 time constants
16 t = np.linspace(0, 10*tau, 1000)
17
```

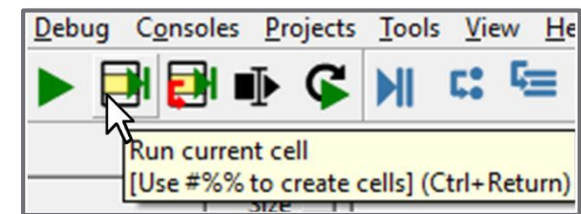
# Cells

8

- Can divide Spyder scripts into **cells**
  - ▣ Code blocks that can be executed at once, without running the entire script
- Cells are defined with a special comment line:
  - ▣ Follow the hash mark, #, with two percent signs, %%
  - ▣ Can also include comment text
    - # %% start of a cell
  - ▣ Cell ends at the start of the next cell
- To run a cell:
  - ▣ Place the cursor in the cell to be run
  - ▣ Ctrl-Enter, or click 'Run current cell'

```
# %% example 2: counter-based using range()
rng = np.random.default_rng()
print('\n')
for i in range(10):
    x = rng.uniform(low=0, high=1)
    print('x = {:.4f}'.format(x))

# %% example 3: find max value in an array, use enumerate()
x = rng.integers(0, 100, 10)
xmax = x[0]
imax = 0
for i, xval in enumerate(x[1:]):
    if xval > xmax:
        xmax = xval
        imax = 1
print('\nx = ', x)
print('\nxmax: x[{:d}] = {:d}'.format(i, xmax))
```





# Pseudocode

- The most important part of the process of writing computer code is ***planning***
  - ▣ Determine exactly what the program should do
  - ▣ And, how it will do it
  
- Before writing any code, write a ***step-by-step description*** of the program
  - ▣ ***Natural language***
  - ▣ ***Graphical*** – flow chart
  
- This may be referred to as ***pseudocode***

# Programming Process

10

## □ Programming process:

---

### □ *Define the problem*

- Ensure you have a complete understanding of the problem
- Determine exactly what the program should do
  - Inputs and outputs
  - Relevant equations

### □ *Design the program*

- *Pseudocode* – language-independent
- 

### □ *Write the program*

- Simple translation from pseudocode
- 

### □ *Validate the program*

- Do the outputs make sense
- Test with inputs that yield known outputs
- Test thoroughly – try to break it

# Pseudocode

11

- Comments can serve as pseudocode
  - Write the comments first
  - Then insert code to do what the comments say
- For example:

```
1 # max_pow_ex.py
2 #
3 # This script calculates the theoretical maximum
4 # power generated by a hydropower facility with
5 # a user-specified head and flow rate
6
7 # define physical constants
8     # density of water
9     # gravitational acceleration
10
11 # prompt user to enter the amount of head [m]
12
13 # prompt user to enter the flow rate [m^3/s]
14
15 # calculate the maximum power
16
17 # display the result
18
19
```

```
1 # max_pow_ex.py
2 #
3 # This script calculates the theoretical maximum
4 # power generated by a hydropower facility with
5 # a user-specified head and flow rate
6
7 # define physical constants
8 rho = 1000 # density of water
9 g = 9.81 # gravitational acceleration
10
11 # prompt user to enter the amount of head [m]
12 h = input('Enter the head [m]: ')
13 h = float(h)
14
15 # prompt user to enter the flow rate [m^3/s]
16 Q = input('Enter the flow rate [m^3/s]: ')
17 Q = float(Q)
18
19 # calculate the maximum power
20 pmax = rho*g*h*Q
21
22 # display the result
23 print('\nMax. Power = {} MW'.format(pmax/1e6))
24
```

# Sequential Code Execution

12

- In general code is executed line-by-line ***sequentially*** from the top of an m-file down
- There are, however, very important ***non-sequential code structures***:
  - ***Conditional statements*** – code that is executed only if certain conditions are met
    - if
    - if ... else
    - if ... elif ... else
  - ***Loops*** – code that is repeated a specified number of times or while certain conditions are met
    - for
    - while

13

# Inputs & Outputs

# Inputs to Scripts

14

- Inputs to a script:
  - ▣ Assignments of variable values
  
- Several input methods:
  - ▣ Within the script
  - ▣ From external files (.csv, Excel, etc.) – more later
  - ▣ Specified by user during execution – `input()`

# User-Specified Input – `input()`

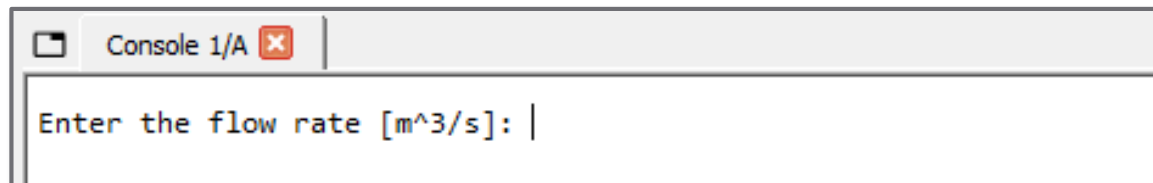
15

- Prompt user for value to be assigned to a variable

```
var = input(Prompt)
```

- *Prompt*: a *string* that will be displayed in the console, prompting the user for an input
- *var*: **string** variable to which the user-specified input is stored
  - Re-cast for different data types (e.g. float)
- For example:

```
15 # prompt user to enter the flow rate [m^3/s]
16 Q = input('Enter the flow rate [m^3/s]: ')
17 Q = float(Q)
```



Console 1/A

```
Enter the flow rate [m^3/s]: |
```

# Outputs from Scripts

16

- Outputs from scripts:
  - Display of values calculated by the script
- Several output methods
  - Plotting (more later)
  - In the console
    - `print()`
  - Writing data to files (more later)



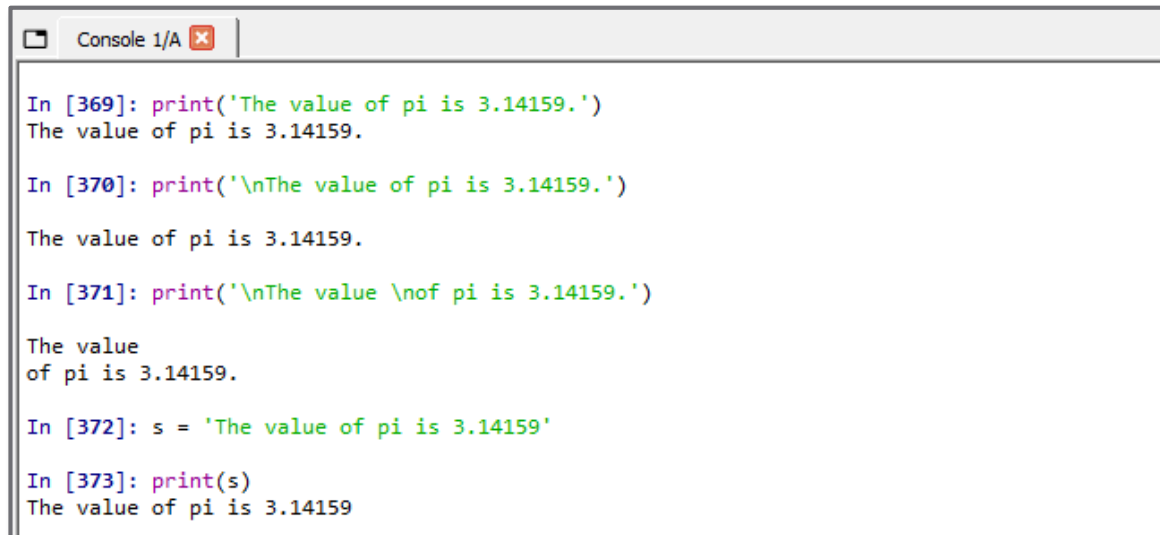
# print()

17

- Output a string to the console

```
print(string)
```

- *string*: a *string* – may contain **formatting sequences** for insertion of variable values
- For example:



```
Console 1/A x  
In [369]: print('The value of pi is 3.14159.')  
The value of pi is 3.14159.  
  
In [370]: print('\nThe value of pi is 3.14159.')  
  
The value of pi is 3.14159.  
  
In [371]: print('\nThe value \nof pi is 3.14159.')  
  
The value  
of pi is 3.14159.  
  
In [372]: s = 'The value of pi is 3.14159'  
  
In [373]: print(s)  
The value of pi is 3.14159
```

# Formatting Strings – .format()

18

- Insert formatted numbers and strings into a string

```
<template>.format(args)
```

- *<template>*: a string containing **replacement fields** for insertion of variable values
    - Replacement fields may include **formatting specifications**
  - *args*: objects to be inserted into the *<template>* string
    - Strings or numeric values
- For example:

```
Console 1/A x
In [379]: s = 'The value of {} is {}'.format('pi', 3.14159)
In [380]: print(s)
The value of pi is 3.14159
In [381]: s = 'The value of {} is {}'.format('pi', np.pi)
In [382]: print(s)
The value of pi is 3.141592653589793
```

# .format() – Syntax & Terminology

19

```
<template>.format(args)
```

- .format() is a **method** applied to the **object**, <template>, which is an **instance** of the **class** str
  - **Class**: a template for creating **objects**
    - For now, think of this as the **data type**
    - Here, the class is **string**, str
    - Classes have **attributes** and **methods** associated with them
  - **Object**: an instance of a class
    - On the previous page, s is an object of type str
  - **Method**: a function associated with a specific class
    - Here, format() is a method that operates on str objects
- These **object-oriented programming** concepts will be covered in detail later in the course

# Formatting Strings – Replacement Fields

20

## □ Replacement fields:

- ▣ Enclosed in curly brackets, {}

```
In [379]: s = 'The value of {} is {}'.format('pi', 3.14159)
```

- ▣ Arguments in `format()` are inserted *in order*
- ▣ May include a **formatting specification**, `format_spec`

{:format\_spec}

- ▣ `format_spec`: specifies how to format numeric values

```
In [389]: s = 'The value of {} is {:.3f}'.format('pi', np.pi)
```

```
In [390]: print(s)  
The value of pi is 3.142
```

# Formatting Strings – `format_spec`

21

## □ `format_spec`:

- Specify how numeric values are formatted

`: [width][group][.prec][type]`

- Always start `format_spec` with a colon, `:`
- **width**: minimum width of the field into which the argument is inserted – may result in white space
- **group**: grouping character for each three digits to the left of the decimal point (e.g. `,` or `_`)
- **.prec**: number of digits after the decimal point for floating point numbers, or maximum field width for strings
- **type**: presentation type, e.g. floating point, integer, string, etc.

# format\_spec – type

22

- Type characters specify how to format variable values within a string

Presentation Type	Type Character
Decimal integer	d
Binary integer	b
Hexadecimal integer	x
Floating point	f or F
Exponential notation (e.g., 1.6e-19 or 1.6E-19)	e or E
More compact of %e or %f	g
More compact of %E or %F	G
Single character	c
String	s
Percentage	%

# format\_spec – Examples

23

□ Fixed-point notation

□ Field-width control

□ Exponential notation

□ Compact format

- Note that .prec specifies number of significant figures for g or G type

```
Console 1/A x
In [468]: x = 10e4 * np.pi/2
In [469]: print('\n\tx = {:.2f}'.format(x))
           x = 157079.63
In [470]: print('\n\tx = {:15.2f}'.format(x))
           x =          157079.63
In [471]: print('\n\tx = {:.2e}'.format(x))
           x = 1.57e+05
In [472]: print('\n\tx = {:.2E}'.format(x))
           x = 1.57E+05
In [473]: print('\n\tx = {:.2g}'.format(x))
           x = 1.6e+05
In [474]: print('\n\tx = {:.2G}'.format(x))
           x = 1.6E+05
```