# SECTION 1: INTRODUCTION

# Course Overview

# What is Programming?

- **Programming**
  - The implementation of ***algorithms*** in a particular computer ***programming language*** for execution on a ***computer***

---

- **Algorithm**
  - A step-by-step procedure for performing a computation, solving a problem, performing some action, etc. – a recipe
  - ***Algorithm design*** is the meat of programming – the rest is just translation into a particular language
- **Programming language**
  - We'll use Python. Others include C, C++, Java, MATLAB, etc.
- **Computer**
  - May be a PC, or may be a microcontroller, FPGA, etc.

# Why Programming?

- I don’t want to be a *software* engineer. Why do I need to learn to program?
  - ***All*** engineers will need to write computer code throughout their careers
    - Design and simulation
    - Numerical solution of mathematical problems
    - Data analysis – from measurements or simulation
    - Firmware for the control of mechatronic systems
  - More importantly: ***development of algorithmic thinking ability***
    - Learn to think like an engineer – single most important takeaway from your engineering education

# Course Overview

- Section 1: Introduction
- Section 2: Vectors and Matrices
- Section 3: Two-Dimensional Plotting
- Section 4: Algorithmic Thinking & Flow Charts
- Section 5: Structured Programming in MATLAB
- Section 6: User-Defined Functions
- Section 7: Three-Dimensional Plotting
- Section 8: File I/O
- Section 9: Engineering Applications

Introductory material:

- Course overview
- Introduction to required tools
- Linear algebra basics

Package-specific tools (matplotlib):

- Data visualization
- Valuable engineering tools

Algorithm fundamentals:

- Generic; Platform-independent
- Engineering thinking – transcends programming

Application of the fundamentals:

- Python-specific, but
- Similar to other languages

# Python

- This a course in ***programming fundamentals*** and ***algorithmic thinking***
- The language we'll use to develop these concepts is ***Python*** (in the ***Spyder*** development environment)
  - Could just as well use another language, e.g., C, C++, Java, MATLAB, Fortran, …
  - The important concepts are not language-specific
- ***Two goals*** of this course:
  - Learn to develop basic algorithms and to write structured computer code
  - Learn to use Python

# Introduction to Python & Spyder

The remainder of this section of notes is intended to provide a brief introduction to Python and the Spyder development environment.

# Python – What is It?

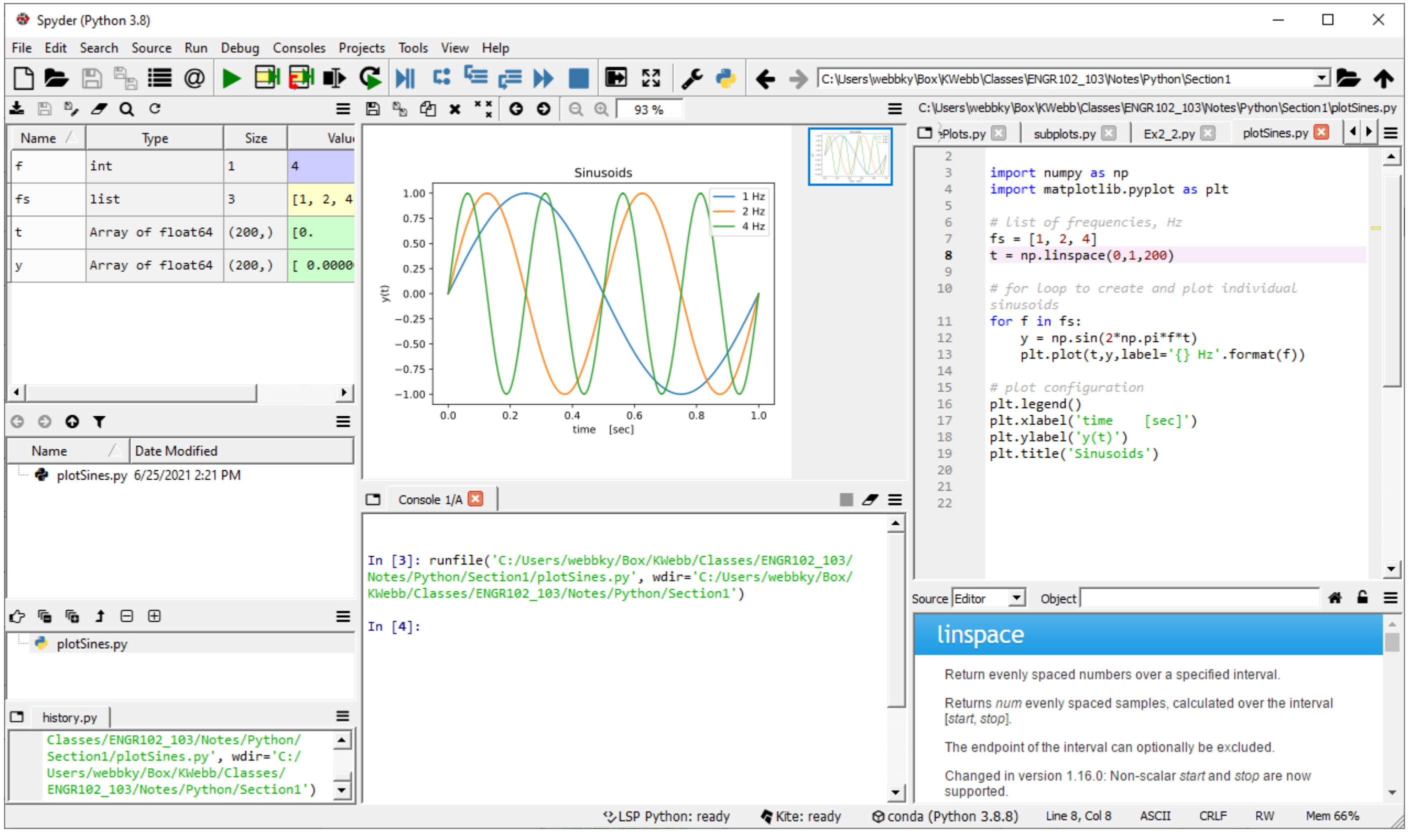
- A ***general-purpose programming language***
  - Used for writing programs to describe procedures to be executed by computers
  - *High-level*
    - Readable code – includes natural-language constructs
    - Makes use of extensive libraries of functions
    - Highly abstracted from the machine-level instructions that will ultimately be passed to the computer
  - *Interpreted*
    - Translation to machine instructions happens at runtime
    - Not *compiled* – translations happens once, creating a separate executable file
  - *Object oriented* – more on this later

# Python – How Do We Use it?

- Different ways to write and execute Python code
  - ***Text editor***
    - Simple editor for writing code
    - May include language specific formatting/coloring, etc.
    - E.g. Vi/Vim, Sublime Text, etc.
  - ***Integrated development environment (IDE)***
    - Software interface to facilitate code development
      - Code editor
      - Debugger
      - Console
      - Variable explorer
      - File browser,
      - Plotting support, etc.
    - E.g. Spyder, Pycharm, IDLE, Visual Studio, etc.

# Spyder – What is It?

- We will use the ***Spyder*** IDE
  - Scientific **PY**thon **D**evelopment **E**nvi**R**onment
  - Designed for scientific, engineering, and data science applications



| Name | Type | Size | Valu |
|---|---|---|---|
| f | int | 1 | 4 |
| fs | list | 3 | [1, 2, 4 |
| t | Array of float64 | (200,) | [0. |
| y | Array of float64 | (200,) | [ 0.0000 |

```python

import numpy as np
import matplotlib.pyplot as plt

# list of frequencies, Hz
fs = [1, 2, 4]
t = np.linspace(0,1,200)

# for loop to create and plot individual sinusoids
for f in fs:
    y = np.sin(2*np.pi*f*t)
    plt.plot(t,y,label='{} Hz'.format(f))

# plot configuration
plt.legend()
plt.xlabel('time    [sec]')
plt.ylabel('y(t)')
plt.title('Sinusoids')



```

```
In [3]: runfile('C:/Users/webbky/Box/KWebb/Classes/ENGR102_103/
Notes/Python/Section1/plotSines.py', wdir='C:/Users/webbky/Box/
KWebb/Classes/ENGR102_103/Notes/Python/Section1')

In [4]:
```

```
Classes/ENGR102_103/Notes/Python/
Section1/plotSines.py', wdir='C:/
Users/webbky/Box/KWebb/Classes/
ENGR102_103/Notes/Python/Section1')
```

**linspace**

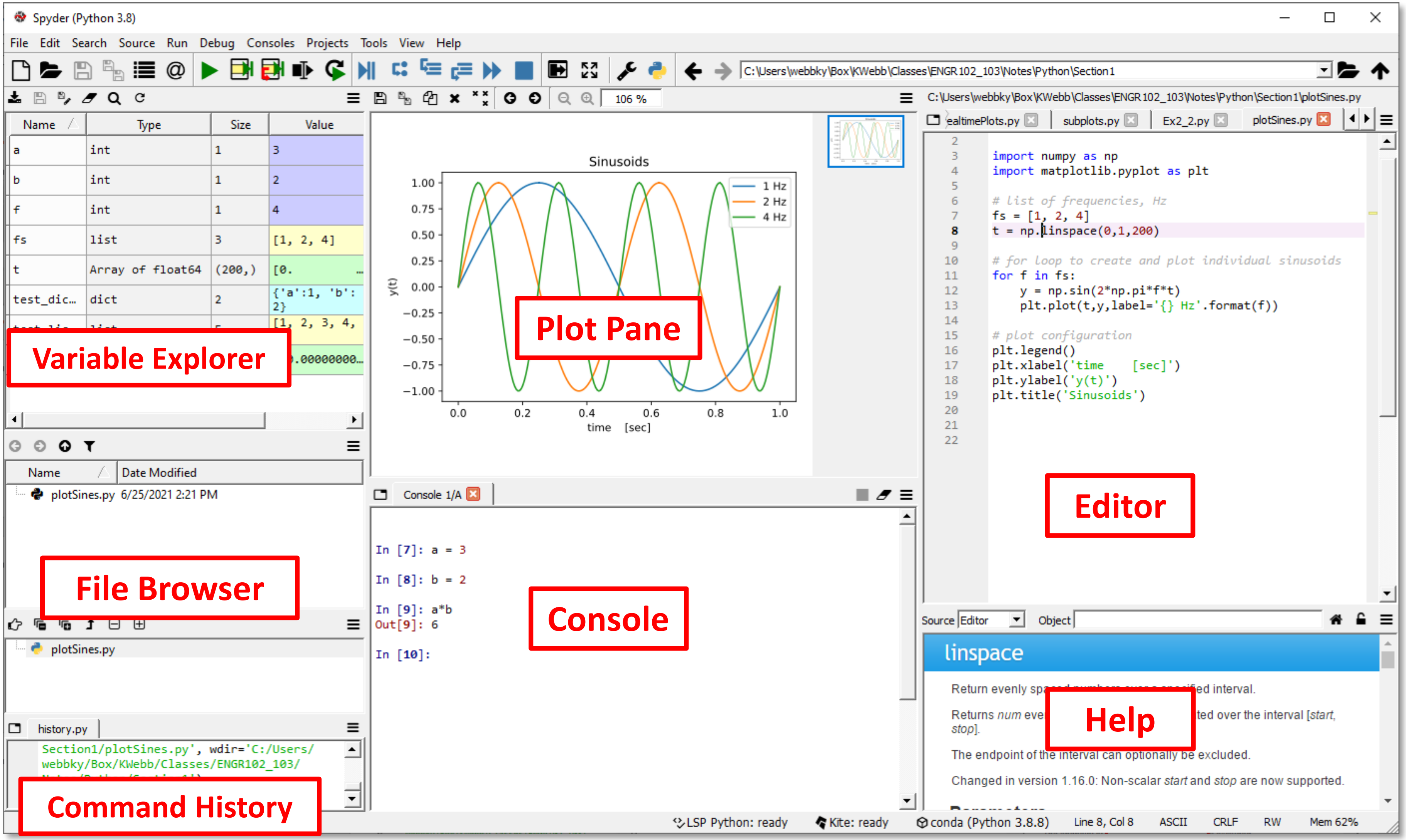
Return evenly spaced numbers over a specified interval.

Returns *num* evenly spaced samples, calculated over the interval [*start*, *stop*].

The endpoint of the interval can optionally be excluded.

Changed in version 1.16.0: Non-scalar *start* and *stop* are now supported.

# The Spyder Interface



Variable Explorer

| Name | Type | Size | Value |
|---|---|---|---|
| a | int | 1 | 3 |
| b | int | 1 | 2 |
| f | int | 1 | 4 |
| fs | list | 3 | [1, 2, 4] |
| t | Array of float64 | (200,) | [0. … |
| test_dic… | dict | 2 | {'a':1, 'b': 2} |
|  |  |  | [1, 2, 3, 4, |
|  |  |  | .00000000… |

File Browser

plotSines.py 6/25/2021 2:21 PM

plotSines.py

Command History

```
Section1/plotSines.py', wdir='C:/Users/
webbky/Box/KWebb/Classes/ENGR102_103/
```

Plot Pane

Console

```
In [7]: a = 3

In [8]: b = 2

In [9]: a*b
Out[9]: 6

In [10]:
```

Editor

```python

import numpy as np
import matplotlib.pyplot as plt

# list of frequencies, Hz
fs = [1, 2, 4]
t = np.linspace(0,1,200)

# for loop to create and plot individual sinusoids
for f in fs:
    y = np.sin(2*np.pi*f*t)
    plt.plot(t,y,label='{} Hz'.format(f))

# plot configuration
plt.legend()
plt.xlabel('time    [sec]')
plt.ylabel('y(t)')
plt.title('Sinusoids')



```

Help

linspace

Return evenly spa ed interval.

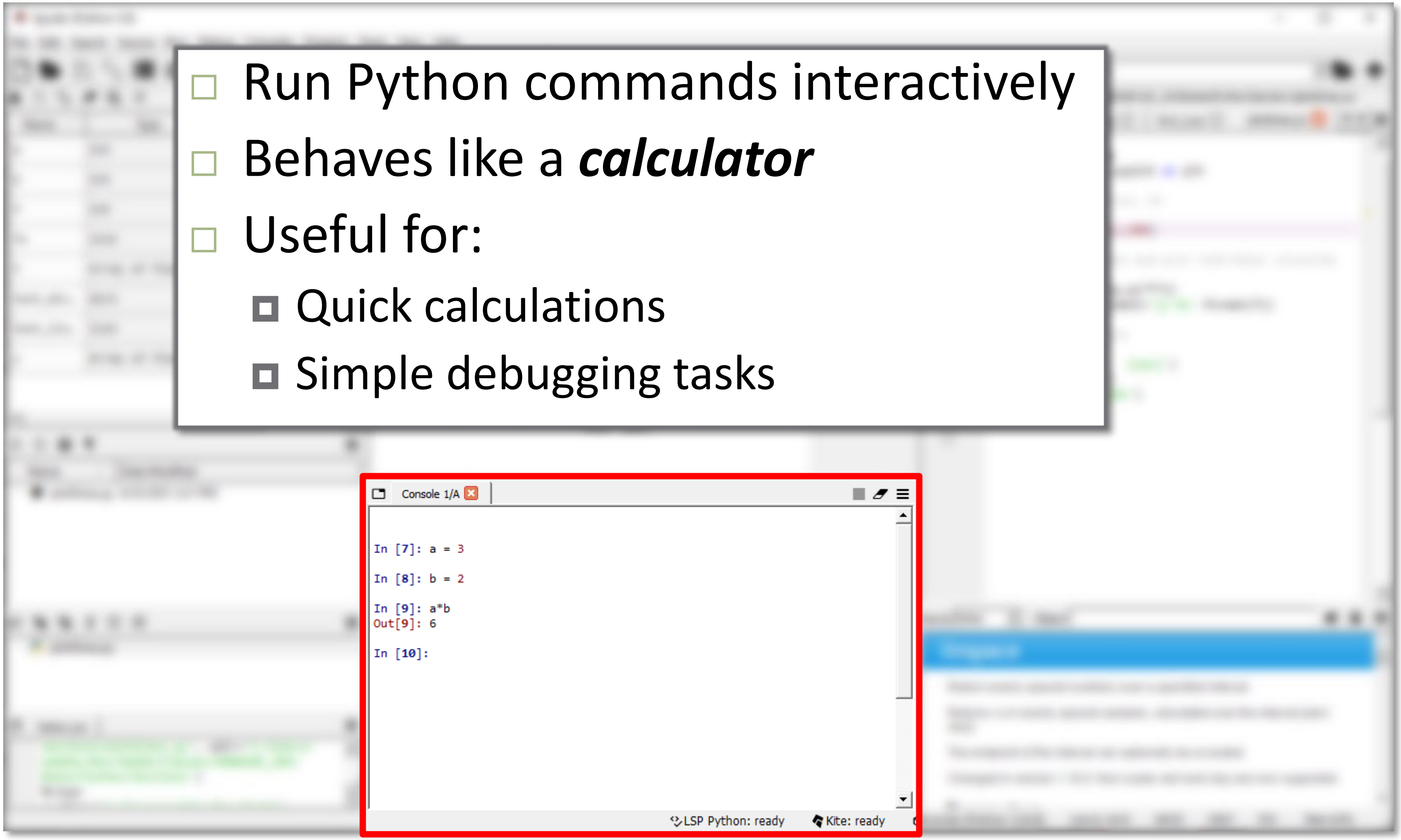
Returns *num* eve ted over the interval [*start*, *stop*].

The endpoint of the interval can optionally be excluded.

Changed in version 1.16.0: Non-scalar *start* and *stop* are now supported.

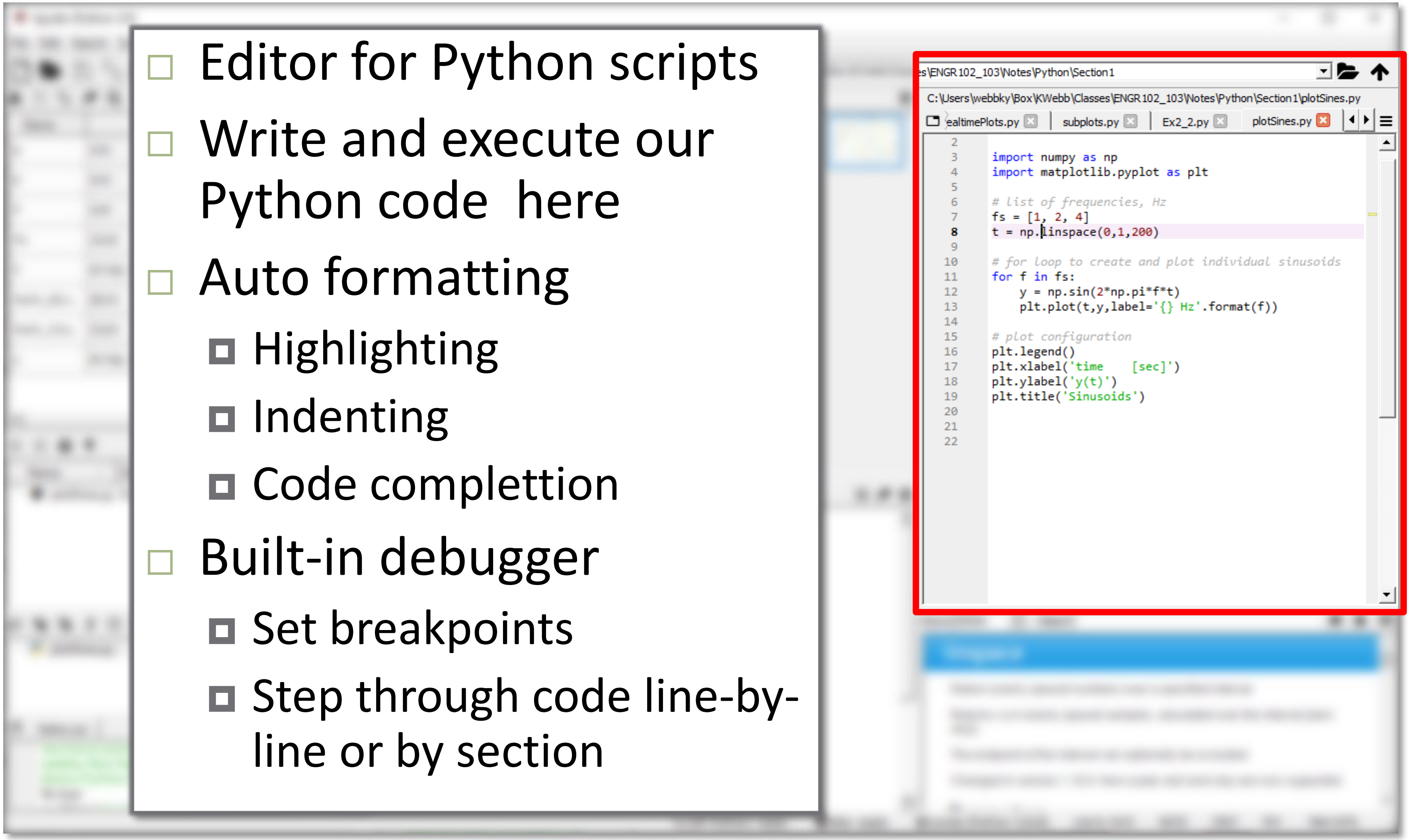
# The Spyder Interface - Console

- Run Python commands interactively
- Behaves like a ***calculator***
- Useful for:
  - Quick calculations
  - Simple debugging tasks



```
In [7]: a = 3

In [8]: b = 2

In [9]: a*b
Out[9]: 6

In [10]:
```

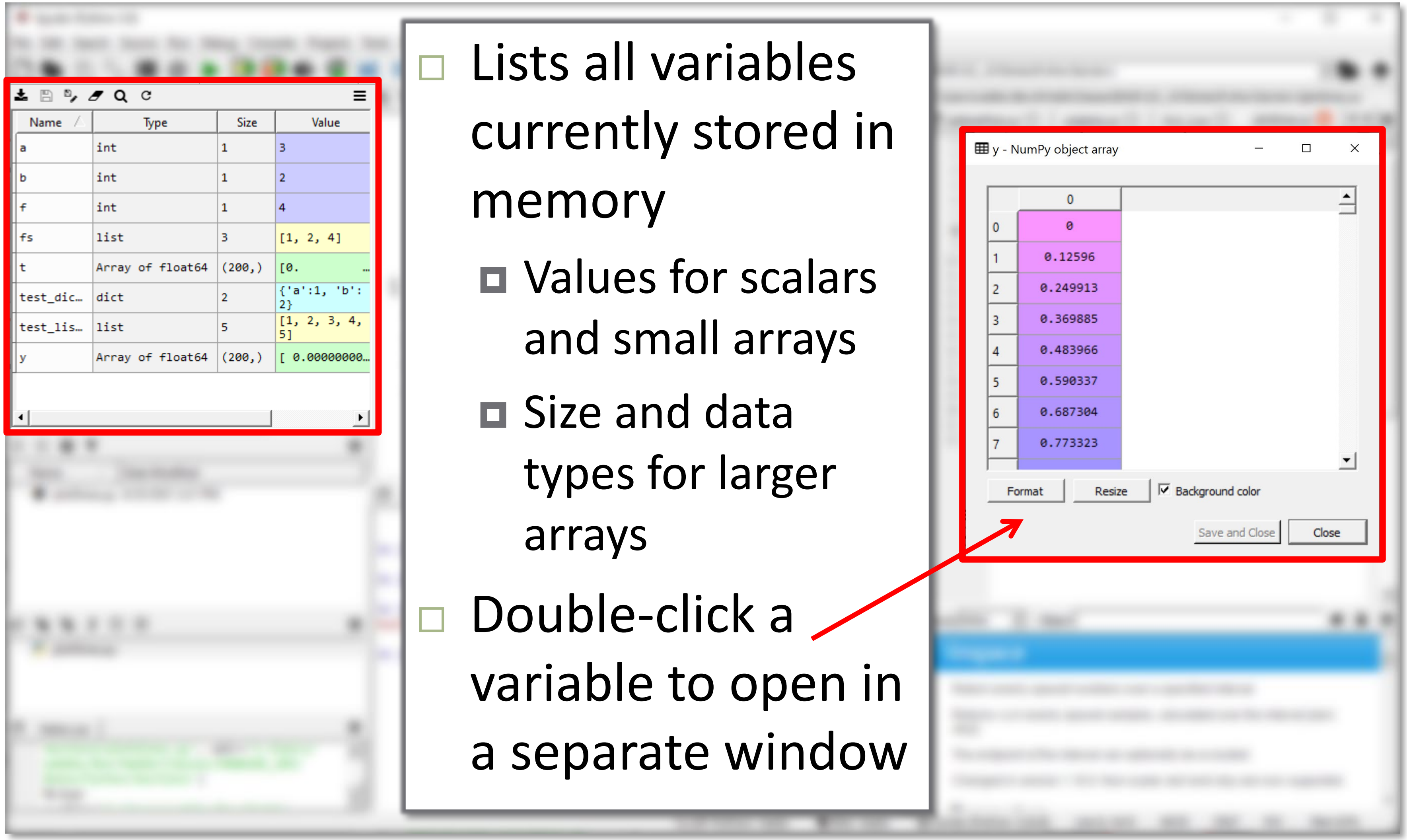
# The Spyder Interface - Editor



- Editor for Python scripts
- Write and execute our Python code here
- Auto formatting
  - Highlighting
  - Indenting
  - Code complettion
- Built-in debugger
  - Set breakpoints
  - Step through code line-by-line or by section

```python

import numpy as np
import matplotlib.pyplot as plt

# list of frequencies, Hz
fs = [1, 2, 4]
t = np.linspace(0,1,200)

# for loop to create and plot individual sinusoids
for f in fs:
    y = np.sin(2*np.pi*f*t)
    plt.plot(t,y,label='{} Hz'.format(f))

# plot configuration
plt.legend()
plt.xlabel('time    [sec]')
plt.ylabel('y(t)')
plt.title('Sinusoids')



```

# The Spyder Interface – Variable Explorer



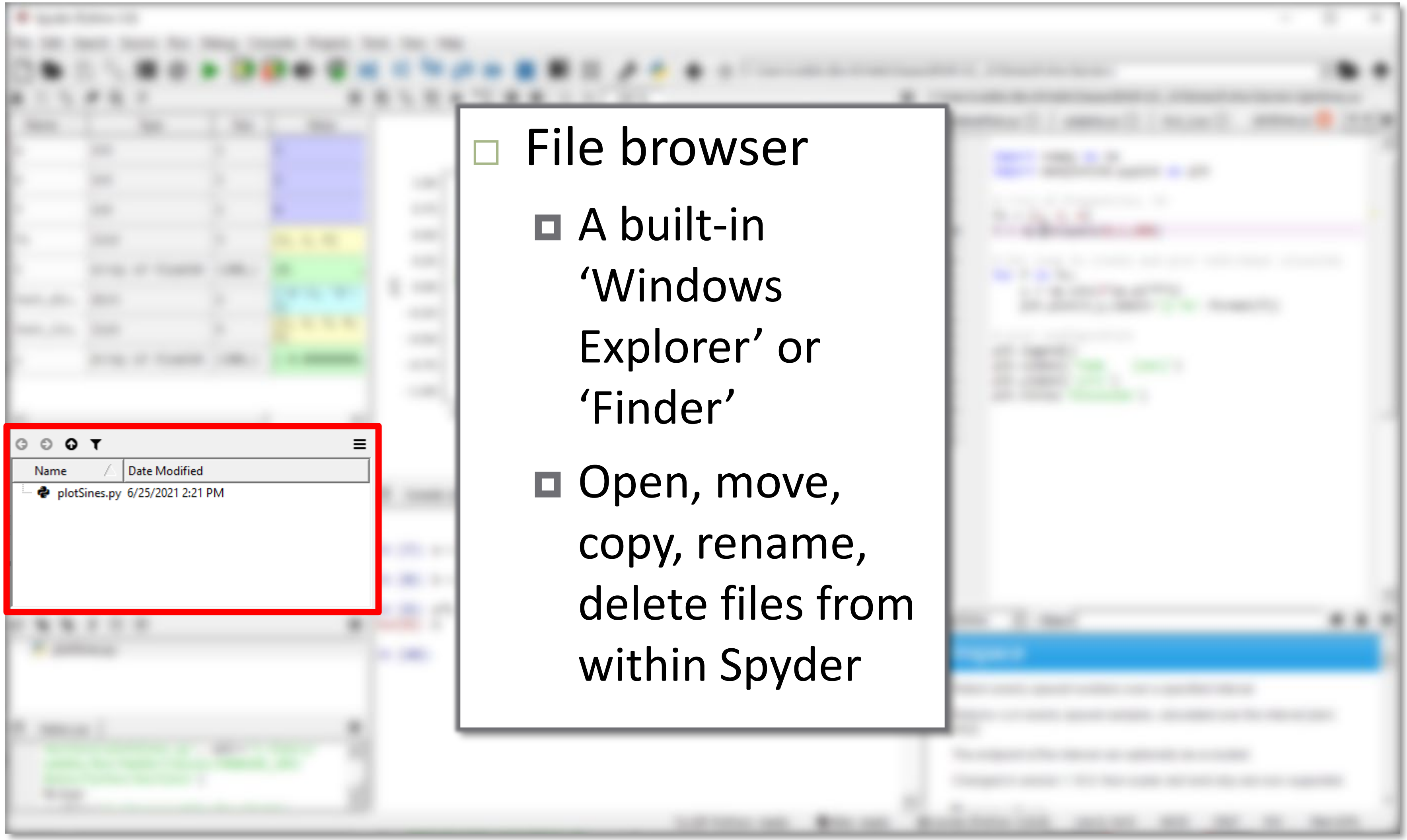
| Name | Type | Size | Value |
|---|---|---|---|
| a | int | 1 | 3 |
| b | int | 1 | 2 |
| f | int | 1 | 4 |
| fs | list | 3 | [1, 2, 4] |
| t | Array of float64 | (200,) | [0. … |
| test_dic… | dict | 2 | {'a':1, 'b':<br>2} |
| test_lis… | list | 5 | [1, 2, 3, 4,<br>5] |
| y | Array of float64 | (200,) | [ 0.00000000… |

y - NumPy object array

|   | 0 |
|---|---|
| 0 | 0 |
| 1 | 0.12596 |
| 2 | 0.249913 |
| 3 | 0.369885 |
| 4 | 0.483966 |
| 5 | 0.590337 |
| 6 | 0.687304 |
| 7 | 0.773323 |

- Lists all variables currently stored in memory
  - Values for scalars and small arrays
  - Size and data types for larger arrays
- Double-click a variable to open in a separate window

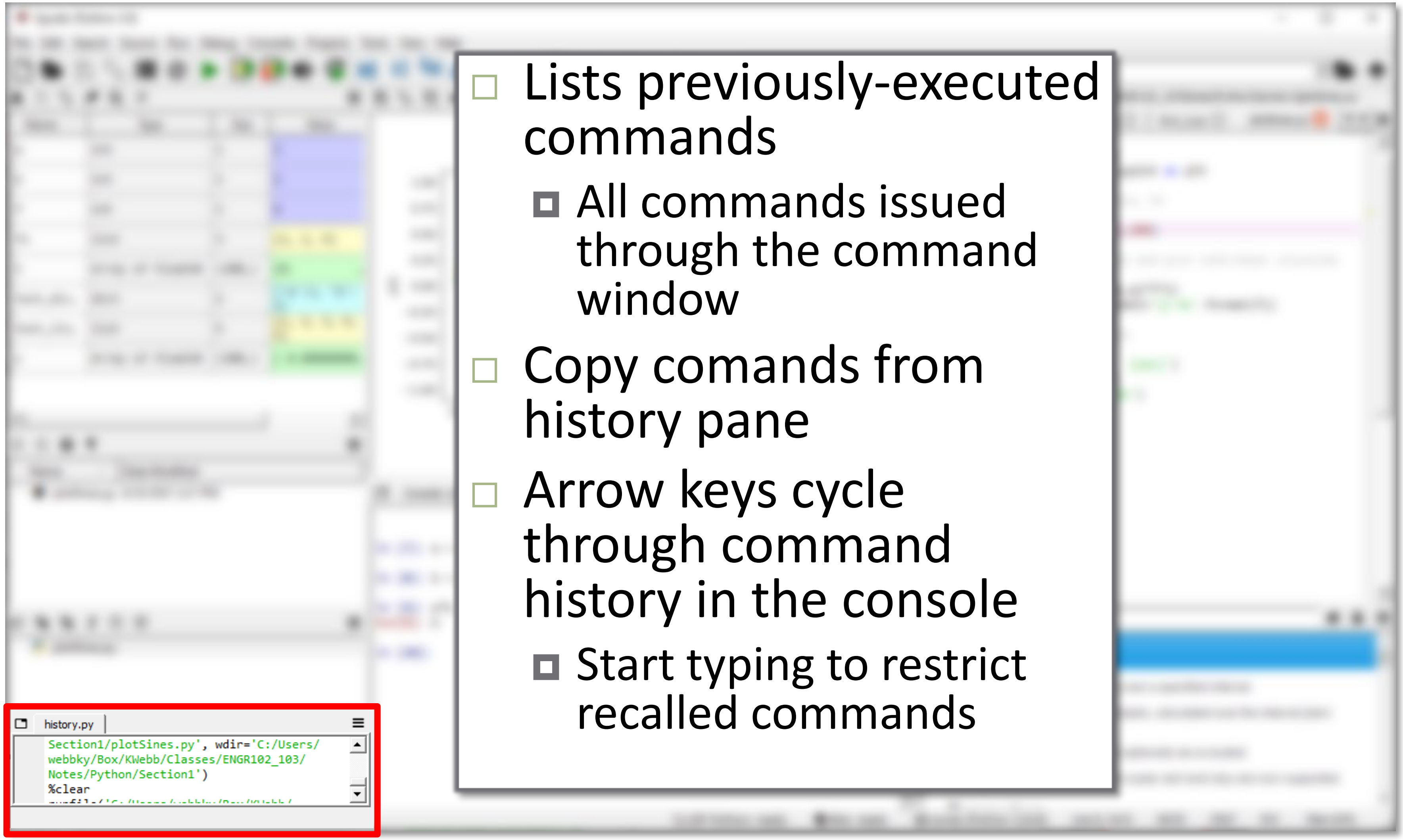
# The Spyder Interface – File Browser



| Name | Date Modified |
|---|---|
| plotSines.py | 6/25/2021 2:21 PM |

- File browser
  - A built-in ‘Windows Explorer’ or ‘Finder’
  - Open, move, copy, rename, delete files from within Spyder

# The Spyder Interface – Command History

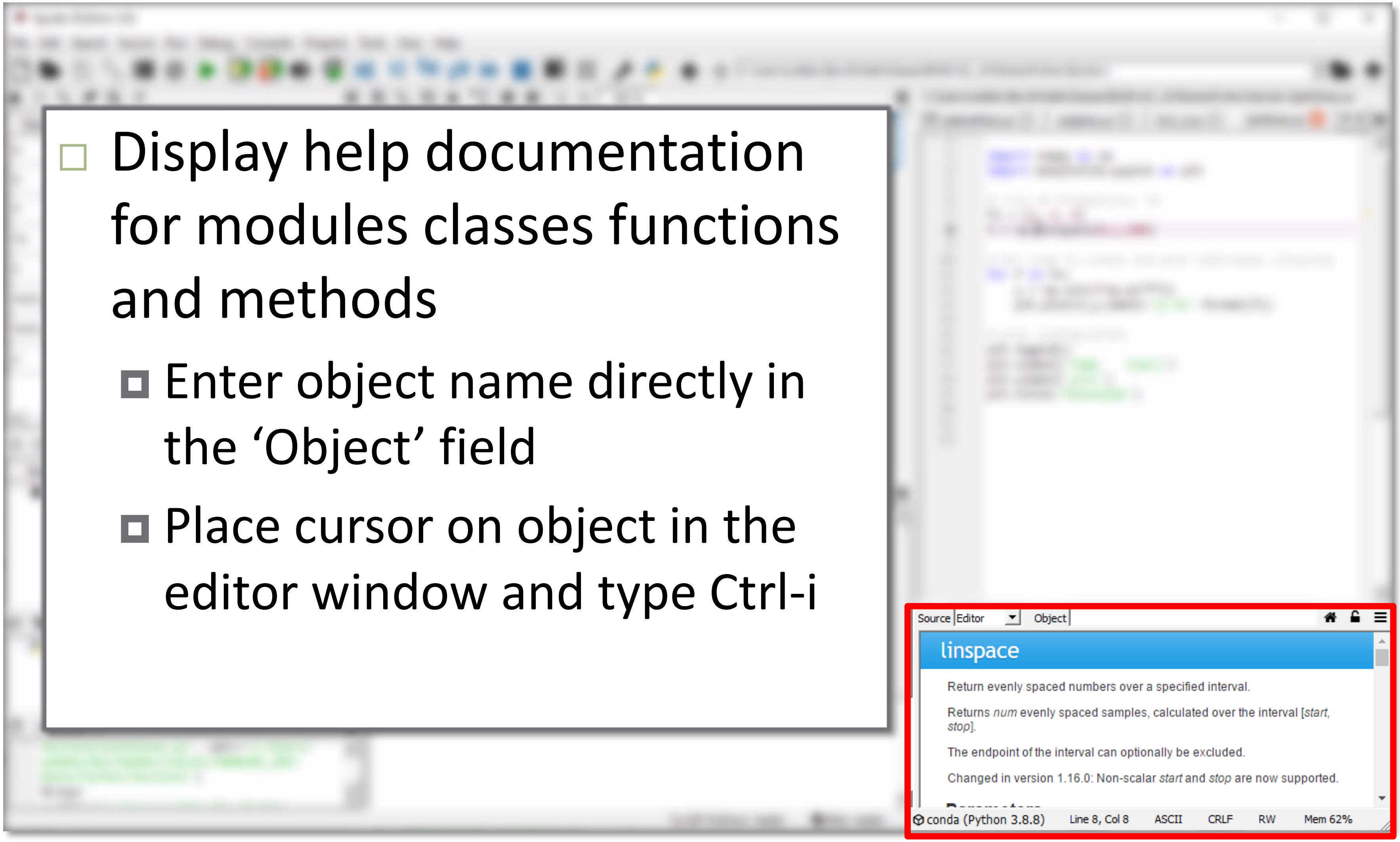


```
Section1/plotSines.py', wdir='C:/Users/
webbky/Box/KWebb/Classes/ENGR102_103/
Notes/Python/Section1')
%clear
```

- Lists previously-executed commands
  - All commands issued through the command window
- Copy comands from history pane
- Arrow keys cycle through command history in the console
  - Start typing to restrict recalled commands

# The Spyder Interface – Help Pane

- Display help documentation for modules classes functions and methods
    - Enter object name directly in the ‘Object’ field
    - Place cursor on object in the editor window and type Ctrl-i



## linspace

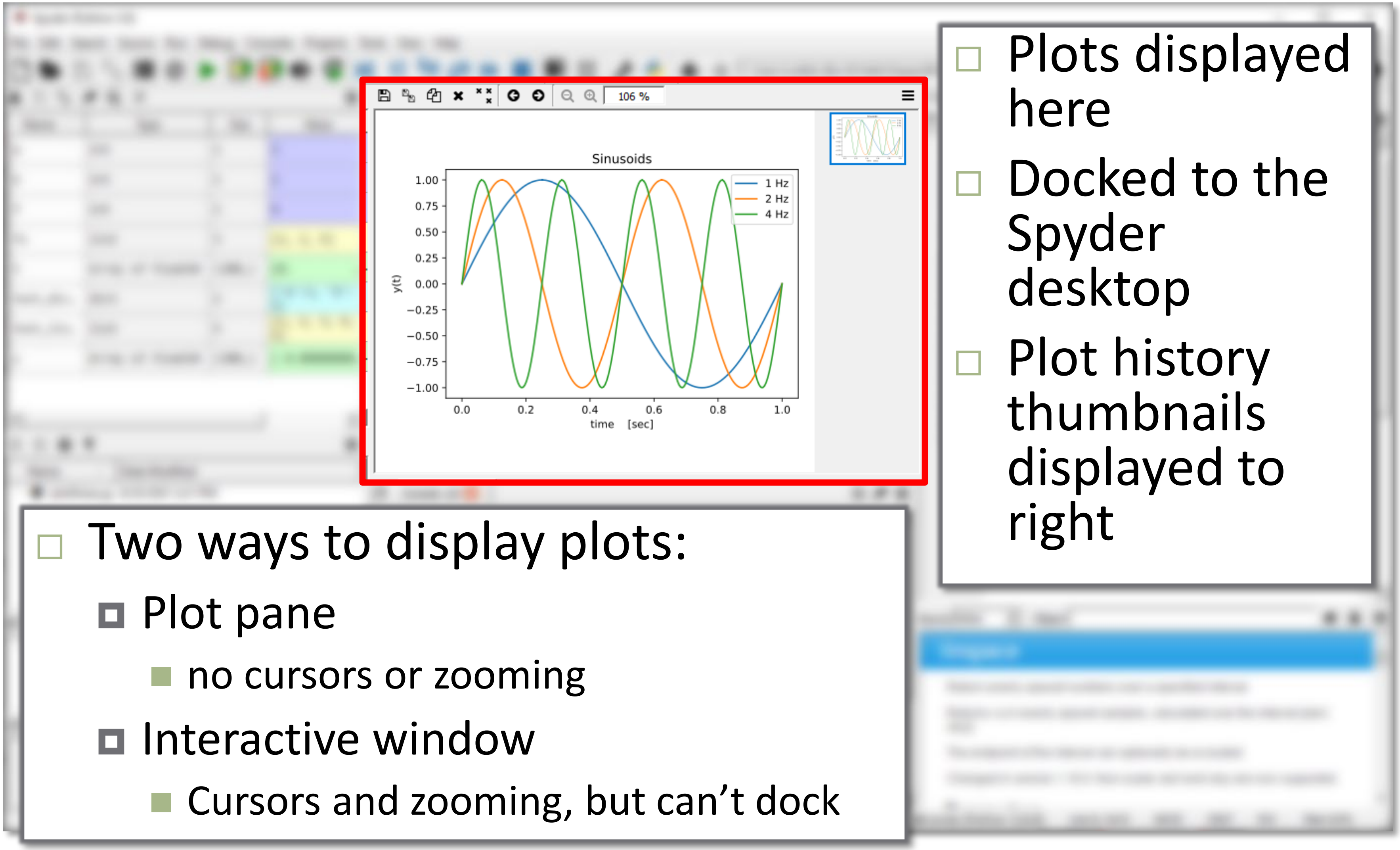
Return evenly spaced numbers over a specified interval.

Returns *num* evenly spaced samples, calculated over the interval [*start*, *stop*].

The endpoint of the interval can optionally be excluded.

Changed in version 1.16.0: Non-scalar *start* and *stop* are now supported.

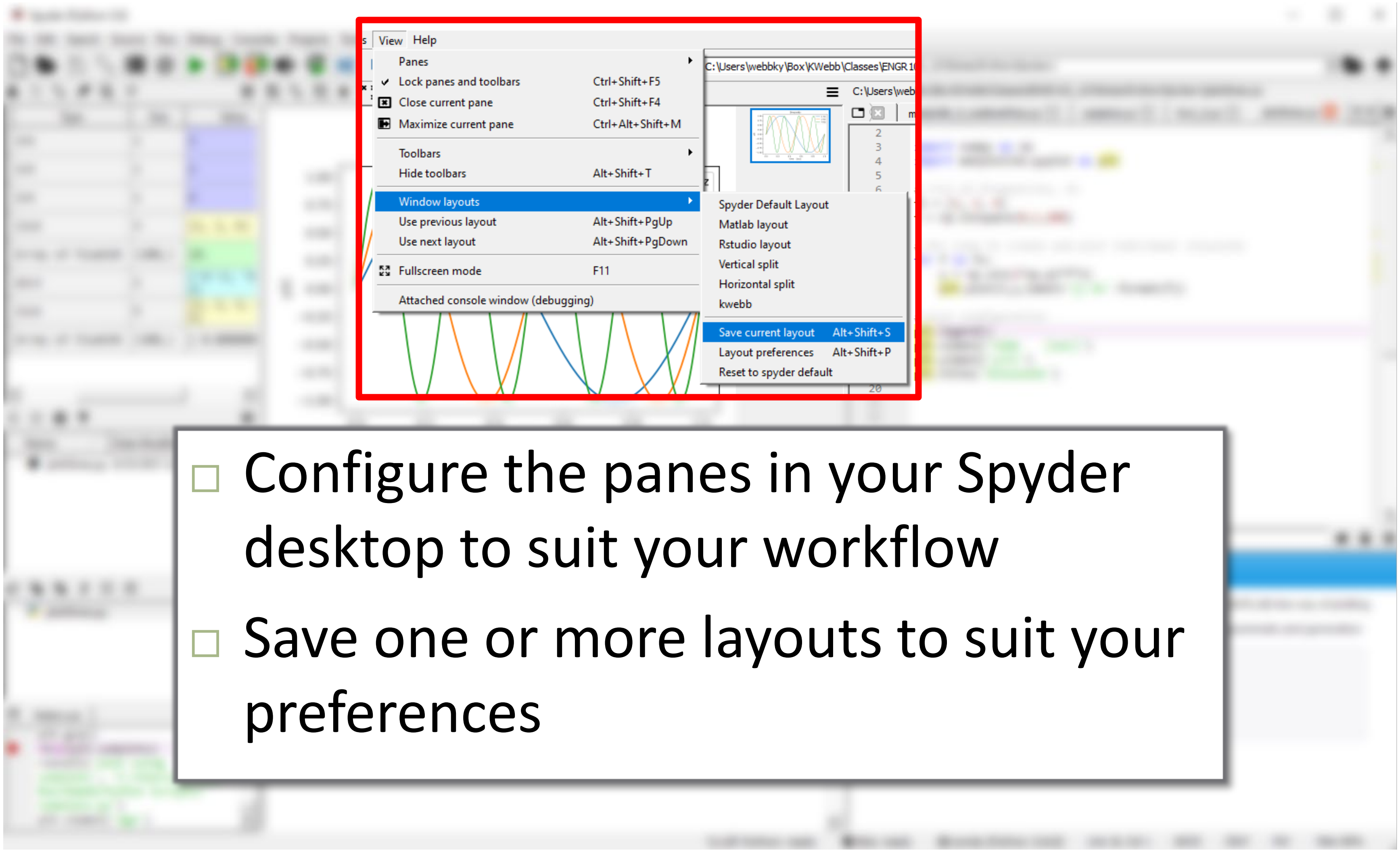
# The Spyder Interface – Plots Pane



- Plots displayed here
- Docked to the Spyder desktop
- Plot history thumbnails displayed to right

- Two ways to display plots:
  - Plot pane
    - no cursors or zooming
  - Interactive window
    - Cursors and zooming, but can’t dock

# The Spyder Interface – Saving Layouts



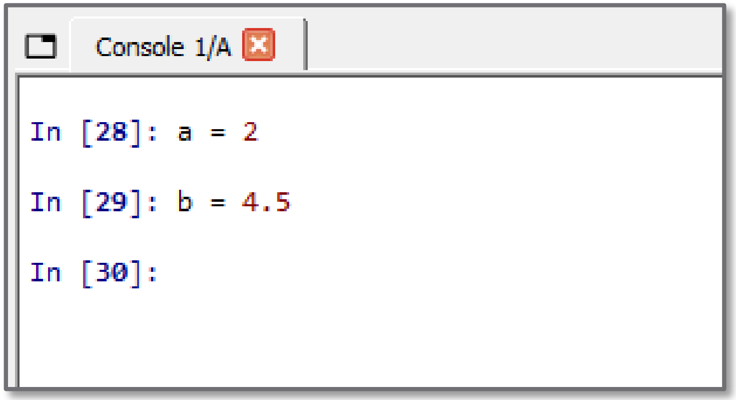
- Configure the panes in your Spyder desktop to suit your workflow
- Save one or more layouts to suit your preferences

# Data Types

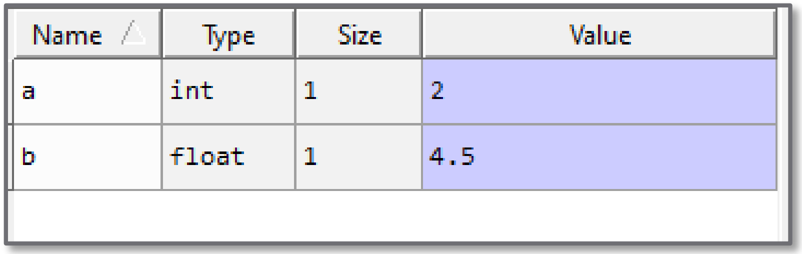
Variables used in Python can be of many different types, e.g. integers, floating-point numbers, alphanumeric characters, etc.

The following section introduces each of these data types. You'll gain a better understanding of each as the course progresses.

# Assignment of Variables



```
In [28]: a = 2

In [29]: b = 4.5

In [30]:
```



| Name | Type | Size | Value |
|---|---|---|---|
| a | int | 1 | 2 |
| b | float | 1 | 4.5 |

- Can define variables and assign values
  - Within a script
  - In the console
- Can then operate on those variables
- Variables appear in variable explorer

# Variable Declaration

- In Python, it isn't necessary to declare a variable before using it, e.g.:

```
a = 7.4039
```

- Declaration occurs automatically upon assignment
- This differs from many other languages, e.g. in C:

```
float a;
a = 7.4039;
```

or

```
float a = 7.4039;
```

# Variable Names

- Variable names must ***start with a letter*** or ***underscore***
- Names may contain ***letters***, ***numbers***, and ***underscore*** characters
  - ***No spaces***
- Some examples:

| Allowed | Not allowed |
|---|---|
| A | Var 3 |
| var1 | 4x_a |
| x_2_a | data file name |
| Avg_price | %pop # |

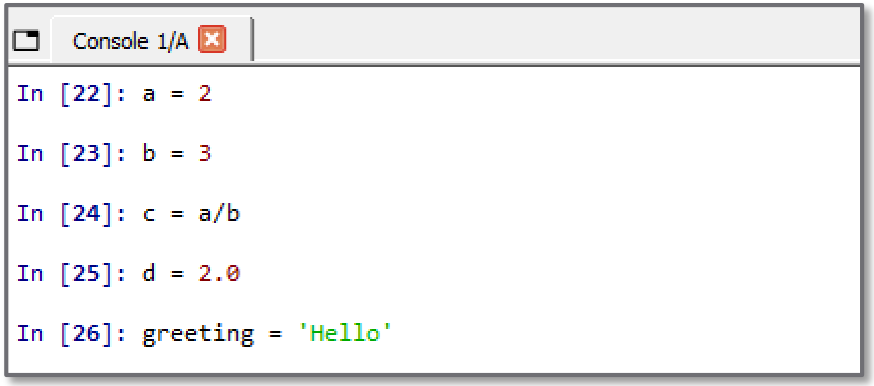
# Variable Names

- Names are ***case sensitive***
  - For example, all three are different:
    - Name_1
    - Name_1
    - NAME_1
- Cannot use Python ***keywords***
  - E.g., for, if, def, True, etc.
- Don’t name variables with names of ***built-in functions***
  - Can be done, but that function will become unavailable

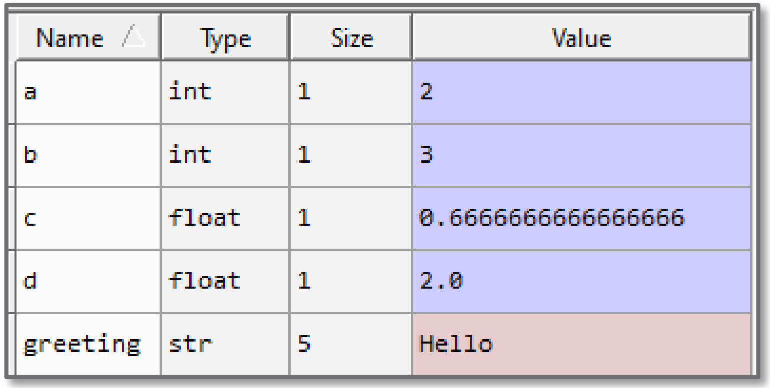
- ***Preferred variable naming convention***:
  - All lowercase
  - Separate multiple words with an underscore

# Variable Declaration – Dynamic Typing

- Python variables are of can be different ***types***, e.g.:
  - Integer: int
  - Floating-point number: float
  - Alpha-numeric string: str
- Python is ***dynamically typed***
  - Don’t need to assign type when defining a variable
  - Python ***interpreter determines type*** at runtime



```
In [22]: a = 2

In [23]: b = 3

In [24]: c = a/b

In [25]: d = 2.0

In [26]: greeting = 'Hello'
```



| Name | Type | Size | Value |
|---|---|---|---|
| a | int | 1 | 2 |
| b | int | 1 | 3 |
| c | float | 1 | 0.6666666666666666 |
| d | float | 1 | 2.0 |
| greeting | str | 5 | Hello |

# Fundamental Python Data Types

- Python supports many different numeric and non-numeric ***data types***, for example

- **Numeric types**
  - `int`
  - `float`
  - `complex`

- **Non-numeric types**
  - `str`
  - `list`
  - `tuple`
  - `set`
  - `dict`
  - `bool`

- We'll introduce each of these types now, but will learn more about them throughout the course

# Mutable vs. Immutable Data Types

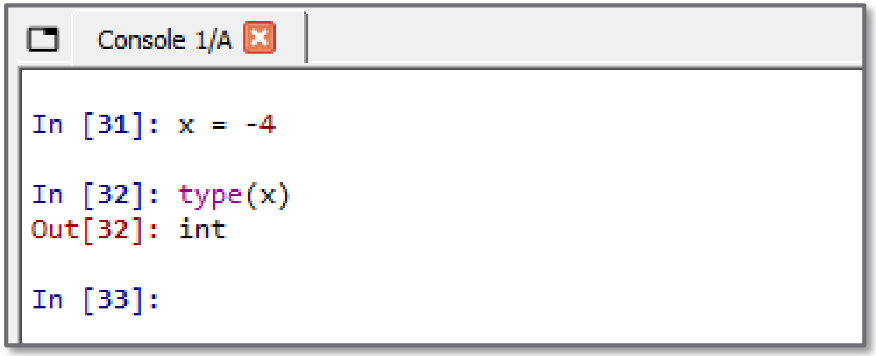
- Data objects of all types are values stored at specific locations in a computer's memory
- All data types fall into one of two categories:
  - ***Immutable***
    - Values cannot be modified after the variable is created in memory
      - Numbers – `int`, `float`, `complex`
      - Strings – `str`
      - Tuples – `tuple`
  - ***Mutable***
    - Values can be modified after variable creation
    - Can add, delete, insert, and rearrange items in a mutable sequence
      - Lists – `list`
      - Dictionaries – `dict`

# Data Types – `int`

- ***Integers***
  - Zero, positive, and negative ***whole numbers***

```
>>> a = 7
>>> x = -4
>>> N = 0
```

- If you assign a whole-number value to a variable, it will automatically be cast as an `int`



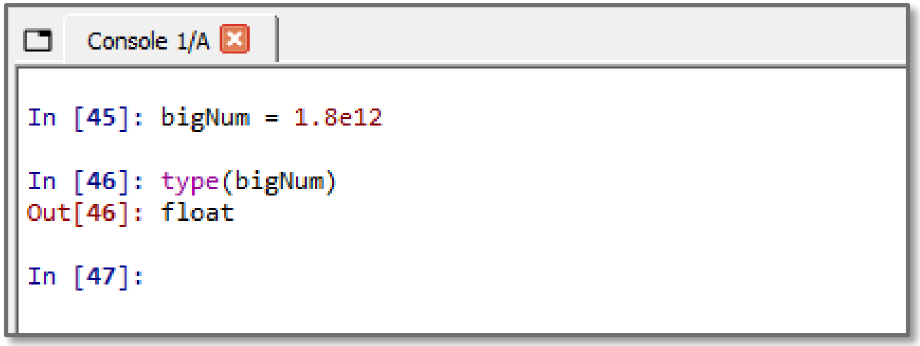
```
In [31]: x = -4

In [32]: type(x)
Out[32]: int

In [33]:
```

# Data Types – `float`

- ***Floating point numbers***
  - Positive, and negative ***non-whole numbers***

```
>>> a = 2.71
>>> x = -4.5
>>> bigNum = 1.8e12
>>> smallNum = 6.4E-9
```

- If you assign a non-whole-number value to a variable, it will automatically be cast as a `float`



```
In [45]: bigNum = 1.8e12

In [46]: type(bigNum)
Out[46]: float

In [47]:
```

# Scientific Notation

- Use ***scientific notation*** to represent very large or very small floating-point numbers, e.g.:

$$1.58 \times 10^{-9}$$

- Very bad practice to type a lot of zeros – ***never do this***:

$$0.00000000158$$

  - Difficult to read, and much too easy to miscount zeros

- In Python use e or E for $\times\ 10^{x}$, e.g.:

```
x = 1.58e-9
```

```
x = 1.58E-9
```

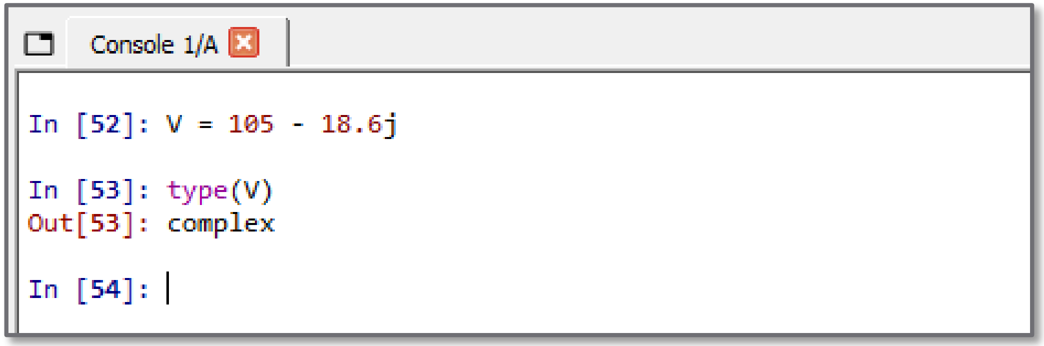
- Don't confuse with the exponential function $e^{x}$ (i.e. $2.718^{x}$)

# Data Types – `complex`

- ***Complex numbers***
  - Numbers with ***real*** and ***imaginary parts***

```
>>> z = 3 + 2j
>>> b = -4.5 + 6j
>>> V = 105 - 18.6j
```

- j is the imaginary unit
  - j = $\sqrt{-1}$

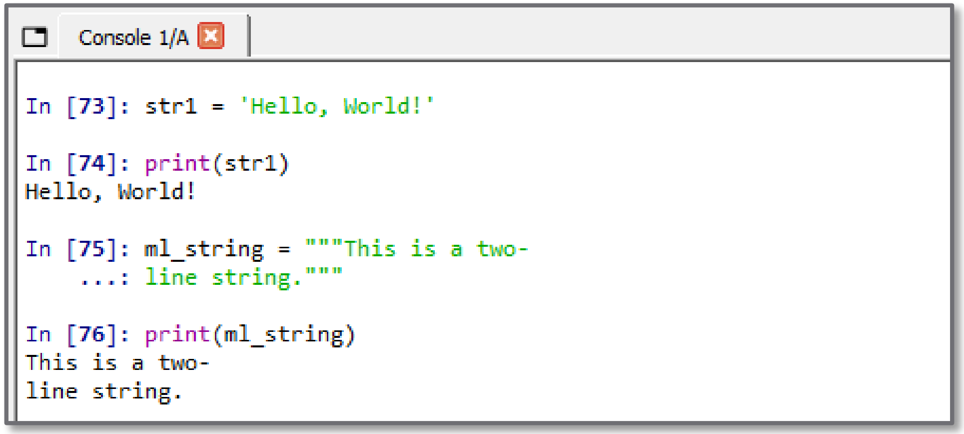


```
In [52]: V = 105 - 18.6j

In [53]: type(V)
Out[53]: complex

In [54]: 
```

# Data Types – str

- ***Strings***
  - Sequences of ***alpha-numeric characters***
  - Enclosed in single, double, or triple quotes

```
>>> str_1 = 'Hello, World!'
>>> Name = "John Doe"
>>> ml_string = '''Multi-line strings
are enclosed in
triple quotes.'''
```

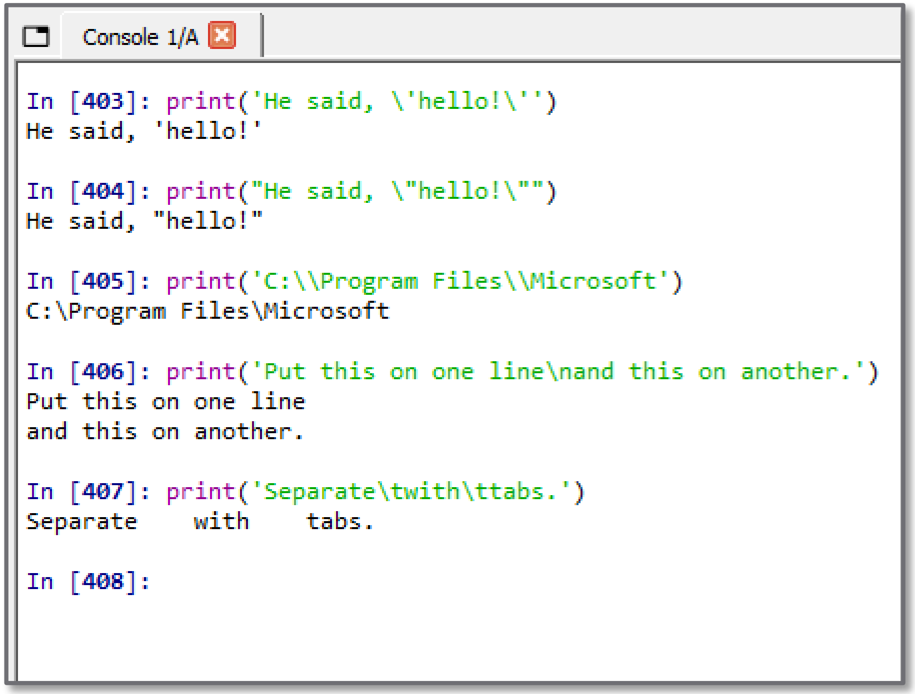


```
In [73]: str1 = 'Hello, World!'

In [74]: print(str1)
Hello, World!

In [75]: ml_string = """This is a two-
    ...: line string."""

In [76]: print(ml_string)
This is a two-
line string.
```

# Data Types – `str` – Escape Characters

- ***Escape characters***
  - Allows you to insert special characters in strings
  - Backslash, \, followed by a special character

| Escape Character | Result |
|---|---|
| \' | Single quote |
| \" | Double quote |
| \\ | Backslash |
| \n | New line |
| \t | Tab |

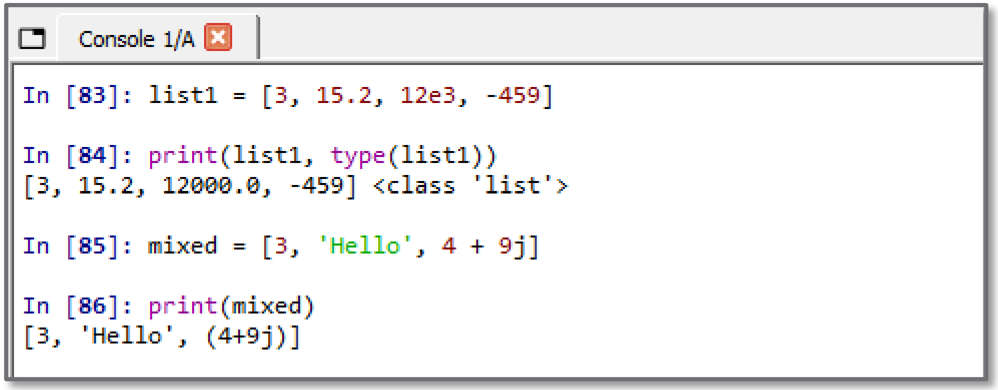


```
In [403]: print('He said, \'hello!\'')
He said, 'hello!'

In [404]: print("He said, \"hello!\"")
He said, "hello!"

In [405]: print('C:\\Program Files\\Microsoft')
C:\Program Files\Microsoft

In [406]: print('Put this on one line\nand this on another.')
Put this on one line
and this on another.

In [407]: print('Separate\twith\ttabs.')
Separate    with    tabs.

In [408]:
```

# Data Types – `list`

- ***Lists***
  - Ordered, mutable collections of one or more different data types
  - Enclosed in square brackets, [ ], separated by commas

```
>>> list1 = [3, 15.2, 12e3, -459]
>>> names = ['Jane', ‘Bob', 'Sally']
>>> mixed = [3, 'Hello', 4 + 9j]
```

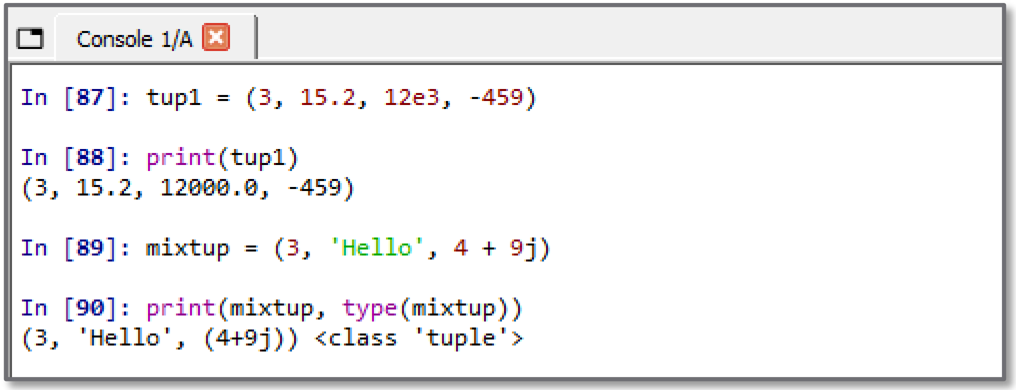


```
In [83]: list1 = [3, 15.2, 12e3, -459]

In [84]: print(list1, type(list1))
[3, 15.2, 12000.0, -459] <class 'list'>

In [85]: mixed = [3, 'Hello', 4 + 9j]

In [86]: print(mixed)
[3, 'Hello', (4+9j)]
```

# Data Types – `tuple`

- ***Tuples***
  - Ordered, immutable collections of one or more different data types
  - Like a list, but immutable
  - Enclosed in square parentheses, ( ), separated by commas

```
>>> tup1 = (3, 15.2, 12e3, -459)
>>> names = ('Jane', 'Bob', 'Sally')
>>> mixtup = (3, 'Hello', 4 + 9j)
```

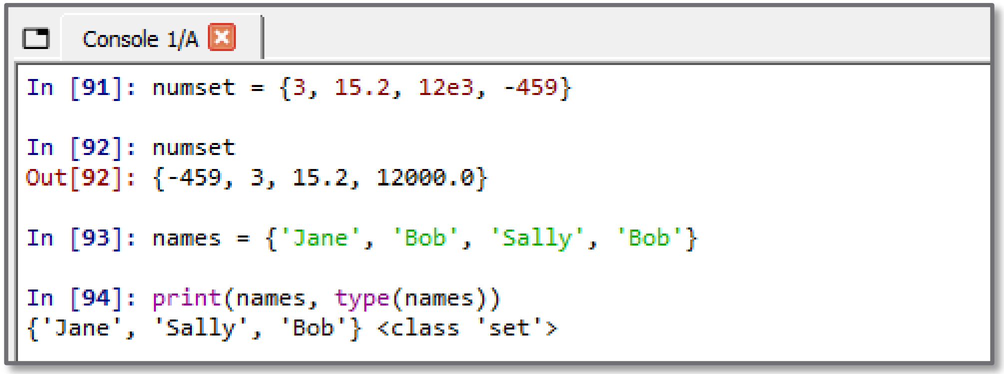


```
In [87]: tup1 = (3, 15.2, 12e3, -459)

In [88]: print(tup1)
(3, 15.2, 12000.0, -459)

In [89]: mixtup = (3, 'Hello', 4 + 9j)

In [90]: print(mixtup, type(mixtup))
(3, 'Hello', (4+9j)) <class 'tuple'>
```

# Data Types – set

- ***Sets***
  - Unordered, mutable collections of one or more different data types
  - Enclosed in square curly brackets, { }, separated by commas
  - Sets do not store duplicate objects
  - Suitable for mathematical set operations, e.g., union, intersection, difference, etc.

```
>>> numset = {3, 15.2, 12e3, -459}
>>> names = {'Jane', 'Bob', 'Sally'}
>>> set3 = {3, 'Hello', 4 + 9j}
```



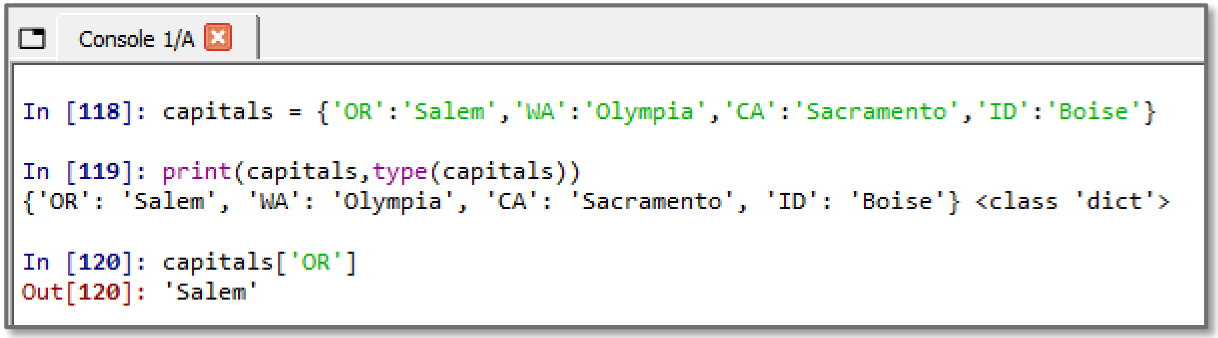
```
In [91]: numset = {3, 15.2, 12e3, -459}

In [92]: numset
Out[92]: {-459, 3, 15.2, 12000.0}

In [93]: names = {'Jane', 'Bob', 'Sally', 'Bob'}

In [94]: print(names, type(names))
{'Jane', 'Sally', 'Bob'} <class 'set'>
```

# Data Types – `dict`

- ***Dictionaries***
  - Ordered, mutable collections of data stored as ***key:value*** pairs
  - Enclosed in square curly brackets, { }
  - Keys and values separated by colons
  - Key:value pairs separated by commas
  - Duplicate keys are not allowed

```
>>> person1 = {'Name':, 'Joe', 'Age':, 32, 'Hair':,
'brown', 'Eyes':, 'green'}
```

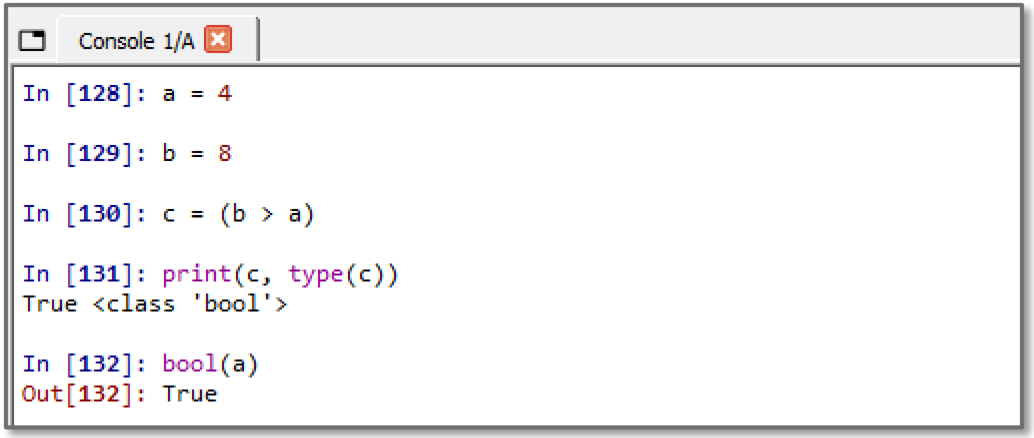
```
>>> capitals = {'OR':, 'Salem', 'WA':, 'Olympia',
'CA':, 'Sacremento', 'ID':, 'Boise}
```



```
In [118]: capitals = {'OR':'Salem','WA':'Olympia','CA':'Sacramento','ID':'Boise'}

In [119]: print(capitals,type(capitals))
{'OR': 'Salem', 'WA': 'Olympia', 'CA': 'Sacramento', 'ID': 'Boise'} <class 'dict'>

In [120]: capitals['OR']
Out[120]: 'Salem'
```

# Data Types – `bool`

- ***Booleans***
  - One of two *logical* values: **`True`** or **`False`**
  - Often the result of a *logical expression*, e.g., `a > b`
  - Any value can be cast as a Boolean using the `bool()` function
    - True:
      - Non-zero numbers
      - Non-empty strings, lists, tuples, sets, or dictionaries
    - False:
      - Zero
      - Empty strings, lists, tuples, sets, or dictionaries



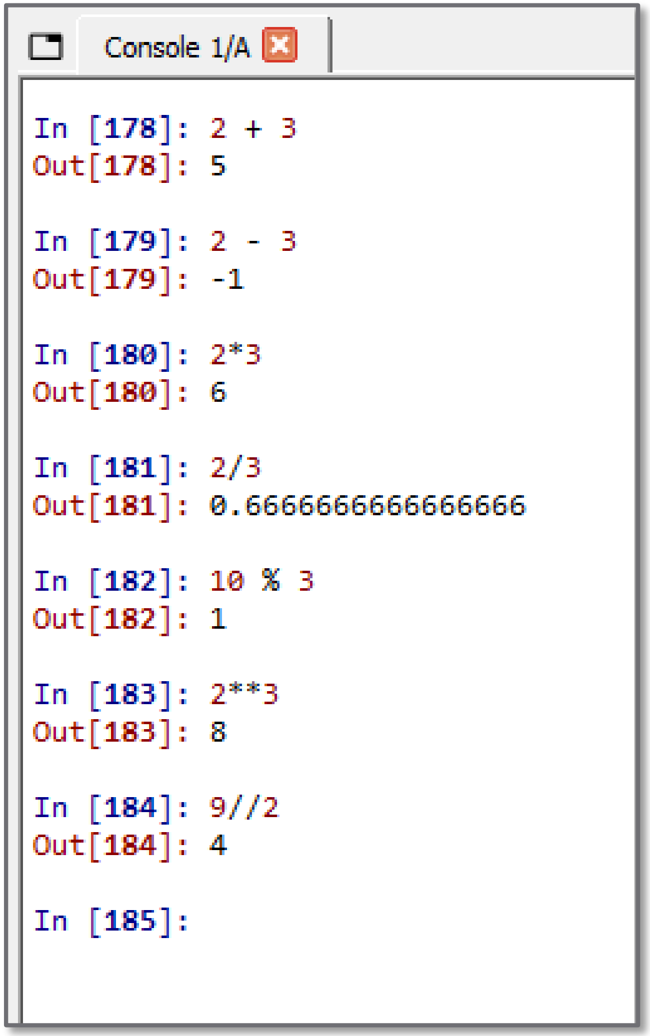
```
In [128]: a = 4

In [129]: b = 8

In [130]: c = (b > a)

In [131]: print(c, type(c))
True <class 'bool'>

In [132]: bool(a)
Out[132]: True
```

# Mathematical Operations

Python includes the most basic mathematical operations. Other math functions will be accessed by importing the NumPy package

# Basic Mathematical Operations

- Python itself includes only seven mathematical operators
  - Addition:  +
  - Subtraction:  –
  - Multiplication:  *
  - Division:  /
  - Modulus:  %
  - Exponentiation:  **
  - Floor division:  //

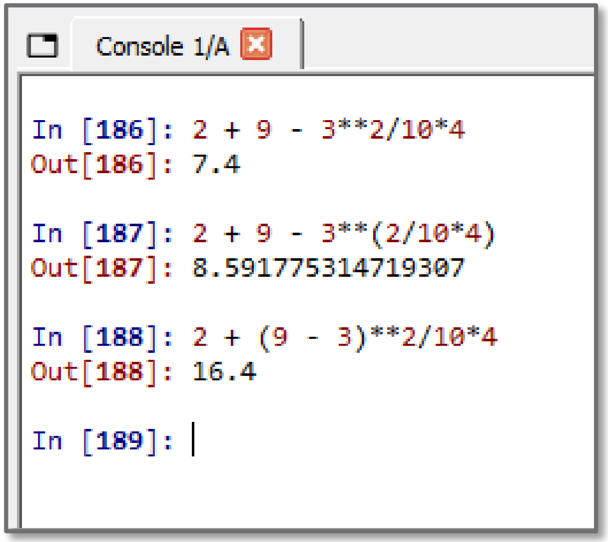


```
In [178]: 2 + 3
Out[178]: 5

In [179]: 2 - 3
Out[179]: -1

In [180]: 2*3
Out[180]: 6

In [181]: 2/3
Out[181]: 0.6666666666666666

In [182]: 10 % 3
Out[182]: 1

In [183]: 2**3
Out[183]: 8

In [184]: 9//2
Out[184]: 4

In [185]:
```

# Order of Operations

- Python order of operations:

| | | |
|---|---|---|
| 1) | ( ) | parentheses |
| 2) | ^ | exponentiation |
| 3) | - | negation |
| 4) | *, / | multiplication, division |
| 5) | +, - | addition, subtraction |



```
In [186]: 2 + 9 - 3**2/10*4
Out[186]: 7.4

In [187]: 2 + 9 - 3**(2/10*4)
Out[187]: 8.591775314719307

In [188]: 2 + (9 - 3)**2/10*4
Out[188]: 16.4

In [189]: 
```

- Expressions are evaluated left to right within each level of the precedence hierarchy

# Other Built-In Python Functions

- A few other math-related built-in Python functions:
  - abs(x): absolute value

```
>>> a = abs(-1.76)
1.76
>>> z = abs(2 – 2j)
2.828
```

  - len(x): returns the length of an object

```
>>> len([2, 4, 5, 3, 1])
5
>>> len('Hello, World!')
13
```

# Other Built-In Python Functions

- A few other math-related built-in Python functions:
  - max(x): maximum value in a sequence

```
>>> x_max = max([2, 4, 5, 3, 1])
5
```

  - min(x): minimum value in a sequence

```
>>> x_min = min([2, 4, 5, 3, 1])
1
```

  - type(x): returns the type of an object

```
>>> type([2, 4, 5, 3, 1])
list
>>> type('Hello, World!')
str
```

# NumPy

Here we will introduce the concept of ***packages***, and will look specifically at the package we will use most for mathematical operations, ***NumPy***.

# Packages

- ***Python packages***
  - Libraries consisting of multiple ***modules***, or individual Python files
  - Modules within a package define
    - Data types
    - Functions
  - Must install a package before we can use it
    - Anaconda distribution includes all the packages we will need
  - Must import a package in our code before we can use it
    - Use the `import` function

- Packages available for
  - Array processing and mathematics
  - Plotting
  - Data analysis
  - GUI development
  - Much, much more …

# NumPy

- We will use the NumPy (**Num**erical **Py**thon) package extensively
- Fundamental data type:
  - Multi-dimensional array object – `ndarray`
    - Useful for engineering computation
- Many built-in functions
  - Mathematical operations, e.g.:
    - Trigonometric functions
    - Exponents and logarithms
    - Complex number operations
  - Array creation an manipulation routines
  - Polynomial creation, manipulation, fitting, etc.
  - Much more …

# Using NumPy

- To use NumPy functions and data types, we must first ***import*** it:

```
>>> import numpy as np
```

  - We can assign it a shortened name, np, to keep our code clean

- To call functions defined in NumPy, ***precede the function name with* `np.`**

```
>>> N = np.log2(1024)

>>> x = 3*np.sin(np.pi/2)
```

- We'll now introduce a small sample of NumPy functions

# NumPy – Trigonometric Functions

- `sin(x)`, `cos(x)`, `tan(x)`
  - Input in radians

```
>>> y = np.sin(x)
>>> y = np.sin(np.radians(x))
```

- `arcsin(x)`, `arccos(x)`, `arctan(x)`
  - Inverse trig functions
  - Output in radians

```
>>> theta = np.arcsin(0.6)
```

# NumPy – Trigonometric Functions

- `arctan2(x)` – quadrant-aware inverse tangent
  - Accounts for the difference between, e.g. , 45° and 225°
  - Output in radians

```
>>> phi = np.arctan2(-4, 3)
>>> phi_deg = np.degrees(np.arctan2(-4, 3))
```

- `degrees(x)` – converts from radians to degrees

```
>>> ang45 = np.degrees(np.pi/4)
```

- `radians(x)` – converts from degrees to radians

```
>>> angPi = np.radians(180)
```

# NumPy – Rounding

- `around(x, decimals=0)` – round to the specified number of decimals (default, 0)

```
>>> xint = np.around(1.6)
2.0

>>> xrnd = np.around(np.pi, decimals=2)
3.14
```

  - Numbers exactly halfway between rounded decimal values round to the nearest ***even value***

```
>>> x0 = np.around(2.5)
2.0

>>> x1 = np.around(1.65, decimals=1)
1.6

>>> y1 = np.around(1.55, decimals=1)
1.6
```

# NumPy – Rounding

- `fix(x)` – round to the nearest integer ***toward zero***

```
>>> xfix = np.fix(1.2)
1.0

>>> yfix = np.fix(-2.8)
-2.0
```

- `floor(x)` – round to the nearest integer ***toward negative infinity***

```
>>> xfloor = np.floor(1.6)
1.0

>>> xflr = np.floor(-1.2)
-2.0
```

- `ceil(x)` – round to the nearest integer ***toward positive infinity***

```
>>> xceil = np.fix(1.2)
2.0

>>> yceil = np.fix(-2.8)
-2.0
```

# NumPy – Exponents

- `exp(x)` – exponential: $e^x$

```
>>> y = np.exp(4.1)
60.3403

>>> e = np.exp(1)
2.71828
```

- `exp2(x)` – power of 2: $2^x$

```
>>> x = np.exp2(3)
8.0

>>> N = np.exp2(10)
1024.0
```

# NumPy – Logarithms

- `log(x)` – natural log

```
>>> y = np.log(5)
1.609
```

- `log10(x)` – base-10 logarithm

```
>>> x = np.log10(1e4)
4.0
```

- `log2(x)` – base-2 logarithm

```
>>> x = np.log2(256)
8.0
```

# NumPy – Complex Numbers

- `real(z)` – real part of a complex number

```
>>> y = np.log(5)
1.609
```

- `imag(z)` – imaginary part of a complex number

```
>>> x = np.log10(1e4)
4.0
```

- `angle(z)` – angle of complex number in radians

```
>>> x = np.log2(256)
8.0
```

- `conj(z)` – complex conjugate

```
>>> x = np.log2(256)
8.0
```

# NumPy – Miscellaneous

- `sqrt(x)` – square root

```
>>> y = np.sqrt(2)
1.4142
```

- `sum(x)` – sum of all elements in a sequence

```
>>> total = np.sum([2, 4, 5, 3, 1])
15
```

- `sign(x)` – returns: -1 if x < 0, 0 if x == 0, 1 if x > 0

```
>>> np.sign([-12, 4, 6, 0, -3])
array([-1, 1, 1, 0, -1])
```

# NumPy – Element-Wise Operations

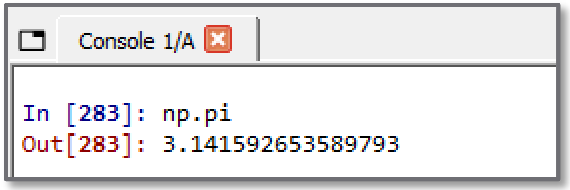
- Numpy functions operate element-by-element on array (or other sequence) inputs
  - Return ***array*** outputs (more later)

```
>>> np.log10([1e4, 0.001, 10, 1e-6])
array([ 4., -3.,  1., -6.])


>>> np.sqrt([4, 9, 25, 1e4])
array([  2.,   3.,   5., 100.])
```

  - Eliminates the need to explicitly perform the operation on each element in an array

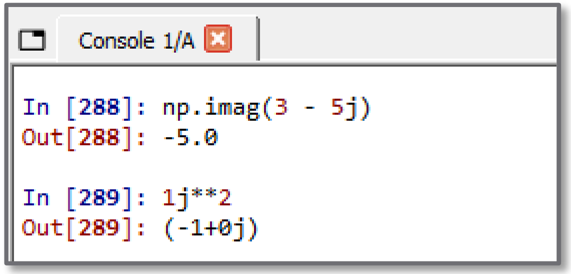
# Built-In Constants

- Some built-in Python and Numpy constants:
  - $\pi$: `np.pi`



```
In [283]: np.pi
Out[283]: 3.141592653589793
```

  - Imaginary unit ($\sqrt{-1}$): `i` or `j`



```
In [288]: np.imag(3 - 5j)
Out[288]: -5.0

In [289]: 1j**2
Out[289]: (-1+0j)
```

  - Infinity ($\infty$): `inf`
  - Not-a-number: NaN or nan
    - Both `inf` and nan often result from algorithmic errors

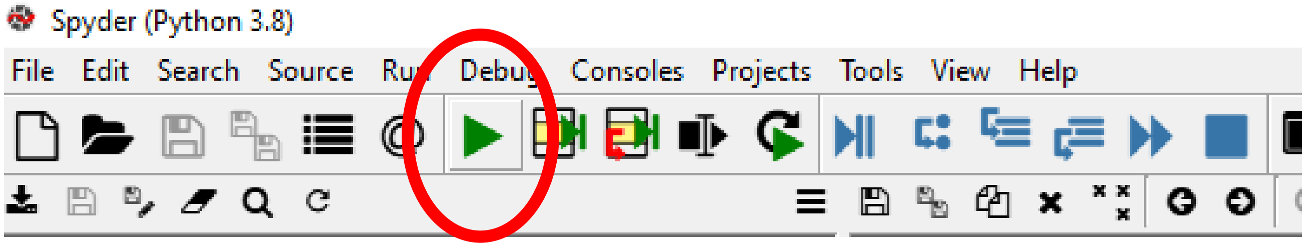
# Python Scripts - Modules

# Spyder Console

- 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

- ***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

- 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

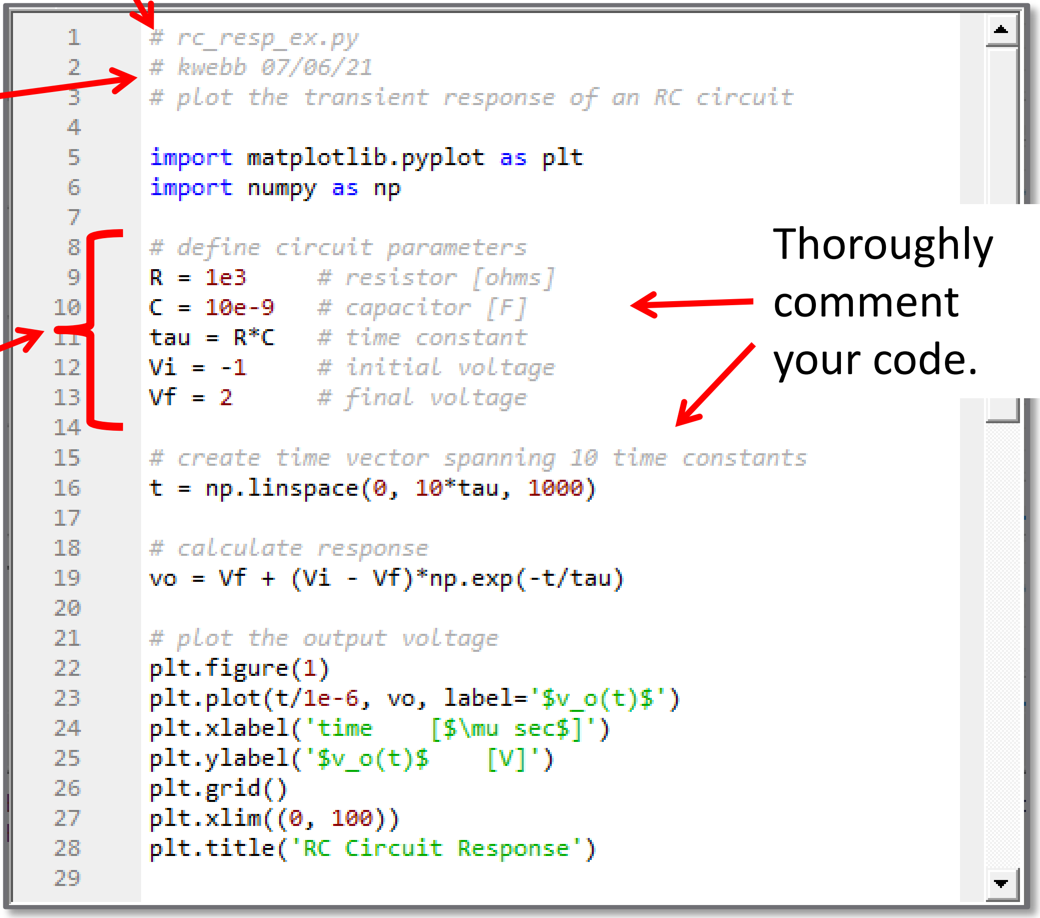
Start scripts with a comment listing the file name.

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**: **D**on't **R**epeat **Y**ourself

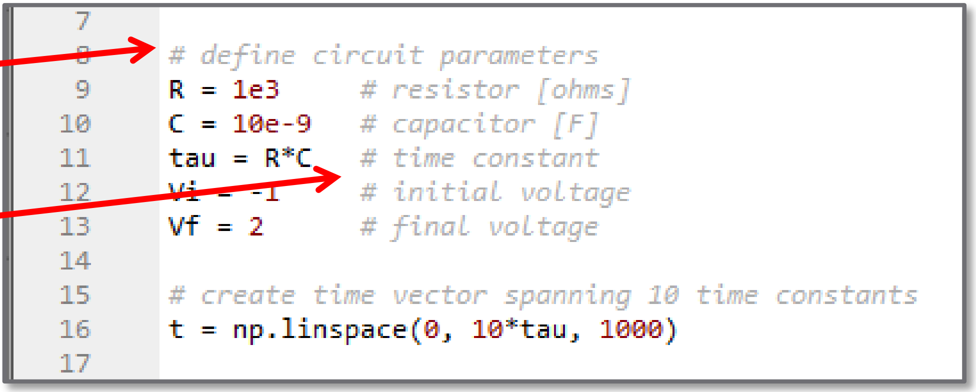
Thoroughly comment your code.



```python
# rc_resp_ex.py
# kwebb 07/06/21
# plot the transient response of an RC circuit

import matplotlib.pyplot as plt
import numpy as np

# define circuit parameters
R = 1e3      # resistor [ohms]
C = 10e-9    # capacitor [F]
tau = R*C    # time constant
Vi = -1      # initial voltage
Vf = 2       # final voltage

# create time vector spanning 10 time constants
t = np.linspace(0, 10*tau, 1000)

# calculate response
vo = Vf + (Vi - Vf)*np.exp(-t/tau)

# plot the output voltage
plt.figure(1)
plt.plot(t/1e-6, vo, label='$v_o(t)$')
plt.xlabel('time    [$\mu sec$]')
plt.ylabel('$v_o(t)$    [V]')
plt.grid()
plt.xlim((0, 100))
plt.title('RC Circuit Response')

```

# Comments

- 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



```
# define circuit parameters
R = 1e3      # resistor [ohms]
C = 10e-9    # capacitor [F]
tau = R*C    # time constant
Vi = -1      # initial voltage
Vf = 2       # final voltage

# create time vector spanning 10 time constants
t = np.linspace(0, 10*tau, 1000)
```

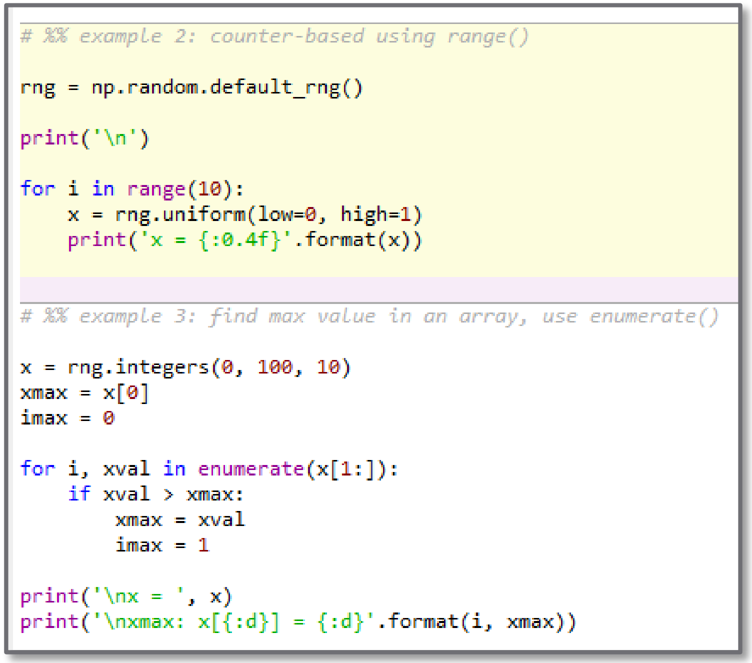
- Ctrl+1 comments and uncomments a line of text in the Spyder editor
- Commenting is useful for temporarily removing instructions from a script

# Cells

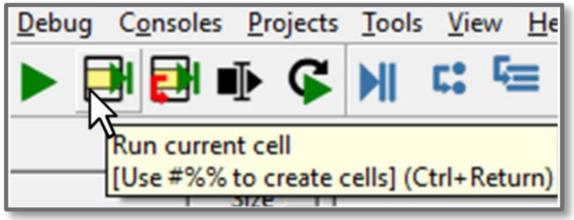
- Can divide Spyder scripts into ***cells***
  - Code blocks that can be executed at once, without running the entire script
- Cells are defined with a special comment line:
  - Follow the hash mark, #, with two percent signs, %%
  - Can also include comment text

```
# %% 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 = {:0.4f}'.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))
```



Run current cell
[Use #%% to create cells] (Ctrl+Return)

# Pseudocode

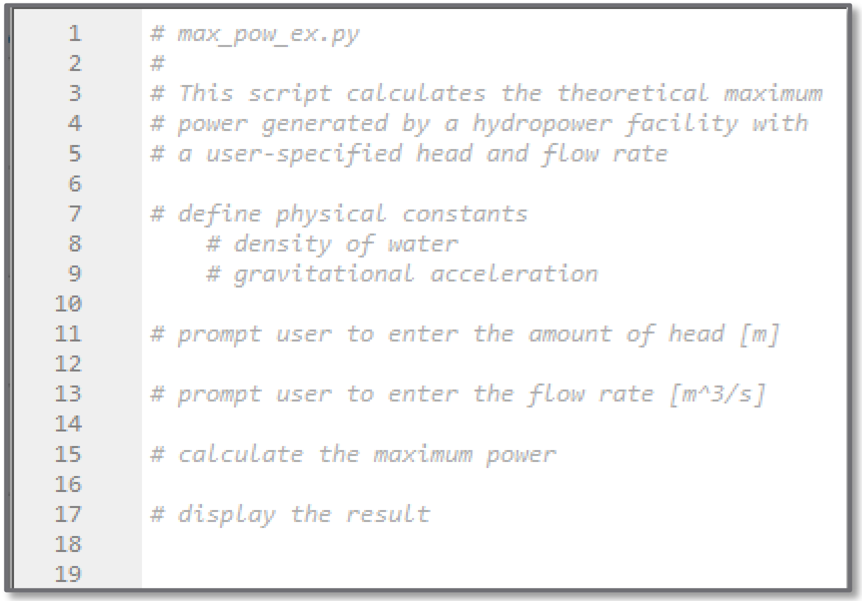
- The most important part of the process of writing computer code is ***planning***
  - Determine exactly what the program should do
  - And, how it will do it
- Before writing any code, write a ***step-by-step description*** of the program
  - ***Natural language***
  - ***Graphical*** – flow chart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

# Programming Process

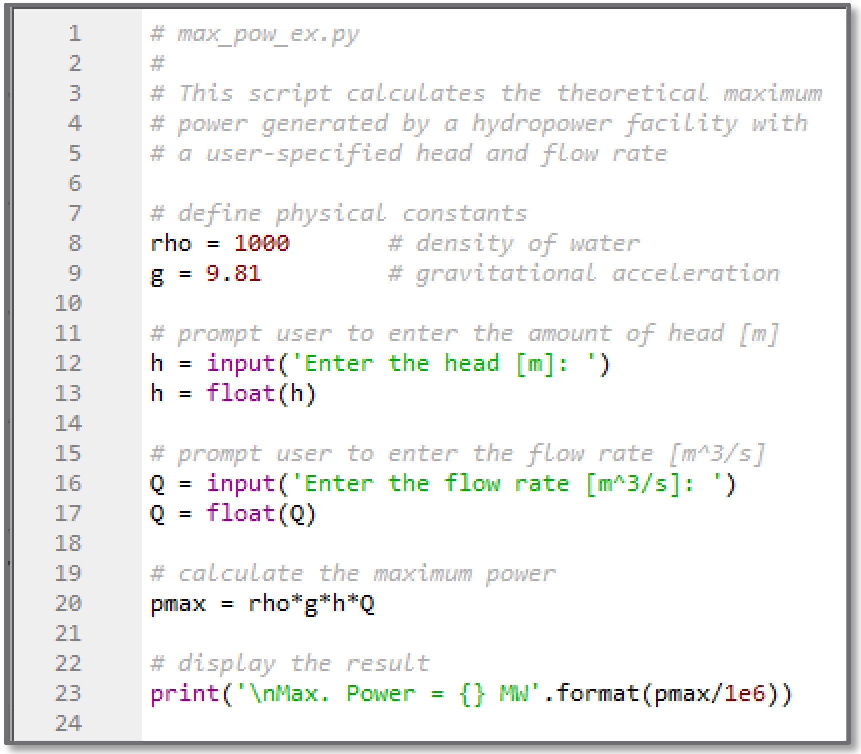
- **Programming process**:
  - ***Define the problem***
    - Ensure you have a complete understanding of the problem
    - Determine exactly what the program should do
      - Inputs and outputs
      - Relevant equations
  - ***Design the program***
    - ***Pseudocode*** – language-independent
  - ***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

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



```python
# max_pow_ex.py
#
# This script calculates the theoretical maximum
# power generated by a hydropower facility with
# a user-specified head and flow rate

# define physical constants
    # density of water
    # gravitational acceleration

# prompt user to enter the amount of head [m]

# prompt user to enter the flow rate [m^3/s]

# calculate the maximum power

# display the result
```



```python
# max_pow_ex.py
#
# This script calculates the theoretical maximum
# power generated by a hydropower facility with
# a user-specified head and flow rate

# define physical constants
rho = 1000       # density of water
g = 9.81         # gravitational acceleration

# prompt user to enter the amount of head [m]
h = input('Enter the head [m]: ')
h = float(h)

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

# calculate the maximum power
pmax = rho*g*h*Q

# display the result
print('\nMax. Power = {} MW'.format(pmax/1e6))
```

# Sequential Code Execution

- 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`

# Inputs & Outputs

# Inputs to Scripts

- Inputs to a script:
  - Assignments of variable values

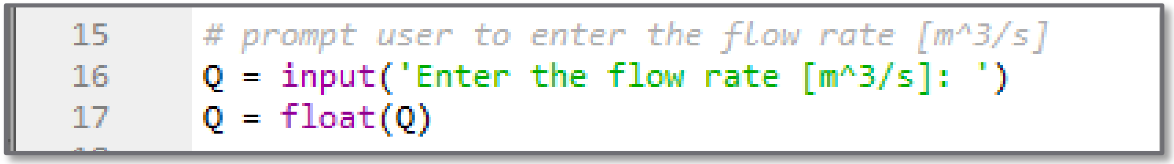
- Several input methods:
  - Within the script
  - From external files (.csv, Excel, etc.) – more later
  - Specified by user during execution – `input()`

# User-Specified Input – `input()`

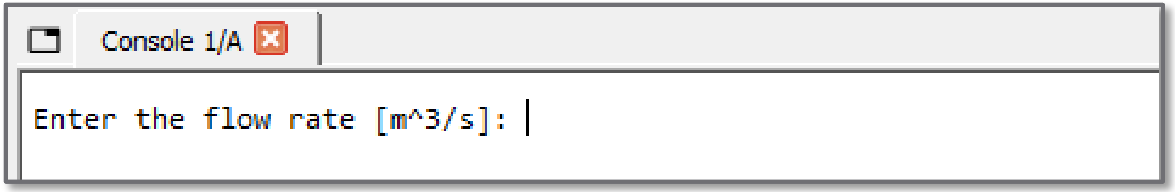
- Prompt user for value to be assigned to a variable

```
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:



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



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

# Outputs from Scripts

- 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)

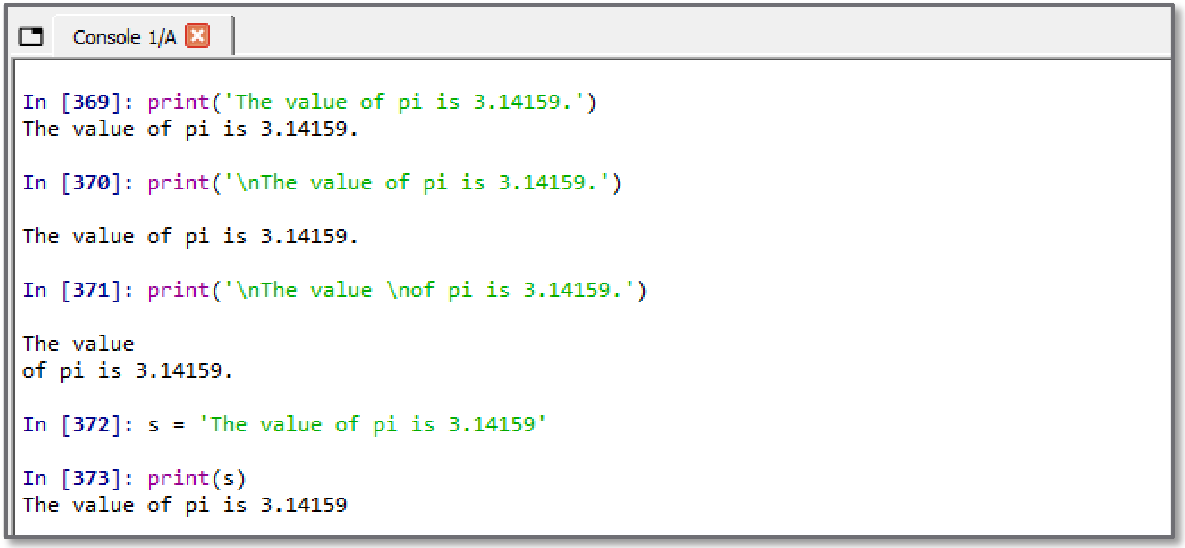
# print()

- Output a string to the console

`print(string)`

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

- For example:



```
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
```

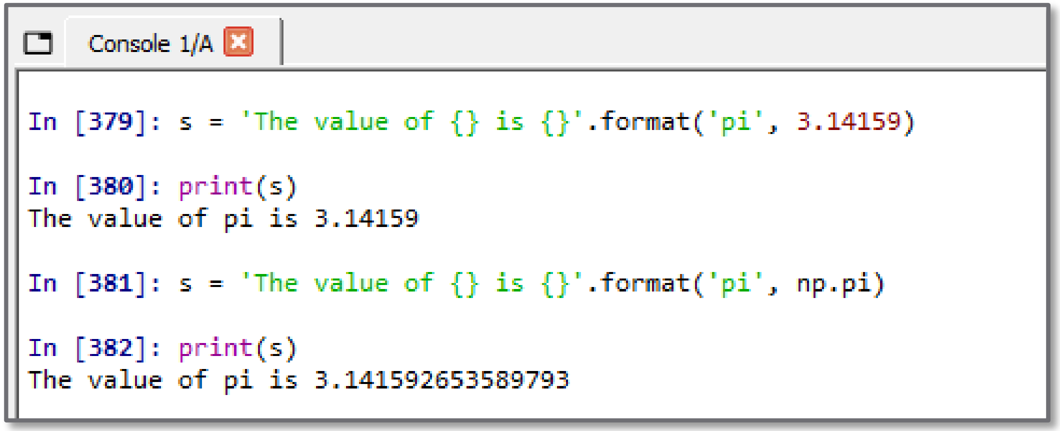
# Formatting Strings – .format()

- 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:



```
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
```

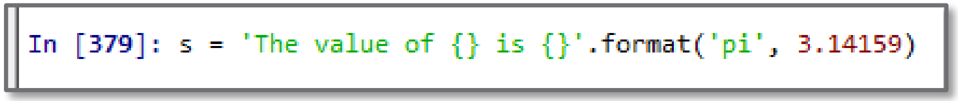
# `.format()` – Syntax & Terminology

```
<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

- Replacement fields:
  - Enclosed in curly brackets, { }

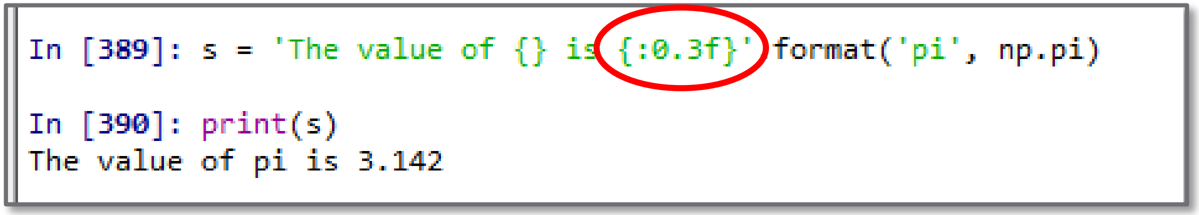


```
In [379]: s = 'The value of {} is {}'.format('pi', 3.14159)
```

  - 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 {:0.3f}' format('pi', np.pi)

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

# Formatting Strings – `format_spec`

- **`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.

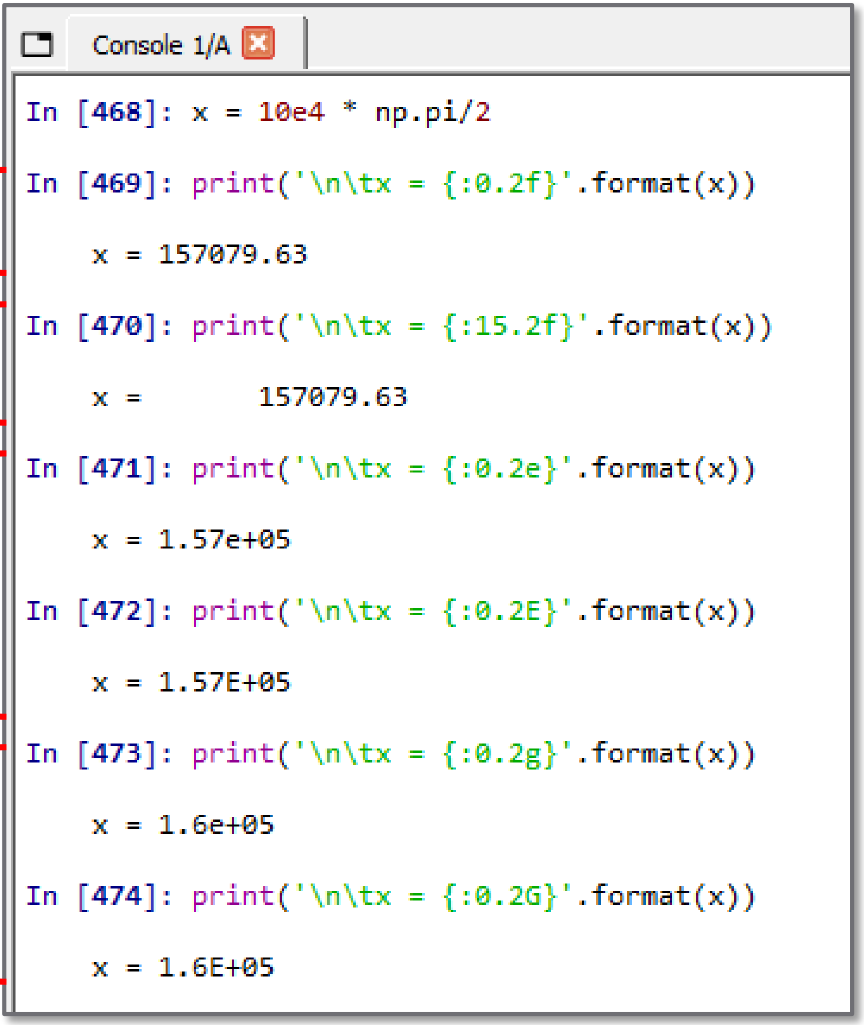
# `format_spec – type`

- 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<br>(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

- Fixed-point notation
- Field-width control
- Exponential notation
- Compact format
  - Note that .prec specifies number of significant figures for g or G type



```
In [468]: x = 10e4 * np.pi/2

In [469]: print('\n\tx = {:0.2f}'.format(x))

    x = 157079.63

In [470]: print('\n\tx = {:15.2f}'.format(x))

    x =       157079.63

In [471]: print('\n\tx = {:0.2e}'.format(x))

    x = 1.57e+05

In [472]: print('\n\tx = {:0.2E}'.format(x))

    x = 1.57E+05

In [473]: print('\n\tx = {:0.2g}'.format(x))

    x = 1.6e+05

In [474]: print('\n\tx = {:0.2G}'.format(x))

    x = 1.6E+05
```