First Steps with Python and Jupyter

Part 1 of "A Gentle Introduction to Programming with Python"

This tutorial is the first in a series of beginner-friendly tutorials on programming using the Python language. These tutorials take a practical coding-based approach, and the best way to learn the material is to execute the code and experiment with the examples. Check out the full series here:

1. First Steps with Python and Jupyter
2. A Quick Tour of Variables and Data Types
3. Branching using Conditional Statements and Loops
4. Writing Reusable Code Using Functions
5. Reading from and Writing to Files
6. Object Oriented Programming with Classes

The following topics are covered in this tutorial:

- Performing arithmetic operations using Python
- Solving multi-step problems using variables
- Evaluating conditions using Python
- Combining conditions with logical operators
- Adding text styles using Markdown

How to run the code

This tutorial is an executable *Jupyter notebook* hosted on Jovian.ml, a platform for sharing data science projects online (don't worry if these terms seem unfamiliar, we'll learn more about them soon). You can "run" this tutorial and experiment with the code examples in a couple of ways: *using free online resources* (recommended) or *on your own computer*.

Option 1: Running using free online resources (1-click, recommended)

The easiest way to start executing this notebook is to click the "Run" button at the top of this page, and select "Run on Binder". This will run the notebook on mybinder.org, a free online service for running Jupyter notebooks. You can also select "Run on Colab" or "Run on Kaggle", but you'll need to create an account on Google Colab or Kaggle to use these platforms.

Option 2: Running on your computer locally

To run this notebook on your computer locally, you'll need to set up Python and download the notebook. We recommend using the Conda distribution of Python. Here's what you need to do to get started:

1. Install Conda by following these instructions. Make sure to add Conda binaries to your system `PATH` to be able to run the `conda` command line tool from your Mac/Linux terminal or Windows command prompt.
2. Create and activate a Conda virtual environment called `zerotopandas` which you can use for this tutorial series, by running the follwing commands on your terminal / command prompt:

   ```
   conda create -n intro-to-python -y python=3.8
   conda activate intro-to-python
   ```

   You'll need to create the environment only once, but you'll have to activate it every time want to run the notebook. When the environment is activated, you should be able to see a prefix `(intro-to-python)` within your terminal or command prompt.
3. Install the required Python libraries within the environment by the running the following command:

   ```
   pip install jovian jupyter numpy pandas matplotlib seaborn --upgrade
   ```

4. Download the notebook for this tutorial using the `jovian clone` command:

   ```
   jovian clone aakashns/first-steps-with-python
   ```

   The notebook is downloaded to the directory `first-steps-with-python`. You can also use the "Download Zip" option on the page instead of using the `jovian clone` command.

5. Enter the project directory and start the Jupyter notebook:

```
cd first-steps-with-python
jupyter notebook
```

6. You can now access Jupyter's web interface by clicking the link that shows up on the terminal or by visiting http://localhost:8888 on your browser. Click on the notebook `first-steps-with-python.ipynb` to open it and run the code. If you want to type out the code yourself, you can also create a new notebook using the "New" button.

> **Jupyter Notebooks**: This tutorial is a Jupyter notebook - a document made of "cells", which can contain explanations in text or code written in Python. Code cells can be executed and their outputs e.g. numbers, messages, graphs, tables, files etc. can be viewed within the notebook, which makes it a really powerful platform for experimentation and analysis. Don't afraid to experiment with the code & break things - you'll learn a lot by encoutering and fixing errors. You can use the "Kernel > Restart & Clear Output" menu option to clear all outputs and start again from the top of the notebook.

Performing Arithmetic Operations using Python

Let's begin by using Python as a calculator. You can write and execute Python using a code cell within Jupyter.

> **Working with Cells**: To create a new cell within Jupyter, you can select "Insert > Insert Cell Below" from the menu bar or just press the "+" button on the toolbar. You can also use the keyboard shortcut `Esc+B` to create a new cell. Once a cell is created, click on it to select it. You can then change the cell type to code or markdown (text) using "Cell > Cell Type" menu option. You can also use the keyboard shortcuts `Esc+Y` and `Esc+M`. Double click a cell to edit the content within the cell. To apply your changes and run a cell, use the "Cell > Run Cells" menu option or click the "Run" button on the toolbar or just use the keyboard shortcut `Shift+Enter`. You can see a full list of keyboard shortcuts using the "Help > Keyboard Shortcuts" menu option.

Run the code cells below to perform calculations and view their result. Try changing the numbers and run the changed cells again to see updated results. Can you guess what the `//`, `%` and `**` operators are used for?

```
2 + 3 + 9
```

```
14
```

```
99 - 73
```

```
26
```

```
23.54 * -1432
```

```
-33709.28
```

```
100 / 7
```

```
14.285714285714286
```

```
100 // 7
```

```
14
```

```
100 % 7
```

```
2
```

```
5 ** 3
```

```
125
```

As you might expect, certain operators like `/` and `*` take precendence over other operators like `+` and `-` as per mathematical conventions. You can use parantheses i.e. `(` and `)` to specify the order in which operations are performed.

```
((2 + 5) * (17 - 3)) / (4 ** 3)
```

```
1.53125
```

Python supports the following arithmetic operators:

Operator	Purpose	Example	Result
`+`	Addition	`2 + 3`	`5`
`-`	Subtraction	`3 - 2`	`1`
`*`	Multiplication	`8 * 12`	`96`
`/`	Division	`100 / 7`	`14.28..`
`//`	Floor Division	`100 // 7`	`14`
`%`	Modulus/Remainder	`100 % 7`	`2`
`**`	Exponent	`5 ** 3`	`125`

Try solving some simple problems from this page: https://www.math-only-math.com/worksheet-on-word-problems-on-four-operations.html . You can use the empty cells below and add more cells if required.

```
Start coding or generate with AI.
```

```
Start coding or generate with AI.
```

```
Start coding or generate with AI.
```

```
Start coding or generate with AI.
```

Solving multi-step problems using variables

Let's try solving the following word problem using Python:

> A grocery store sells a bag of ice for $1.25, and makes 20% profit. If it sells 500 bags of ice, how much total profit does it make?

We can list out the information provided, and gradually convert the word problem into a mathematical expression which can be evaluated using Python.

Cost of ice bag ($) = 1.25

Profit margin = 20% = .2

Profit per bag ($) = profit margin * cost of ice bag = .2 * 1.25

No. of bags = 500

Total profit = no. of bags * profit per bag = 500 * (.2 * 1.25)

```
500 * (.2 * 1.25)
```

```
125.0
```

Thus, the grocery store makes a total profit of $125.0 . While this is a reasonable way to solve a problem, it's not quite clear by looking at the code cell what the numbers represent. We can give names to each of the numbers by creating Python *variables*.

> **Variables**: While working with a programming language such as Python, informations is stored in *variables*. You can think of variables as containers for storing data. The data stored within a variable is called it's *value*.

```
cost_of_ice_bag = 1.25
```

```
profit_margin = .2
```

```
number_of_bags = 500
```

The variables `cost_of_ice_bag`, `profit_margin` and `number_of_bags` now contain the information provided in the word problem. We can check the value of a variable by typing its name into a cell, and we can combine variables using arithmetic operations to create other variables.

> Tip: While typing the name of an existing variable in a code cell within Jupyter, you can type the first few characters and press the `Tab` key to autocomplete the variable's name. Try typing `pro` in a code cell below and press `Tab` to autocomplete to `profit_margin`.

```
profit_margin
```

```
0.2
```

```
profit_per_bag = cost_of_ice_bag * profit_margin
```

```
profit_per_bag
```

```
0.25
```

```
total_profit = number_of_bags * profit_per_bag
```

```
total_profit
```

```
125.0
```

If you try to see the value of a variable that has not been *defined* i.e. given a value using the assignment statement `variable_name = value`, then Python shows an error.

```
# net_profit
```

Storing and manipulating data using appropriately named variables is a great way to explain what your code does.

Let's display the result of the word problem using a friendly message. We can do this using the `print` *function*.

> **Functions**: A function is a reusable set of instructions. A function takes one or more inputs, performs certain operations, and often returns an output. Python provides many in-built functions like `print`, and also allows us to define our own functions.

```
print("The grocery store makes a total profit of $", total_profit)
```

```
The grocery store makes a total profit of $ 125.0
```

> **`print`**: The `print` function is used to display information. It takes one or more inputs which can be text (within quotes e.g. `"this is some text"`), numbers, variables, mathematical expressions etc. We'll learn more about variables & functions in the next section.

Creating a code cell for each variable or mathematical operation can get tedious. Fortunately, Jupyter allows you write multiple lines of code within a single code cell. Let's rewrite the solution to our word problem within a single cell.

```
# Store input data in variables
cost_of_ice_bag = 1.25
profit_margin = .2
number_of_bags = 500

# Perform the required calculations
profit_per_bag = cost_of_ice_bag * profit_margin
total_profit = number_of_bags * profit_per_bag

# Display the result
print("The grocery store makes a total profit of $", total_profit)
```

```
The grocery store makes a total profit of $ 125.0
```

Note that we're using the `#` character to add *comments* within our code.

> **Comments**: Comments and blank lines are ignored during execution, but they are useful for providing information to other humans (including yourself) about what the code does. Comments can be inline (at the end of some code), on a separate line, or even span multiple lines.

Inline and single line comments start with `#`, whereas multi-line comments begin and end with three quotes i.e `"""`. Here are some examples of code comments:

```
my_favorite_number = 1 # an inline comment
```

```
# This comment gets its own line
my_least_favorite_number = 3
```

```
"""This is a multi-line comment.
Write as little or as much as you'd like.

Comments are really helpful for people reading
your code, but try to keep them short & to-the-point.

Also, if you use good variable names, then your code is
often self explanatory, and you may not even need comments!
"""
a_neutral_number = 5
```

Evaluating conditions using Python

Apart from arithmetic operations, Python also provides several oprations for comparing numbers & variables.

Operator	Description
==	Check if operands are equal
!=	Check if operands are not equal
>	Check if left operand is greater than right operand
<	Check if left operand is less than right operand
>=	Check if left operand is greater than or equal to right operand
<=	Check if left operand is less than or equal to right operand

The result of a comparision operation is either `True` or `False` (note the uppercase `T` and `F`). These are special keywords in Python. Let's try out some experiment with comparision operators.

```
my_favorite_number = 1
my_least_favorite_number = 5
a_neutral_number = 3
```

```
# Equality check - True
my_favorite_number == 1
```

```
True
```

```
# Equality check - False
my_favorite_number == my_least_favorite_number
```

```
False
```

```
# Not equal check - True
my_favorite_number != a_neutral_number
```

```
True
```

```
# Not equal check - False
a_neutral_number != 3
```

```
False
```

```
# Greater than check - True
my_least_favorite_number > a_neutral_number
```

```
True
```

```
# Greater than check - False
my_favorite_number > my_least_favorite_number
```

```
False
```

```
# Less than check - True
my_favorite_number < 10
```

```
True
```

```
# Less than check - False
my_least_favorite_number < my_favorite_number
```

```
False
```

```
# Greater than or equal check - True
my_favorite_number >= 1
```

```
True
```

```
# Greater than or equal check - False
my_favorite_number >= 3
```

```
False
```

```
# Less than or equal check - True
3 + 6 <= 9
```

```
True
```

```
# Less than or equal check - False
my_favorite_number + a_neutral_number <= 3
```

```
False
```

Just like arithmetic operations, the result of a comparison opration can also be stored in a variable.

```
cost_of_ice_bag = 1.25
is_ice_bag_expensive = cost_of_ice_bag >= 10
print("Is the ice bag expensive?", is_ice_bag_expensive)
```

```
Is the ice bag expensive? False
```

Combining conditions with logical operators

The logical operators `and`, `or` and `not` operate upon conditions and `True` & `False` values (also known as *booleans*). `and` and `or` operate on two conditions, whereas `not` operates on a single condition.

The `and` operator returns `True` when both the conditions evalute to `True`. Otherwise it returns `False`.

a	b	a and b
True	True	True
True	False	False
False	True	False
False	False	False

```
my_favorite_number
```

```
1
```

```
my_favorite_number > 0 and my_favorite_number <= 3
```

```
True
```

```
my_favorite_number < 0 and my_favorite_number <= 3
```

```
False
```

```
my_favorite_number > 0 and my_favorite_number >= 3
```

```
False
```

```
True and False
```

```
False
```

```
True and True
```

```
True
```

The `or` operator returns `True` if at least one of the conditions evalute to `True`. It returns `False` only if both conditions are `False`.

a	b	a or b
True	True	True
True	False	True
False	True	True
False	False	False

```
a_neutral_number = 3
```

```
a_neutral_number == 3 or my_favorite_number < 0
```

```
True
```

```
a_neutral_number != 3 or my_favorite_number < 0
```

```
False
```

```
my_favorite_number < 0 or True
```

```
True
```

```
False or False
```

```
False
```

The `not` operator returns `False` if a condition is `True` and `True` if the condition is `False`.

```
not a_neutral_number == 3
```

```
False
```

```
not my_favorite_number < 0
```

```
True
```

```
not False
```

```
True
```

```
not True
```

```
False
```

Logical operators can be combined to form complex conditions. Use round brackets or parantheses `(` and `)` to indicate the order in which logical operators should be applied.

```
(2 > 3 and 4 <= 5) or not (my_favorite_number < 0 and True)
```

```
True
```

```
not (True and 0 < 1) or (False and True)
```

```
False
```

If parantheses are not used, logical operators are applied from left to right.

```
not True and 0 < 1 or False and True
```

```
False
```

Experiment with arithmetic, conditional and logical operators in Python using the interactive nature of Jupyter notebook. We will learn more about variables and functions in future tutorials.

Adding text styles using Markdown

Adding explanations using text cells (like this one) is great way to make your notebook informative for other readers, and for yourself, if you need to refer back to it in the future. Double click on a text cell within Jupyter to edit it. In the edit mode, you'll notice that the text looks a little different (for instance the heading has a `##` prefix. This text is writted using Markdown, a simple way to add styles to your text. Execute this cell to see the output without the special characters. You can switch back and forth between the source and the output to see how to create a specific style.

For, instance, you can use one or more `#` characters at the start of a line to create headers of different sizes:

Header 1

Header 2

Header 3

Header 4

To create a bulleted or numbered list, simply start a line with `*` or `1.` .

A bulleted list:

- Item 1
- Item 2
- Item 3

A numbered list:

1. Apple
2. Banana
3. Pineapple

You can make some text bold using `**` e.g. **this is some bold text**, or make it italic using `*` e.g. *this is some italic text*. You can also create links e.g. this is a link. Images are easily embedded too:

Jovian.ml

Another really nice feature of Markdown is ability to include blocks of code. Note that code blocks inside Markdown cells cannot be executed.

```
# Perform the required calculations
profit_per_bag = cost_of_ice_bag * profit_margin
total_profit = number_of_bags * profit_per_bag

# Display the result
print("The grocery store makes a total profit of $", total_profit)
```

You can learn the full syntax of Markdown here: https://learnxinyminutes.com/docs/markdown/

Save and upload your notebook

Whether you're running this Jupyter notebook on an online service like Binder or on your local machine, it's important to save your work from time, so that you can access it later, or share it online. You can upload this notebook to your Jovian.ml account using the `jovian` Python library.

First, you need to install the Jovian python library, if it isn't already installed.

```
!pip install jovian --upgrade --quiet
```

```
  Preparing metadata (setup.py) ... done
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 68.6/68.6 kB 3.3 MB/s eta 0:00:00
  Building wheel for uuid (setup.py) ... done
```

Next, the library needs to be imported.

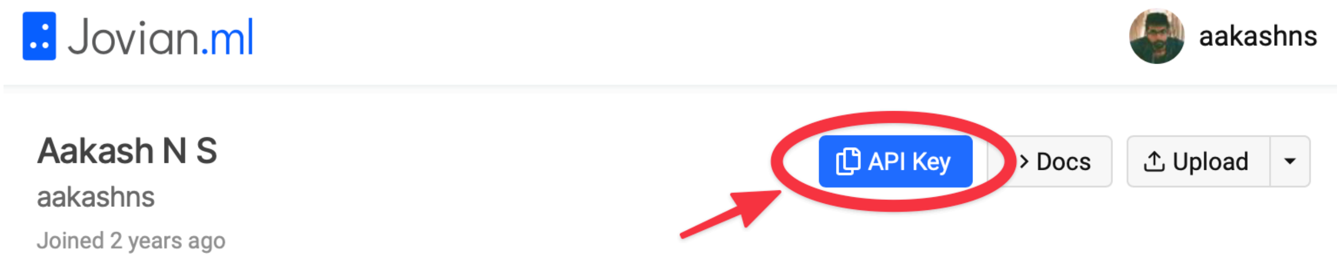
```
import jovian
```

Finally, you can run `jovian.commit` to capture and upload a snapshot of the notebook.

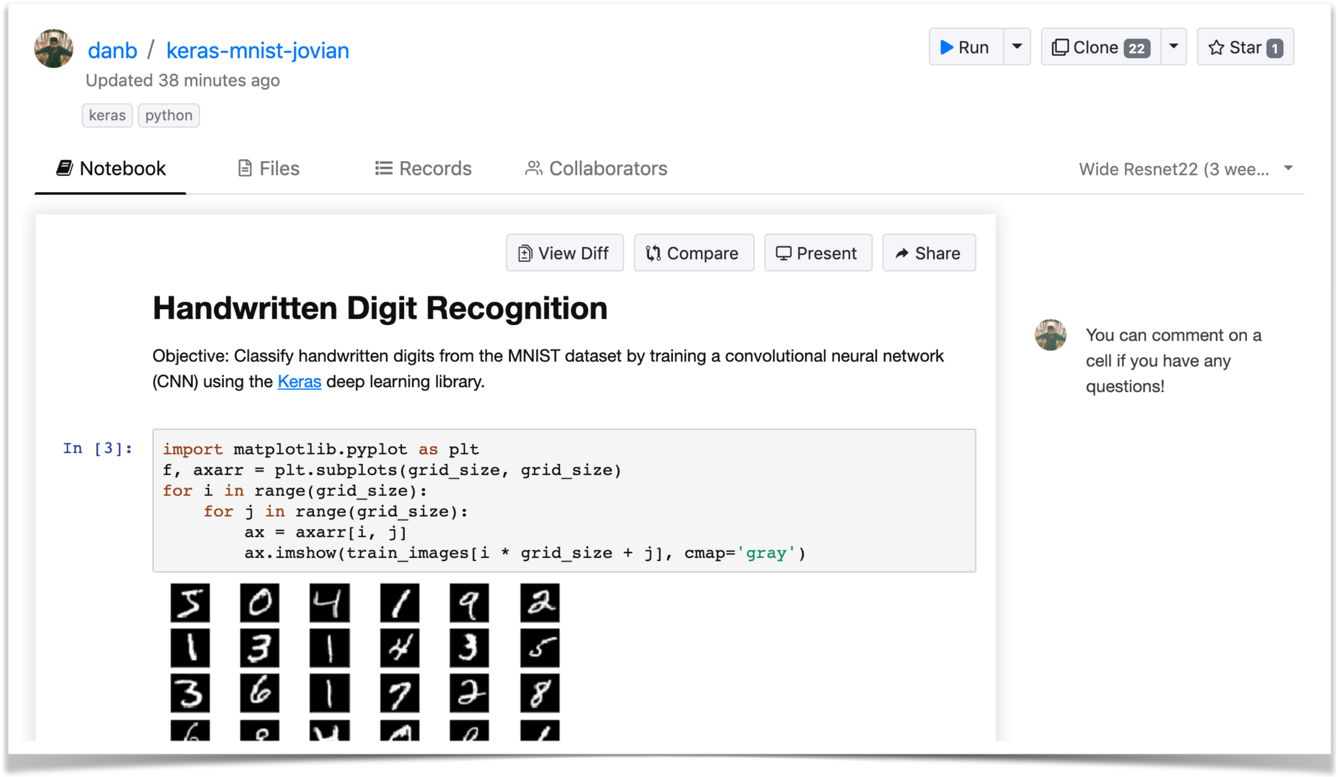
```
jovian.commit(project='first-steps-with-python')
```

```
[jovian] Detected Colab notebook...
[jovian] jovian.commit() is no longer required on Google Colab. If you ran this notebook from Jovian,
then just save this file in Colab using Ctrl+S/Cmd+S and it will be updated on Jovian.
Also, you can also delete this cell, it's no longer necessary.
```

The first time you run `jovian.commit`, you'll be asked to provide an API Key, to securely upload the notebook to your Jovian.ml account. You can get the API key from your Jovian.ml profile page after logging in / signing up.



`jovian.commit` uploads the notebook to your Jovian.ml account, captures the Python environment and creates a shareable link for your notebook as shown above. You can use this link to share your work and let anyone (including you) run your notebooks and reproduce your work. Jovian also includes a powerful commenting interface, so you can discuss & comment on specific parts of your notebook:



Handwritten Digit Recognition

Objective: Classify handwritten digits from the MNIST dataset by training a convolutional neural network (CNN) using the Keras deep learning library.

```
In [3]: import matplotlib.pyplot as plt
        f, axarr = plt.subplots(grid_size, grid_size)
        for i in range(grid_size):
            for j in range(grid_size):
                ax = axarr[i, j]
                ax.imshow(train_images[i * grid_size + j], cmap='gray')
```

You can do a lot more with the `jovian` Python library. Visit the documentation site to learn more: https://jovian.ml/docs/index.html

Further Reading and References

Following are some resources to learn about more arithmetic, conditional and logical operations in Python:

- Python Tutorial at W3Schools: https://www.w3schools.com/python/
- Practical Python Programming: https://dabeaz-course.github.io/practical-python/Notes/Contents.html
- Python official documentation: https://docs.python.org/3/tutorial/index.html

Now that you have taken your first steps with Python, you are ready to move on to the next tutorial: "A Quick Tour of Variables and Data Types in Python".