Python Notebook from Zero to Advanced

None

Zhenyuan Lu

Table of contents

1. Python Notebook from Zero to Advanced

Author: Zhenyuan Lu

Version: 1.0.0

Created: 12/05/2022, *last modified on 3/31/2023*

Welcome to the Python Notebook from Zero to Advanced, an enjoyable place on the internet to learn Python (besides the best place official Python documentation, of course 😉)

Python Notebook from Zero to Advanced

1.0.1 About This Book

This book is designed for those with little or no Python programming experience, and it is filled with concise, easy-to-understand examples that will help you learn quickly and effectively.

Throughout this comprehensive guide, we'll cover a wide range of topics, including data types, control structures, functions, and more.

2. Disclaimer

I am pretty sure there are some typing errors, spelling mistakes, and other inaccuracies. If you find any such issues, please do not hesitate to contact me via lu [dot] zhenyua [at] northeastern [dot] edu.

This tutorial is aimed to those who have zero or less Python programming experience. The content has been created mainly based on the official Python documentation, but with more concise and simple examples throughout the entire book. If you believe any content has been used inappropriately, please let me know, and I will address the issue.

3. Get Started

Welcome! Python is a versatile and powerful programming language, widely used for web development, data analysis, artificial intelligence, and more. This tutorial will guide you through the installation of Python and setting up a Python development environment in PyCharm, a popular integrated development environment (IDE) for Python.

To get started with Python, you'll first need to install it on your computer. Follow the instructions below for your operating system:

3.1 1.1. Installation with Anaconda

Anaconda is a popular distribution of Python and R programming languages, which simplifies package management and deployment. It comes with many pre-installed packages and tools for data science and machine learning.

For Windows:

Visit the Anaconda website at https://www.anaconda.com/products/distribution and download the installer for your operating system. Run the installer and follow the installation instructions. After the installation is complete, you can verify the installation by opening a terminal (or Anaconda Prompt on Windows) and typing conda --version. You should see the installed Anaconda version displayed. To check the Python version, type `python --version`.

For macOS:

Visit the Anaconda website at https://www.anaconda.com/products/distribution and download the installer for macOS. Open the downloaded package (.pkg file) and follow the installation instructions. After the installation is complete, you can verify the installation by opening Terminal and typing conda --version. You should see the installed Anaconda version displayed. To check the Python version, type `python --version`.

3.2 1.2. Standard Installation

For Windows:

Visit the official Python website at https://www.python.org/downloads/ and download the latest version of Python. Run the installer. Be sure to check the box "Add Python to PATH" before clicking "Install Now." This will make it easier to run Python from the command prompt. After the installation is complete, you can verify the installation by opening a command prompt and typing `python --version`. You should see the installed Python version displayed.

For macOS:

Visit the official Python website at https://www.python.org/downloads/ and download the latest version of Python. Open the downloaded package and follow the installation instructions. After the installation is complete, you can verify the installation by opening Terminal and typing `python --version`. You should see the installed Python version displayed.

3.3 1.2. Setting up a Python Development Environment in PyCharm

Now that Python is installed, let's set up a development environment in PyCharm.

1. Download and install PyCharm from https://www.jetbrains.com/pycharm/download/. There are two editions available: Community Edition (free) and Professional Edition (paid). For this tutorial, the Community Edition is sufficient.
2. Open PyCharm and create a new project by clicking "Create New Project" on the welcome screen.
3. Choose a location for your project and make sure the "Python Interpreter" field is set to the Python version you installed earlier. If not, click the gear icon next to the field and select "Add Interpreter." Choose "System Interpreter" and select the Python executable from the list. Click "Create" to create your new Python project.
4. You're now ready to start writing Python code! In the next chapter, we'll dive into Python basics, including syntax, variables, and data types.

4. 2. Data Types

4.1 2.1. Numbers

In this section, we will cover the different number types in Python, such as integers and floating-point numbers, and how to work with them.

4.1.1 2.1.1. Integers and Floats

Python has two primary numeric types: integers (int) and floating-point numbers (float).

Assign an integer `5` to a variable named `num`. The `print()` function is then used to output the value of `num` to the console.

Input

```
# Integers
num = 5
print(num)
```

Output

```
5
```

Assign a floating-point number `5.2` to a variable named `num`. The `print()` function is then used to output the value of `num` to the console.

Input

```
# Floats
num = 5.2
print(num)
```

Output

```
5.2
```

4.1.2 2.1.2. `type()` and `__class__`

Info

`type()` is a built-in function that returns the type of an object. It is the same as calling the object's `__class__` attribute, e.g. `object.__class__`, but which is less commonly used.

Use `type()` to check the type of a variable.

Input

```
num = 5
print(type(num))
```

Output

```
<class 'int'>
```

The num variable is a floating-point number, so the `type()` function returns `<class 'float'>`.

Input

```
# Floats
num = 5.2
print(type(num))
```

Output

```
<class 'float'>
```

Floats with scientific notation.

Input

```
num = 5.2e3
print(type(num))
```

Output

```
<class 'float'>
```

Use `__class__` to check the type of a variable.

Input

```
print(num.__class__)
```

Output

```
<class 'int'>
```

4.1.3 2.1.3. Math Functions

Python supports various mathematical operations that can be performed on numbers, such as addition, subtraction, multiplication, division, and more. Here's a list of common mathematical operations and their corresponding symbols:

- Addition: `+`
- Subtraction: `-`
- Multiplication: `*`
- Division: `/`
- Floor Division: `//`
- Exponentiation: `**`
- Modulus: `%`

Let's see some examples of using these mathematical operations:

Addition Subtraction Multiplication Division Floor Division Exponentiation Modulus

Input

```
num = 5
print(num + 2)
```

Output

```
7
```

Input

```
num = 5
print(num - 2)
```

Output

```
3
```

Input

```
num = 5
print(num * 2)
```

Output

```
10
```

Input

```
num = 5
print(num / 2)
```

Output

```
2.5
```

Input

```
num = 5
print(num // 2)
```

Output

```
2
```

Input

```
num = 5
print(num ** 2)
```

Output

```
25
```

Input

```
num = 5
print(num % 2)
```

Output

```
1
```

These operations can be used in combination, following the standard order of operations (PEMDAS), to perform more complex calculations. Parentheses can be used to specify the order of operations explicitly.

`abs()` and `round()`

`abs()` and `round()` are two built-in functions that can be used to perform mathematical operations on numbers.

The `abs()` function returns the absolute value of a number.

Input

```
# abs() function
print(abs(-5))
```

Output

```
5
```

The `round()` function rounds a number to a specified number of decimal places.

Input

```
# round() function
print(round(5.75))
```

Output

```
6
```

`round()` with 2nd argument to specify the number of decimal places. Here we round `5.75` to 1 decimal place.

Input

```
# round() with 2nd argument
print(round(5.75, 1))
```

Output

```
5.8
```

4.1.4 2.1.4. Increment and Decrement

In Python, you can increment or decrement the value of a variable using the `+=` and `-=` operators, respectively. These operators are shorthand for adding or subtracting a value to the variable and then assigning the result back to the variable.

We use `num = num + 1` to increment the value of `num` by 1.

Input

```
# Increment
num = 5
num = num + 1
print(num)
```

Output

```
6
```

Increment using shorthand `+=` operator. The `+=` operator is equivalent to `num = num + 1`.

Input

```
# Increment using shorthand +=
num = 5
num += 1
print(num)
```

Output

```
6
```

The `num = num - 1` is used to decrement the value of `num` by 1.

Input

```
# Decrement
num = 5
num = num - 1
print(num)
```

Output

```
4
```

Decrement using shorthand `-=` operator. The `-=` operator is equivalent to `num = num - 1`.

Input

```
# Decrement using shorthand -=
num = 5
num -= 1
print(num)
```

Output

```
4
```

Using the `+=` and `-=` operators can make your code shorter and more readable, especially when performing multiple increment or decrement operations on the same variable.

4.1.5 2.1.5. Comparison Operators

- Equal: `==`
- Not Equal: `!=`
- Greater Than: `>`
- Less Than: `<`
- Greater or Equal: `>=`
- Less or Equal: `<=`

Setting

```
num_1 = 5
num_2 = 2
```

Equal: == **Not Equal: !=** **Greater Than: >** **Less Than: <** **Greater or Equal: >=** **Less or Equal: <=**

Input

```
print(num_1 == num_2)
```

Output

```
False
```

Input

```
print(num_1 != num_2)
```

Output

```
True
```

Input

```
print(num_1 > num_2)
```

Output

```
True
```

Input

```
print(num_1 < num_2)
```

Output

```
False
```

Input

```
print(num_1 >= num_2)
```

Output

```
True
```

Input

```
print(num_1 <= num_2)
```

Output

```
False
```

4.1.6 2.1.6. Casting

Casting is the process of converting a value from one data type to another. In Python, casting is achieved using built-in functions like `int()`, `float()`, and `complex()`.

For example, when working with numbers, you might need to convert a string to an integer or a float. This is useful when you want to perform mathematical operations on string representations of numbers.

`int()` : Convert a value to an integer `float()` : Convert a value to a float `complex()` : Convert a value to a complex number

Check the type of variable `num_1`.

Input

```
num_1 = '5'
print(type(num_1))
```

Output

```
<class 'str'>
```

If we have two numbers as strings, when we `+` them, they are concatenated instead of added.

Input

```
num_1 = '5'
num_2 = '2'
print(num_1 + num_2)
```

Output

```
52
```

If we want to add them, we need to convert them to integers first.

Input

```
# Convert string to int
num_1 = int(num_1)
num_2 = int(num_2)
print(num_1 + num_2)
```

Output

```
7
```

If we convert a floating number to an integer, the decimal part will be removed.

Input

```
num = 5.2
print(int(num))
```

Output

```
5
```

If we convert an integer to a float, the result will be a float with `.0` at the end.

Input

```
# float()
num = 5
print(float(num))
```

Output

```
5.0
```

If we convert an integer to a complex number, the result will be a complex number with `j` at the end.

Input

```
# complex()
num = 5
print(complex(num))
```

Output

```
(5+0j)
```

`zfill()` method adds zeros (0) at the beginning of the string, until it reaches the specified length.

Input

```
# zfill method
num = 5
print(str(num).zfill(3))
```

Output

```
005
```

4.2 2.2. Strings

Strings are one of the most important and commonly used data types in Python. A string is simply a sequence of characters, such as letters, numbers, and symbols. In Python, strings are created using either single quotes `'<str>'` or double quotes `"<str>"`. This tutorial will cover the basics of string manipulation in Python, including string indexing, slicing, concatenation, formatting, and various string methods.

A string variable named sentence is defined and assigned the value `"Hello World"`. The `print()` function is then used to output the value of sentence to the console. This code demonstrates how to create and output a basic string in Python.

Input

```
sentence = 'Hello World'
print(sentence)
```

Output

```
Hello World
```

4.2.1 2.2.1. String Basics

Quotes

Single quotes are faster, more readable, and more commonly used than double quotes in Python. However, if a string itself contains a single quote, then it must be escaped using a backslash \ so that Python can properly interpret the string.

In the example code, a new string variable named sentence is defined using single quotes and contains the word `"Tub's World"`. To escape the single quote in the middle of the string, a backslash is used before it. The `print()` function is then used to output the value of sentence to the console.

Input

```
# Single quote
# Use backslash to escape single quote
sentence = 'Tub\'s World'
print(sentence)
```

Output

```
Tub's World
```

The following is the same example as above, but using double quotes instead of single quotes.

Input

```
# Use double quote
sentence = "Tub's World"
print(sentence)
```

Output

```
Tub's World
```

Triple quotes are used to create multi-line strings. In the example code, a new string variable named sentence is defined using triple quotes and contains the string `"Tub's World"` and `"is a good place to live in"`. The `print()` function is then used to output the value of sentence to the console.

Input

```
# Three double quotes for multi-line string
sentence = """Tub's World
is a good place to live in"""
print(sentence)
```

Output

```
Tub's World
is a good place to live in
```

Length of a String

The following code demonstrates how to use the `len()` function to get the length of a string.

Input

```
# Use len() to get the length of a string
sentence = "Tub's World"
print(len(sentence))
```

Output

```
12
```

Upper and Lower Case

The `lower()` method converts all the characters in a string to lowercase. This is useful for case-insensitive comparisons or normalization of text data.

Input

```
# Lowercase (all letters)
sentence = "Tub's World"
print(sentence.lower())
```

Output

```
tub's world
```

The `upper()` method converts all the characters in a string to uppercase.

Input

```
# Uppercase (all letters)
sentence = "Tub's World"
print(sentence.upper())
```

Output

```
TUB'S WORLD
```

Input

```
# Capitalize
sentence = "Tub's World"
print(sentence.capitalize())
```

Output

```
Tub's world
```

The `capitalize()` method capitalizes the first character of a string and makes the rest of the characters lowercase.

4.2.2 2.2.2. String Indexing

Indexing allows us to access individual characters within a string using their position, also known as the index. It is essential to understand that in Python, indexing starts from 0, meaning the first character in the string has an index of 0, the second character has an index of 1, and so on.

Here, we define a string sentence containing the text `"Tub's World"`. We then use indexing to access the character at position 0, which is `'T'`. The `print()` function displays the output, confirming that the first character in the string is indeed `'T'`.

Input

```
# String indexing
# Indexing starts from 0
sentence = "Tub's World"
print(sentence[0])
```

Output

```
T
```

Now, let's see how negative indexing works in Python:

Input

```
# If index is negative, it starts from the end
sentence = "Tub's World"
print(sentence[-1])
```

Output

```
d
```

Accessing an index that is out of range will result in an error.

Input

```
# If we input 11, it will throw an error
sentence = "Tub's World"
print(sentence[11])
```

Output

```
IndexError: string index out of range
```

4.2.3 2.2.3. String Slicing

String slicing is a technique used to extract a subset of characters from a string. Slicing is done by specifying the starting and ending indices of the slice, separated by a colon. The starting index is inclusive, and the ending index is exclusive. If either the starting or ending index is omitted, it defaults to the beginning or end of the string, respectively.

Input

```
# Slicing starts from 0
# The first index is inclusive, the second index is exclusive
sentence = "Tub's World"
print(sentence[0:3])
```

Output

```
Tub
```

Input

```
# We can also omit the first index
sentence = "Tub's World"
print(sentence[:3])
```

Output

```
Tub
```

Input

```
# We can also omit the second index
sentence = "Tub's World"
print(sentence[3:])
```

Output

```
's World
```

Input

```
# We can also omit both index
sentence = "Tub's World"
print(sentence[:])
```

Output

```
Tub's World
```

4.2.4 2.2.4. `count()`, `find()`, and `replace()`

The `count()`, `find()`, and `replace()` methods are used to count, find, and replace substrings within a string, respectively.

`count()` returns the number of occurrences of a specified substring in the given string.

Input

```
# Count (return the number of occurrences)
sentence = "Tub's World"
print(sentence.count('o'))
```

Output

```
1
```

`find()` returns the index of the first occurrence of a specified substring in the given string.

Input

```
# Find (return the index of the first occurrence)
sentence = "Tub's World"
print(sentence.find('o'))
```

Output

```
8
```

`replace()` replaces all occurrences of a specified substring (old) with a new substring in the given string.

Input

```
# Replace (replace old with new)
sentence = "Tub's World"
print(sentence.replace('Tub', 'Tom'))
```

Output

```
Tom's World
```

We can also assign the result of the replace() method back to the same variable if we want to update the original string:

Input

```
# We can also assign the result to the same variable
sentence = "Tub's World"
sentence = sentence.replace('Tub', 'Tom')
print(sentence)
```

Output

```
Tom's World
```

4.2.5 2.2.5. String Concatenation

String concatenation is a technique used to join two or more strings together. In Python, string concatenation is done using the `+` operator.

Input

```
# String concatenation
name = 'Tub'
age = 5
sentence = name + 'is' + str(age) + 'years old'
print(sentence)
```

Output

```
Tub is 5 years old
```

String concatenation is essential when working with dynamic content, such as user input or data from external sources, as it enables you to construct meaningful and contextually relevant strings. When concatenating strings, it's essential to pay attention to spaces and punctuation to ensure the resulting string is formatted correctly. For instance, in this example, we've added a space character between the punctuation and noun variables to separate the words in the final string.

4.2.6 2.2.6. String Formatting

String formatting is a technique used to embed values within a string. In Python, there are several ways to format strings, including using the `format()` method and using f-strings. The `format()` method allows you to embed values within a string using placeholders, which are represented by curly braces `{}`. You can also use named placeholders to improve the readability of your code. F-strings are a more recent addition to Python and provide a more concise and intuitive way to embed values within a string.

Here, we have three string variables: `name`, and an integer variable `age`. We introduce three methods for including non-string variables in a string using string formatting:

Input: Setting up variables

```
# If we want to concatenate a lot of strings,
# it is better to use string formatting
name = 'Tub'
age = 5
```

Method 1: Using the `format()` method

Input

```
# Method 1
# Use format() method
sentence = '{} is {} years old'.format(name, age)
print(sentence)
```

Output

```
Tub is 5 years old
```

Method 2: Using the `format()` method with keyword arguments

Input

```
# Method 2
# use format() method with keyword arguments
sentence = '{n} is {a} years old'.format(n=name, a=age)
print(sentence)
```

Output

```
Tub is 5 years old
```

Method 3: Using `f` -strings (Python 3.6+)

Input

```
# Method 3
# Use f-string (3.6+)
sentence = f'{name} is {age} years old'
print(sentence)
```

Output

```
Tub is 5 years old
```

In addition to the methods above, we can also modify the string content within the string formatting. In this case, we convert the name variable to uppercase using the `.upper()` method:

Input

```
# With upper case
sentence = f'{name.upper()} is {age} years old'
print(sentence)
```

Output

```
TUB is 5 years old
```

All three methods allow you to easily include non-string variables in your string, without the need for explicit type conversion, making them more efficient and readable than traditional string concatenation.

4.2.7 2.2.7. `dir()` and `help()`

The `dir()` function returns a list of all attributes and methods of the specified object, while the `help()` function provides documentation on a specific attribute or method.

We pass the name variable, which is a string, to the dir() function. This will return a list of all available attributes and methods for the string object.

Input

```
# dir() function returns a list of all
# attributes and methods of the specified object
name = 'Tub'
print(dir(name))
```

Output

```
['__add__', '__class__', '__contains__', '__delattr__', '__dir__', '__doc__', '__eq__', '__format__',
 '__ge__', '__getattribute__', '__getitem__', '__getnewargs__', '__getstate__', '__gt__', '__hash__', '__init__',
 '__init_subclass__', '__iter__', '__le__', '__len__', '__lt__', '__mod__', '__mul__', '__ne__', '__new__',
 '__reduce__', '__reduce_ex__', '__repr__', '__rmod__', '__rmul__', '__setattr__', '__sizeof__', '__str__',
 '__subclasshook__', 'capitalize', 'casefold', 'center', 'count', 'encode', 'endswith', 'expandtabs', 'find',
 'format', 'format_map', 'index', 'isalnum', 'isalpha', 'isascii', 'isdecimal', 'isdigit', 'isidentifier', 'islower',
 'isnumeric', 'isprintable', 'isspace', 'istitle', 'isupper', 'join', 'ljust', 'lower', 'lstrip', 'maketrans',
 'partition', 'removeprefix', 'removesuffix', 'replace', 'rfind', 'rindex', 'rjust', 'rpartition', 'rsplit', 'rstrip',
 'split', 'splitlines', 'startswith', 'strip', 'swapcase', 'title', 'translate', 'upper', 'zfill']
```

We can also use `help()` function to display information about the string variable, but instead of passing the variable name, we pass the string object `str` itself.

Input

```
# help() function
name = 'Tub'
print(help(str))
```

Output (partial)

```
Help on class str in module builtins:...
...
```

4.3 2.3. Lists

Lists in Python are ordered, mutable collections of items. They can store elements of different types, such as strings, integers, or other objects.

Creating a list with species names:

Input

```
# A list of species names
species = ['Tub', 'Furrytail', 'Cat', 'Barkalot']
print(species)
```

Output

```
['Tub', 'Furrytail', 'Cat', 'Barkalot']
```

Create an empty list using the `[]` or `list()` function:

Input

```
# Create empty list
empty_list = []
# or
empty_list = list()
print(empty_list)
```

Output

```
[]
```

4.3.1 2.3.1. List Indexing

Check the length of the list using the `len()` function:

Input

```
species = ['Tub', 'Furrytail', 'Cat', 'Barkalot']
print(len(species))
```

Output

```
4
```

In Python the first element of a list has index `0`, which is different from other programming languages, e.g. R, where the first element has index `1`.

Access the first element of the list using the index `0`:

Input

```
# Indexing starts from 0
species = ['Tub', 'Furrytail', 'Cat', 'Barkalot']
print(species[0])
```

Output

```
Tub
```

Access the last element of the list using the index `-1`:

Input

```
# If index is negative, it starts from the end
species = ['Tub', 'Furrytail', 'Cat', 'Barkalot']
print(species[-1])
```

Output

```
Barkalot
```

Access the second element of the list using the index `1`:

Input

```
```python title="Input"
# If we input 4, it will throw an error
species = ['Tub', 'Furrytail', 'Cat', 'Barkalot']
print(species[4])
```

Output

```
IndexError: list index out of range
```

4.3.2 2.3.2. List Slicing

Slicing is a way to access a subset of a list. We can use the colon `:` to specify the start and end index of the slice. The slice will include the start index, but not the end index.

Slice the first two elements of the list:

Input

```
# Slicing starts from 0
species = ['Tub', 'Furrytail', 'Cat', 'Barkalot']
print(species[0:2])
```

Output

```
['Tub', 'Furrytail']
```

If we omit the start index, the slice will start from the beginning of the list:

Input

```
# Omit the first index
species = ['Tub', 'Furrytail', 'Cat', 'Barkalot']
print(species[:2])
```

Output

```
['Tub', 'Furrytail']
```

If we omit the end index, the slice will end at the end of the list:

Input

```
# Omit the second index
species = ['Tub', 'Furrytail', 'Cat', 'Barkalot']
print(species[2:])
```

Output

```
['Cat', 'Barkalot']
```

We can also omit both indices to return the entire list:

Input

```
# Omit both index
species = ['Tub', 'Furrytail', 'Cat', 'Barkalot']
print(species[:])
```

Output

```
['Tub', 'Furrytail', 'Cat', 'Barkalot']
```
```

4.3.3 2.3.2. List Methods

There are many methods that can be used with lists. In the following examples, we will introduce couple of common methods to manipulate the list.

Add an item to the end of the list using the `append()` method:

Input

```
# Add an item to the end of the list
species = ['Tub', 'Furrytail', 'Cat', 'Barkalot']
species.append('Hootsworth ')
print(species)
```

Output

```
['Tub', 'Furrytail', 'Cat', 'Barkalot', 'Hootsworth ']
```

Add an item to a specific index, e.g. `2`, using the `insert()` method:

Input

```
# Add an item to a specific index
species = ['Tub', 'Furrytail', 'Cat', 'Barkalot']
species.insert(2, 'Fish')
print(species)
```

Output

```
['Tub', 'Furrytail', 'Fish', 'Cat', 'Barkalot']
```

Now, we want to insert a list into a list after a specific location, e.g. 0, using the `insert()` method:

Input

```
# Insert a list into a list
species = ['Tub', 'Furrytail', 'Cat', 'Barkalot']
species_2 = ['Bumblefluff ', 'Whiskerfloof']
species.insert(0, species_2)
print(species)
```

Output

```
[['Bumblefluff ', 'Whiskerfloof'], 'Tub', 'Furrytail', 'Cat', 'Barkalot']
```

> ⚠ **`insert()` method inserts the list as a single element**
>
> However, this is not what we want because we want to insert the elements of the list `species_2` into the list `species`.

Then we can use the `extend()` instead to do this:

Input

```
# Extend a list
species = ['Tub', 'Furrytail', 'Cat', 'Barkalot']
species_2 = ['Bumblefluff ', 'Whiskerfloof']
species.extend(species_2)
print(species)
```

Output

```
['Tub', 'Furrytail', 'Cat', 'Barkalot', 'Bumblefluff ', 'Whiskerfloof']
```

Now, we have successfully inserted the elements of the list `species_2` into the list `species`.

We can remove an item from the list using the `remove()` method:

Input

```
# Remove an item from the list
species = ['Tub', 'Furrytail', 'Cat', 'Barkalot']
species.remove('Tub')
print(species)
```

Output

```
['Furrytail', 'Cat', 'Barkalot']
```

We can also remove an item from the list using the `pop()` method. If we do not specify the index, it will remove the last item from the list:

Input

```
species = ['Tub', 'Furrytail', 'Cat', 'Barkalot']
species.pop()
print(species)
```

Output

```
['Tub', 'Furrytail', 'Cat']
```

Or we can specify the index to remove the item at that index:

Input

```
species = ['Tub', 'Furrytail', 'Cat', 'Barkalot']
species.pop(1)
print(species)
```

Output

```
['Tub', 'Cat', 'Barkalot']
```

We can also use the `pop()` method to get the item that we removed from the list, and assign it to the `popped_sp` variable:

Input

```
species = ['Tub', 'Furrytail', 'Cat', 'Barkalot']
popped_sp = species.pop()
print(popped_sp)
```

Output

```
Barkalot
```

This is very helpful when we have a queue to keep popping until the queue is empty and we want to keep track of the items that we have popped.

If we want to search for an index of an item in the list, we can use the `index()` method:

Input

```
# Search for an index of an item in the list
species = ['Tub', 'Furrytail', 'Cat', 'Barkalot']
print(species.index('Tub'))
```

Output

```
0
```

We can check if the `'Tub'` is in the species list using the `in` keyword:

Input

```
# If an item is in the list
print('Tub' in species)
```

Output

```
True
```

Sometimes, we want to join a list of strings into a single string. We can use the `join()` method to do this:

Input

```
# Join a list of strings
species = ['Tub', 'Furrytail', 'Cat', 'Barkalot']
species_str = ','.join(species)
print(species_str)
```

Output

```
Tub,Furrytail,Cat,Barkalot
```

Or we can split the single string into a list by a specific character, e.g. `,` :

Input

```
# Split a string into a list by ','
species_str = 'Tub,Furrytail,Cat,Barkalot'
species_list = species_str.split(',')
print(species_list)
```

Output

```
['Tub', 'Furrytail', 'Cat', 'Barkalot']
```

When dealing with a list of numbers, we can use the `min()` , `max()` , and `sum()` functions to get the minimum, maximum, and sum of the numbers in the list:

`min()` `max()` `sum()`

Input

```
nums = [5, 3, 2, 4, 1]
print(min(nums))
```

Output

```
1
```

Input

```
nums = [5, 3, 2, 4, 1]
print(max(nums))
```

Output

```
5
```

Input

```
nums = [5, 3, 2, 4, 1]
print(sum(nums))
```

Output

```
15
```

4.3.4 2.3.3. Sorting and Reversing

We can reverse a list of strings using the `reverse()` method in reverse alphabetical order:

Input

```
# Reverse a list
species = ['Tub', 'Furrytail', 'Cat', 'Barkalot']
```

```
species.reverse()
print(species)
```

Output

```
['Barkalot', 'Cat', 'Furrytail', 'Tub']
```

We can sort a list of strings using the `sort()` method in alphabetical order:

Input

```
# Sort a list
species = ['Tub', 'Furrytail', 'Cat', 'Barkalot']
species.sort()
print(species)
```

Output

```
['Barkalot', 'Cat', 'Furrytail', 'Tub']
```

We can sort a list of numbers using the `sort()` method in alphabetical order:

Input

```
nums = [5, 3, 2, 4, 1]
nums.sort()
print(nums)
```

Output

```
[1, 2, 3, 4, 5]
```

Of course, we can also sort a list of numbers in reverse order by using `sort(reverse = True)`:

Input

```
# Instead of using .reverse(), we can use reverse = True
nums.sort(reverse = True)
print(nums)
```

Output

```
[5, 4, 3, 2, 1]
```

However, the above methods `sort()` and `reverse()` are changing our original variables. What if we want to keep the original variables? We can use the `sorted()` function to sort a list of strings in alphabetical order:

Input

```
species = ['Tub', 'Furrytail', 'Cat', 'Barkalot']
sorted_species = sorted(species)
print(sorted_species)
print(species)
```

Output

```
['Barkalot', 'Cat', 'Furrytail', 'Tub'] # sorted_species
['Tub', 'Furrytail', 'Cat', 'Barkalot'] # original species
```

Here, we don't change the original variable `species` but create a new variable `sorted_species` to store the sorted list. This is useful when we want to keep the original list unchanged.

4.4 2.4. Tuples

Tuples are similar to lists, but they are immutable. This means that we cannot change the contents of a tuple once it is created. Tuples are useful when we want to store a list of items that cannot be changed.

4.4.1 2.4.1. Creating a Tuple

Create an empty tuple is similar to creating an empty list, the only difference is that we use `()` instead of `[]`, or you can use the `tuple()` function:

Input

```
# Create empty tuple
empty_tuple = ()
# or
empty_tuple = tuple()
print(empty_tuple)
```

Output

```
()
```

Create a tuple based on the same strings as the list `species`:

Input

```
species_tup_1 = ('Tub', 'Furrytail', 'Cat', 'Barkalot')
print(species_tup_1)
```

Output

```
('Tub', 'Furrytail', 'Cat', 'Barkalot')
```

4.5 2.5. Immutable vs. Mutable

Immutable means that we cannot change the contents of the object. Mutable means that we can change the contents of the object.

Here we compare list and tuple. List is mutable, and tuple is immutable.

4.5.1 2.5.1. List is Mutable

Input

```
species_1 = ['Tub', 'Furrytail', 'Cat', 'Barkalot']
species_2 = species_1
species_1[1] = 'Furrytail 2'
print(species_1)
print(species_2)
```

Output

```
['Tub', 'Furrytail 2', 'Cat', 'Barkalot']
['Tub', 'Furrytail 2', 'Cat', 'Barkalot']
```

We can see that when we change the contents of `species_1`, the contents of `species_2` also change. This is because `species_1` and `species_2` are both references to the same list in memory.

Python vs. R in Variable Assignment

Python and R handle variable assignment differently, particularly when it comes to mutable objects like lists in Python.

In R, when you assign one variable to another, it creates a copy of the original variable's data. This means that if you change one variable's contents, the other variable's contents remain unchanged. This behavior is known as "copy-on-write" and allows R to save memory by not duplicating data until it is necessary.

In Python, when you assign one variable to another, you are actually creating a reference to the original variable's data, rather than copying the data itself. This means that if you change the contents of one variable, the other variable's contents will also change because they both point to the same data in memory.

If you want to create a new list that is a copy of `species_1` like that in R and not a reference to `species_1`, you can use the `copy()` method:

Input

```
species_1 = ['Tub', 'Furrytail', 'Cat', 'Barkalot']
species_2 = species_1.copy()
species_1[1] = 'Furrytail 2'
print(species_1)
print(species_2)
```

Output

```
['Tub', 'Furrytail 2', 'Cat', 'Barkalot']
['Tub', 'Furrytail', 'Cat', 'Barkalot']
```

We can see that when we change the contents of `species_1`, the `species_2` remain unchanged. This is because `species_2` refers to a new list that is a copy of `species_1`.

4.5.2 2.5.2. Tuple is Immutable

In the other hand, tuple is immutable, so we cannot change the contents of a tuple once it is created. For example, we cannot change the second item `Furrytail` in the tuple `species_tup_1`:

Input

```
# Immutable
species_tup_1 = ('Tub', 'Furrytail', 'Cat', 'Barkalot')
species_tup_1[1] = 'Furrytail 2'
print(species_tup_1)
```

Output

```
TypeError: 'tuple' object does not support item assignment
```

Here we get an error message `TypeError: 'tuple' object does not support item assignment` on changing the second item in the `species_tup_1` tuple.

4.6 2.6. Dictionaries

Dictionaries are similar to lists, but instead of using an index to access an item, we use a key. Dictionaries are unordered, and we cannot sort them. Dictionaries are mutable, so we can change the contents of a dictionary once it is created.

4.6.1 2.6.1. Creating a Dictionary

Let's create a dictionary that stores one of our old friend `'Tub'`, its `age`, and favoriate `habitat`:

Input

```
species = {'species': 'Tub', 'age': 5, 'habitat': ['bathroom', 'kitchen']}
print(species)
```

Output

```
{'species': 'Tub', 'age': 5, 'habitat': ['bathroom', 'kitchen']}
```

4.6.2 2.6.2. Accessing Items in a Dictionary

We can access the items in a dictionary by using the key `'species'`, `'age'`, and `'habitat'`:

Input

```
print(species['species'])
print(species['age'])
print(species['habitat'])
```

Output

```
Tub
5
['bathroom', 'kitchen']
```

Of course, we can use number as the key, but it is not recommended:

Input

```
species = {1: 'Tub', 'age': 5, 'habitat': ['bathroom', 'kitchen']}
print(species[1])
```

Output

```
Tub
```

What if we accidently acces a key that not exist in the dictionary?

Input

```
# Access a key that not exist
species = {'species': 'Tub', 'age': 5, 'habitat': ['bathroom', 'kitchen']}
print(species['weight'])
```

Output

```
KeyError: 'weight'
```

We will get an error message `KeyError: 'weight'`.

But practically this is not ideal, sometimes we just want to check if the key is in the dictionary or not without showing error, but return a `flag`.

We can use `get()` method to do this:

Input

```
# Get method
species = {'species': 'Tub', 'age': 5, 'habitat': ['bathroom', 'kitchen']}
print(species.get('age'))
print(species.get('weight'))
```

Output

```
5
None
```

We can see that when we use `get()` method, if the key is in the dictionary, it will return the value of the key, e.g. `5`, but if the key is not in the dictionary, it will return `None`.

We can also specify a default value to return if the key is not in the dictionary instead of `None`:

Input

```
species = {'species': 'Tub', 'age': 5, 'habitat': ['bathroom', 'kitchen']}
print(species.get('weight', 'Not there'))
```

Output

```
Not there
```

4.6.3 2.6.3. Changing Items in a Dictionary

We can update the value of a key, e.g. `'weight'`:

Input

```
species = {'species': 'Tub', 'age': 5, 'habitat': ['bathroom', 'kitchen']}
species['weight'] = 2000
print(species)
```

Output

```
{'species': 'Tub', 'age': 5, 'habitat': ['bathroom', 'kitchen'], 'weight': 2000}
```

We can also use `update()` to update the values from keys in a dictionary:

Input

```
species = {'species': 'Tub', 'age': 5, 'habitat': ['bathroom', 'kitchen']}
species.update({'weight': 1000, 'name': 'Fluffy', 'age': 6})
print(species)
```

Output

```
{'species': 'Tub', 'age': 6, 'habitat': ['bathroom', 'kitchen'], 'weight': 1000, 'name': 'Fluffy'}
```

While if we want to delete the key `'age'` and its value from the dictionary, we can use `del`:

Input

```
species = {'species': 'Tub', 'age': 5, 'habitat': ['bathroom', 'kitchen']}
del species['age']
print(species)
```

Output

```
{'species': 'Tub', 'habitat': ['bathroom', 'kitchen']}
```

Or use `pop()`:

Input

```
species = {'species': 'Tub', 'age': 5, 'habitat': ['bathroom', 'kitchen']}
species.pop('age')
print(species)
```

Output

```
{'species': 'Tub', 'habitat': ['bathroom', 'kitchen']}
```

We can also assign the popped value to a variable:

Input

```
age_popped = species.pop('age')
print(species)
print(age_popped)
```

Output

```
{'species': 'Tub', 'habitat': ['bathroom', 'kitchen']}
5
```

This can be useful if we want to use the popped value later.

As we mentioned in the previous sections, `len()` can also be used to check how many keys in a dictionary:

Input

```
species = {'species': 'Tub', 'age': 5, 'habitat': ['bathroom', 'kitchen']}
print(len(species))
```

Output

```
3
```

4.6.4 2.6.4. Accessing Keys and Values

We can access all the keys from a dictionary using `keys()`:

Input

```
species = {'species': 'Tub', 'age': 5, 'habitat': ['bathroom', 'kitchen']}
print(species.keys())
```

Output

```
dict_keys(['species', 'age', 'habitat'])
```

We can also access all the values from a dictionary using `values()`:

Input

```
print(species.values())
```

Output

```
dict_values(['Tub', 5, ['bathroom', 'kitchen']])
```

If we want to access the key-value pairs, we can use `items()`:

Input

```
print(species.items())
```

Output

```
dict_items([('species', 'Tub'), ('age', 5), ('habitat', ['bathroom', 'kitchen'])])
```

This is convenient if we want to loop through the key-value pairs.

4.7 2.7. Sets

Sets are unordered collections of unique elements and are useful when we want to store a collection of items that are not in any particular order and we don't want to store duplicate items.

4.7.1 2.7.1. Creating a Set

Create an empty set is similar to creating an empty list, you can use `set()`. Although the brackets `{}` are also used to create a set, it is actually creating an empty dictionary. To create an empty set, you need to use `set()`:

Input

```
empty_dict = {} # This is to create an empty dictionary
empty_set = set() # This is to create an empty set
print(type(empty_dict))
print(empty_set)
print(type(empty_set))
```

Output

```
<class 'dict'>
set()
<class 'set'>
```

Create a set based on the same strings as the list `species`:

Input

```
species = {'Tub', 'Furrytail', 'Cat', 'Barkalot'}
print(type(species))
print(species)
```

Output

```
<class 'set'>
{'Tub', 'Cat', 'Barkalot', 'Furrytail'}
```

`set` Doesn't Care About Order

Notice that the order of the elements in the set is different from the order of the elements in the list `species`. This is because sets are unordered collections of unique elements. If you run it multiple times, the order will be different as sets don't care about the order of the elements.

4.7.2 2.7.2. Set Methods

If we create a set with duplicate elements, the set will only keep one copy of the element:

Input

```
# Sets throw away the duplicate elements.
species = {'Tub', 'Furrytail', 'Cat', 'Barkalot', 'Tub'}
print(species)
```

Output

```
{'Tub', 'Cat', 'Barkalot', 'Furrytail'}
```

Sets are optimized for checking whether an element is contained in the set.

Input

```
species = {'Tub', 'Furrytail', 'Cat', 'Barkalot'}
print('Tub' in species)
print('Tub' not in species)
```

Output

```
True
False
```

We can check if two sets of species in common using `intersection()`:

Input

```
# What these species have in common
species_1 = {'Tub', 'Furrytail', 'Cat', 'Barkalot'}
species_2 = {'Tub', 'Furrytail', 'Bumblefluff ', 'Whiskerfloof'}
print(species_1.intersection(species_2))
```

Output

```
{'Tub', 'Furrytail'}
```

We can also check if two sets of species not in common using `difference()`:

Input

```
# What these species don't have in common
species_1 = {'Tub', 'Furrytail', 'Cat', 'Barkalot'}
species_2 = {'Tub', 'Furrytail', 'Bumblefluff ', 'Whiskerfloof'}
print(species_1.difference(species_2))
```

Output

```
{'Cat', 'Barkalot'}
```

Finally, we can also union both of the sets using `union()`:

Input

```
# What these species have in common and what they don't have in common
print(species_1.union(species_2))
```

Output

```
{'Tub', 'Cat', 'Barkalot', 'Furrytail', 'Bumblefluff ', 'Whiskerfloof'}
```

4.8 2.8. Advanced Operations for Dicts, Lists, Numbers, and Dates

View Source Code | View Slides

5. 05. PyTorch Going Modular

This section answers the question, "how do I turn my notebook code into Python scripts?"

To do so, we're going to turn the most useful code cells in notebook 04. PyTorch Custom Datasets into a series of Python scripts saved to a directory called `going_modular`.

5.1 What is going modular?

Going modular involves turning notebook code (from a Jupyter Notebook or Google Colab notebook) into a series of different Python scripts that offer similar functionality.

For example, we could turn our notebook code from a series of cells into the following Python files:

- `data_setup.py` - a file to prepare and download data if needed.
- `engine.py` - a file containing various training functions.
- `model_builder.py` or `model.py` - a file to create a PyTorch model.
- `train.py` - a file to leverage all other files and train a target PyTorch model.
- `utils.py` - a file dedicated to helpful utility functions.

Note: The naming and layout of the above files will depend on your use case and code requirements. Python scripts are as general as individual notebook cells, meaning, you could create one for almost any kind of functionality.

5.2 Why would you want to go modular?

Notebooks are fantastic for iteratively exploring and running experiments quickly.

However, for larger scale projects you may find Python scripts more reproducible and easier to run.

Though this is a debated topic, as companies like Netflix have shown how they use notebooks for production code.

Production code is code that runs to offer a service to someone or something.

For example, if you have an app running online that other people can access and use, the code running that app is considered **production code**.

And libraries like fast.ai's `nb-dev` (short for notebook development) enable you to write whole Python libraries (including documentation) with Jupyter Notebooks.

5.2.1 Pros and cons of notebooks vs Python scripts

There's arguments for both sides.

But this list sums up a few of the main topics.

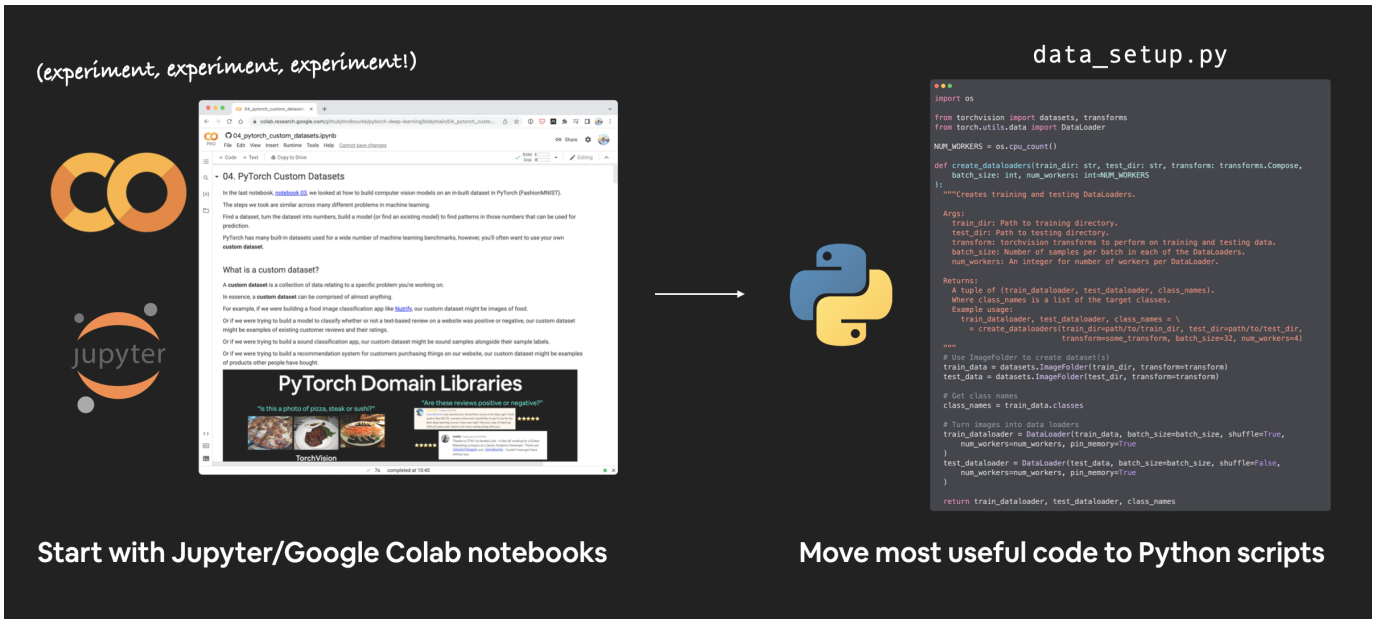
	Pros	Cons
Notebooks	Easy to experiment/get started	Versioning can be hard
	Easy to share (e.g. a link to a Google Colab notebook)	Hard to use only specific parts
	Very visual	Text and graphics can get in the way of code

	Pros	Cons
Python scripts	Can package code together (saves rewriting similar code across different notebooks)	Experimenting isn't as visual (usually have to run the whole script rather than one cell)
	Can use git for versioning	
	Many open source projects use scripts	
	Larger projects can be run on cloud vendors (not as much support for notebooks)	

5.2.2 My workflow

I usually start machine learning projects in Jupyter/Google Colab notebooks for quick experimentation and visualization.

Then when I've got something working, I move the most useful pieces of code to Python scripts.



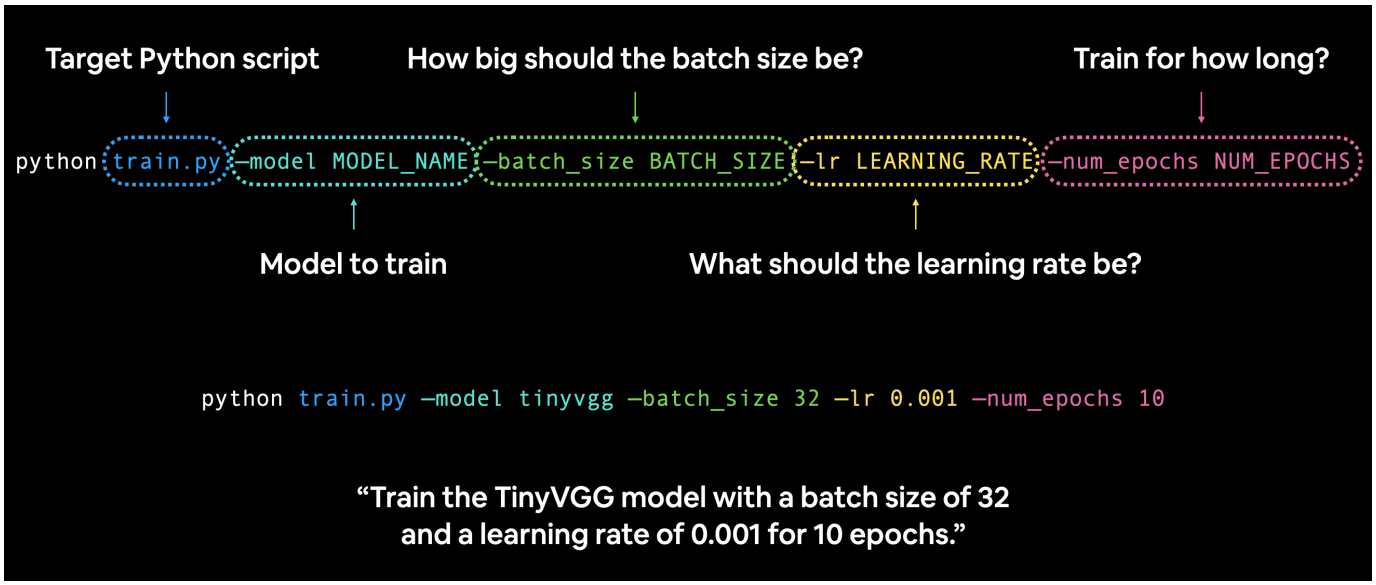
There are many possible workflows for writing machine learning code. Some prefer to start with scripts, others (like me) prefer to start with notebooks and go to scripts later on.

5.2.3 PyTorch in the wild

In your travels, you'll see many code repositories for PyTorch-based ML projects have instructions on how to run the PyTorch code in the form of Python scripts.

For example, you might be instructed to run code like the following in a terminal/command line to train a model:

```
python train.py --model MODEL_NAME --batch_size BATCH_SIZE --lr LEARNING_RATE --num_epochs NUM_EPOCHS
```



Running a PyTorch `train.py` script on the command line with various hyperparameter settings.

In this case, `train.py` is the target Python script, it'll likely contain functions to train a PyTorch model.

And `--model`, `--batch_size`, `--lr` and `--num_epochs` are known as argument flags.

You can set these to whatever values you like and if they're compatible with `train.py`, they'll work, if not, they'll error.

For example, let's say we wanted to train our TinyVGG model from notebook 04 for 10 epochs with a batch size of 32 and a learning rate of 0.001:

```
python train.py --model tinyvgg --batch_size 32 --lr 0.001 --num_epochs 10
```

You could setup any number of these argument flags in your `train.py` script to suit your needs.

The PyTorch blog post for training state-of-the-art computer vision models uses this style.

> Using our standard training reference script, we can train a ResNet50 using the following command:

```
torchrun --nproc_per_node=8 train.py --model resnet50 --batch-size 128 --lr 0.5 \
--lr-scheduler cosineannealinglr --lr-warmup-epochs 5 --lr-warmup-method linear \
--auto-augment ta_wide --epochs 600 --random-erase 0.1 --weight-decay 0.00002 \
--norm-weight-decay 0.0 --label-smoothing 0.1 --mixup-alpha 0.2 --cutmix-alpha 1.0 \
--train-crop-size 176 --model-ema --val-resize-size 232 --ra-sampler --ra-reps 4
```

PyTorch command line training script recipe for training state-of-the-art computer vision models with 8 GPUs. Source: PyTorch blog.

5.3 What we're going to cover

The main concept of this section is: **turn useful notebook code cells into reusable Python files.**

Doing this will save us writing the same code over and over again.

There are two notebooks for this section:

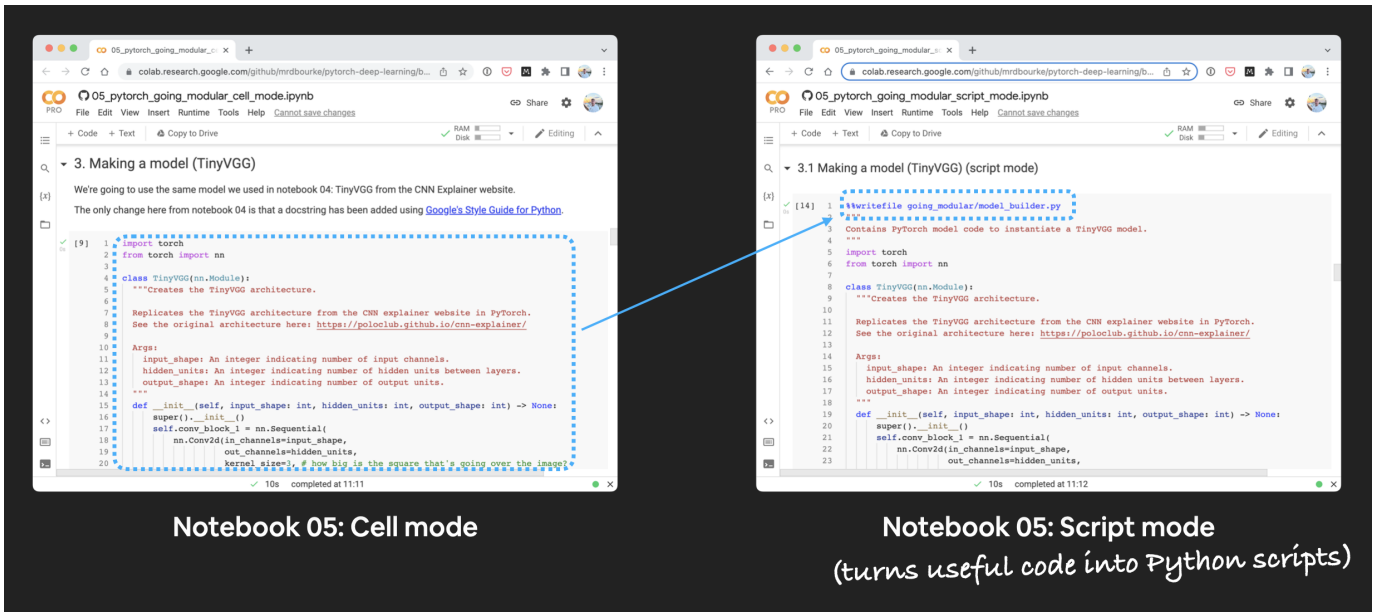
1. **05. Going Modular: Part 1 (cell mode)** - this notebook is run as a traditional Jupyter Notebook/Google Colab notebook and is a condensed version of notebook 04.
2. **05. Going Modular: Part 2 (script mode)** - this notebook is the same as number 1 but with added functionality to turn each of the major sections into Python scripts, such as, `data_setup.py` and `train.py`.

The text in this document focuses on the code cells 05. Going Modular: Part 2 (script mode), the ones with `%%writefile ...` at the top.

5.3.1 Why two parts?

Because sometimes the best way to learn something is to see how it *differs* from something else.

If you run each notebook side-by-side you'll see how they differ and that's where the key learnings are.



Running the two notebooks for section 05 side-by-side. You'll notice that the script mode notebook has extra code cells to turn code from the cell mode notebook into Python scripts.

5.3.2 What we're working towards

By the end of this section we want to have two things:

1. The ability to train the model we built in notebook 04 (Food Vision Mini) with one line of code on the command line: `python train.py`.
2. A directory structure of reusable Python scripts, such as:

```
going_modular/
├── going_modular/
│   ├── data_setup.py
│   ├── engine.py
│   ├── model_builder.py
│   ├── train.py
│   └── utils.py
├── models/
│   ├── 05_going_modular_cell_mode_tinyvgg_model.pth
│   └── 05_going_modular_script_mode_tinyvgg_model.pth
└── data/
    └── pizza_steak_sushi/
        ├── train/
        │   ├── pizza/
        │   │   ├── image01.jpeg
        │   │   └── ...
        │   ├── steak/
        │   └── sushi/
        └── test/
            ├── pizza/
            ├── steak/
            └── sushi/
```

5.3.3 Things to note

- **Docstrings** - Writing reproducible and understandable code is important. And with this in mind, each of the functions/classes we'll be putting into scripts has been created with Google's Python docstring style in mind.

- **Imports at the top of scripts** - Since all of the Python scripts we're going to create could be considered a small program on their own, all of the scripts require their input modules be imported at the start of the script for example:

```
# Import modules required for train.py
import os
import torch
import data_setup, engine, model_builder, utils

from torchvision import transforms
```

5.4 Where can you get help?

All of the materials for this course are available on GitHub.

If you run into trouble, you can ask a question on the course GitHub Discussions page.

And of course, there's the PyTorch documentation and PyTorch developer forums, a very helpful place for all things PyTorch.

5.5 0. Cell mode vs. script mode

A cell mode notebook such as 05. Going Modular Part 1 (cell mode) is a notebook run normally, each cell in the notebook is either code or markdown.

A script mode notebook such as 05. Going Modular Part 2 (script mode) is very similar to a cell mode notebook, however, many of the code cells may be turned into Python scripts.

Note: You don't *need* to create Python scripts via a notebook, you can create them directly through an IDE (integrated developer environment) such as VS Code. Having the script mode notebook as part of this section is just to demonstrate one way of going from notebooks to Python scripts.

5.6 1. Get data

Getting the data in each of the 05 notebooks happens the same as in notebook 04.

A call is made to GitHub via Python's `requests` module to download a `.zip` file and unzip it.

```
import os
import requests
import zipfile
from pathlib import Path

# Setup path to data folder
data_path = Path("data/")
image_path = data_path / "pizza_steak_sushi"

# If the image folder doesn't exist, download it and prepare it...
if image_path.is_dir():
    print(f"{image_path} directory exists.")
else:
    print(f"Did not find {image_path} directory, creating one...")
    image_path.mkdir(parents=True, exist_ok=True)

# Download pizza, steak, sushi data
with open(data_path / "pizza_steak_sushi.zip", "wb") as f:
    request = requests.get("https://github.com/mrdbourke/pytorch-deep-learning/raw/main/data/pizza_steak_sushi.zip")
    print("Downloading pizza, steak, sushi data...")
    f.write(request.content)

# Unzip pizza, steak, sushi data
with zipfile.ZipFile(data_path / "pizza_steak_sushi.zip", "r") as zip_ref:
    print("Unzipping pizza, steak, sushi data...")
    zip_ref.extractall(image_path)

# Remove zip file
os.remove(data_path / "pizza_steak_sushi.zip")
```

This results in having a file called `data` that contains another directory called `pizza_steak_sushi` with images of pizza, steak and sushi in standard image classification format.

```
data/
└── pizza_steak_sushi/
    ├── train/
    │   ├── pizza/
    │   │   ├── train_image01.jpeg
```

```
    |   |   ├── test_image02.jpeg
    |   |   └── ...
    |   ├── steak/
    |   |   └── ...
    |   └── sushi/
    |       └── ...
    └── test/
        ├── pizza/
        |   ├── test_image01.jpeg
        |   └── test_image02.jpeg
        ├── steak/
        └── sushi/
```

5.7 2. Create Datasets and DataLoaders (`data_setup.py`)

Once we've got data, we can then turn it into PyTorch `Dataset`'s and `DataLoader`'s (one for training data and one for testing data).

We convert the useful `Dataset` and `DataLoader` creation code into a function called `create_dataloaders()`.

And we write it to file using the line `%%writefile going_modular/data_setup.py`.

data_setup.py

```
%%writefile going_modular/data_setup.py
"""
Contains functionality for creating PyTorch DataLoaders for
image classification data.
"""
import os

from torchvision import datasets, transforms
from torch.utils.data import DataLoader

NUM_WORKERS = os.cpu_count()

def create_dataloaders(
    train_dir: str,
    test_dir: str,
    transform: transforms.Compose,
    batch_size: int,
    num_workers: int=NUM_WORKERS
):
  """Creates training and testing DataLoaders.

  Takes in a training directory and testing directory path and turns
  them into PyTorch Datasets and then into PyTorch DataLoaders.

  Args:
    train_dir: Path to training directory.
    test_dir: Path to testing directory.
    transform: torchvision transforms to perform on training and testing data.
    batch_size: Number of samples per batch in each of the DataLoaders.
    num_workers: An integer for number of workers per DataLoader.

  Returns:
    A tuple of (train_dataloader, test_dataloader, class_names).
    Where class_names is a list of the target classes.
    Example usage:
      train_dataloader, test_dataloader, class_names = \
        = create_dataloaders(train_dir=path/to/train_dir,
                             test_dir=path/to/test_dir,
                             transform=some_transform,
                             batch_size=32,
                             num_workers=4)
  """
  # Use ImageFolder to create dataset(s)
  train_data = datasets.ImageFolder(train_dir, transform=transform)
  test_data = datasets.ImageFolder(test_dir, transform=transform)

  # Get class names
  class_names = train_data.classes

  # Turn images into data loaders
  train_dataloader = DataLoader(
      train_data,
      batch_size=batch_size,
      shuffle=True,
      num_workers=num_workers,
      pin_memory=True,
  )
  test_dataloader = DataLoader(
      test_data,
      batch_size=batch_size,
      shuffle=True,
      num_workers=num_workers,
      pin_memory=True,
  )
```

```
  return train_dataloader, test_dataloader, class_names
```

If we'd like to make `DataLoader`'s we can now use the function within `data_setup.py` like so:

```
# Import data_setup.py
from going_modular import data_setup

# Create train/test dataloader and get class names as a list
train_dataloader, test_dataloader, class_names = data_setup.create_dataloaders(...)
```

5.8 3. Making a model (`model_builder.py`)

Over the past few notebooks (notebook 03 and notebook 04), we've built the TinyVGG model a few times.

So it makes sense to put the model into its file so we can reuse it again and again.

Let's put our `TinyVGG()` model class into a script with the line `%%writefile going_modular/model_builder.py`:

model_builder.py

```
%%writefile going_modular/model_builder.py
"""
Contains PyTorch model code to instantiate a TinyVGG model.
"""
import torch
from torch import nn

class TinyVGG(nn.Module):
  """Creates the TinyVGG architecture.

  Replicates the TinyVGG architecture from the CNN explainer website in PyTorch.
  See the original architecture here: https://poloclub.github.io/cnn-explainer/

  Args:
    input_shape: An integer indicating number of input channels.
    hidden_units: An integer indicating number of hidden units between layers.
    output_shape: An integer indicating number of output units.
  """
  def __init__(self, input_shape: int, hidden_units: int, output_shape: int) -> None:
      super().__init__()
      self.conv_block_1 = nn.Sequential(
          nn.Conv2d(in_channels=input_shape,
                    out_channels=hidden_units,
                    kernel_size=3,
                    stride=1,
                    padding=0),
          nn.ReLU(),
          nn.Conv2d(in_channels=hidden_units,
                    out_channels=hidden_units,
                    kernel_size=3,
                    stride=1,
                    padding=0),
          nn.ReLU(),
          nn.MaxPool2d(kernel_size=2,
                       stride=2)
      )
      self.conv_block_2 = nn.Sequential(
          nn.Conv2d(hidden_units, hidden_units, kernel_size=3, padding=0),
          nn.ReLU(),
          nn.Conv2d(hidden_units, hidden_units, kernel_size=3, padding=0),
          nn.ReLU(),
          nn.MaxPool2d(2)
      )
      self.classifier = nn.Sequential(
          nn.Flatten(),
          # Where did this in_features shape come from?
          # It's because each layer of our network compresses and changes the shape of our inputs data.
          nn.Linear(in_features=hidden_units*13*13,
                    out_features=output_shape)
      )

  def forward(self, x: torch.Tensor):
      x = self.conv_block_1(x)
      x = self.conv_block_2(x)
      x = self.classifier(x)
      return x
      # return self.classifier(self.conv_block_2(self.conv_block_1(x))) # <- leverage the benefits of operator fusion
```

Now instead of coding the TinyVGG model from scratch every time, we can import it using:

```
import torch
# Import model_builder.py
from going_modular import model_builder
```

```
device = "cuda" if torch.cuda.is_available() else "cpu"

# Instantiate an instance of the model from the "model_builder.py" script
torch.manual_seed(42)
model = model_builder.TinyVGG(input_shape=3,
                              hidden_units=10,
                              output_shape=len(class_names)).to(device)
```

5.9 4. Creating `train_step()` and `test_step()` functions and `train()` to combine them

We wrote several training functions in notebook 04:

1. `train_step()` - takes in a model, a `DataLoader`, a loss function and an optimizer and trains the model on the `DataLoader`.
2. `test_step()` - takes in a model, a `DataLoader` and a loss function and evaluates the model on the `DataLoader`.
3. `train()` - performs 1. and 2. together for a given number of epochs and returns a results dictionary.

Since these will be the *engine* of our model training, we can put them all into a Python script called `engine.py` with the line `%%writefile going_modular/engine.py`:

engine.py

```
%%writefile going_modular/engine.py
"""
Contains functions for training and testing a PyTorch model.
"""
import torch

from tqdm.auto import tqdm
from typing import Dict, List, Tuple

def train_step(model: torch.nn.Module,
               dataloader: torch.utils.data.DataLoader,
               loss_fn: torch.nn.Module,
               optimizer: torch.optim.Optimizer,
               device: torch.device) -> Tuple[float, float]:
  """Trains a PyTorch model for a single epoch.

  Turns a target PyTorch model to training mode and then
  runs through all of the required training steps (forward
  pass, loss calculation, optimizer step).

  Args:
    model: A PyTorch model to be trained.
    dataloader: A DataLoader instance for the model to be trained on.
    loss_fn: A PyTorch loss function to minimize.
    optimizer: A PyTorch optimizer to help minimize the loss function.
    device: A target device to compute on (e.g. "cuda" or "cpu").

  Returns:
    A tuple of training loss and training accuracy metrics.
    In the form (train_loss, train_accuracy). For example:

    (0.1112, 0.8743)
  """
  # Put model in train mode
  model.train()

  # Setup train loss and train accuracy values
  train_loss, train_acc = 0, 0

  # Loop through data loader data batches
  for batch, (X, y) in enumerate(dataloader):
      # Send data to target device
      X, y = X.to(device), y.to(device)

      # 1. Forward pass
      y_pred = model(X)

      # 2. Calculate  and accumulate loss
      loss = loss_fn(y_pred, y)
      train_loss += loss.item()

      # 3. Optimizer zero grad
      optimizer.zero_grad()

      # 4. Loss backward
      loss.backward()

      # 5. Optimizer step
      optimizer.step()

      # Calculate and accumulate accuracy metric across all batches
      y_pred_class = torch.argmax(torch.softmax(y_pred, dim=1), dim=1)
      train_acc += (y_pred_class == y).sum().item()/len(y_pred)
```

```
  # Adjust metrics to get average loss and accuracy per batch
  train_loss = train_loss / len(dataloader)
  train_acc = train_acc / len(dataloader)
  return train_loss, train_acc

def test_step(model: torch.nn.Module,
              dataloader: torch.utils.data.DataLoader,
              loss_fn: torch.nn.Module,
              device: torch.device) -> Tuple[float, float]:
  """Tests a PyTorch model for a single epoch.

  Turns a target PyTorch model to "eval" mode and then performs
  a forward pass on a testing dataset.

  Args:
    model: A PyTorch model to be tested.
    dataloader: A DataLoader instance for the model to be tested on.
    loss_fn: A PyTorch loss function to calculate loss on the test data.
    device: A target device to compute on (e.g. "cuda" or "cpu").

  Returns:
    A tuple of testing loss and testing accuracy metrics.
    In the form (test_loss, test_accuracy). For example:

    (0.0223, 0.8985)
  """
  # Put model in eval mode
  model.eval()

  # Setup test loss and test accuracy values
  test_loss, test_acc = 0, 0

  # Turn on inference context manager
  with torch.inference_mode():
      # Loop through DataLoader batches
      for batch, (X, y) in enumerate(dataloader):
          # Send data to target device
          X, y = X.to(device), y.to(device)

          # 1. Forward pass
          test_pred_logits = model(X)

          # 2. Calculate and accumulate loss
          loss = loss_fn(test_pred_logits, y)
          test_loss += loss.item()

          # Calculate and accumulate accuracy
          test_pred_labels = test_pred_logits.argmax(dim=1)
          test_acc += ((test_pred_labels == y).sum().item()/len(test_pred_labels))

  # Adjust metrics to get average loss and accuracy per batch
  test_loss = test_loss / len(dataloader)
  test_acc = test_acc / len(dataloader)
  return test_loss, test_acc

def train(model: torch.nn.Module,
          train_dataloader: torch.utils.data.DataLoader,
          test_dataloader: torch.utils.data.DataLoader,
          optimizer: torch.optim.Optimizer,
          loss_fn: torch.nn.Module,
          epochs: int,
          device: torch.device) -> Dict[str, List]:
  """Trains and tests a PyTorch model.

  Passes a target PyTorch models through train_step() and test_step()
  functions for a number of epochs, training and testing the model
  in the same epoch loop.

  Calculates, prints and stores evaluation metrics throughout.

  Args:
    model: A PyTorch model to be trained and tested.
    train_dataloader: A DataLoader instance for the model to be trained on.
    test_dataloader: A DataLoader instance for the model to be tested on.
    optimizer: A PyTorch optimizer to help minimize the loss function.
    loss_fn: A PyTorch loss function to calculate loss on both datasets.
    epochs: An integer indicating how many epochs to train for.
    device: A target device to compute on (e.g. "cuda" or "cpu").

  Returns:
    A dictionary of training and testing loss as well as training and
    testing accuracy metrics. Each metric has a value in a list for
    each epoch.
    In the form: {train_loss: [...],
                  train_acc: [...],
                  test_loss: [...],
                  test_acc: [...]}
    For example if training for epochs=2:
                 {train_loss: [2.0616, 1.0537],
                  train_acc: [0.3945, 0.3945],
                  test_loss: [1.2641, 1.5706],
                  test_acc: [0.3400, 0.2973]}
  """
```

```
    # Create empty results dictionary
    results = {"train_loss": [],
        "train_acc": [],
        "test_loss": [],
        "test_acc": []
    }

    # Loop through training and testing steps for a number of epochs
    for epoch in tqdm(range(epochs)):
        train_loss, train_acc = train_step(model=model,
                                           dataloader=train_dataloader,
                                           loss_fn=loss_fn,
                                           optimizer=optimizer,
                                           device=device)
        test_loss, test_acc = test_step(model=model,
            dataloader=test_dataloader,
            loss_fn=loss_fn,
            device=device)

        # Print out what's happening
        print(
            f"Epoch: {epoch+1} | "
            f"train_loss: {train_loss:.4f} | "
            f"train_acc: {train_acc:.4f} | "
            f"test_loss: {test_loss:.4f} | "
            f"test_acc: {test_acc:.4f}"
        )

        # Update results dictionary
        results["train_loss"].append(train_loss)
        results["train_acc"].append(train_acc)
        results["test_loss"].append(test_loss)
        results["test_acc"].append(test_acc)

    # Return the filled results at the end of the epochs
    return results
```

Now we've got the `engine.py` script, we can import functions from it via:

```
# Import engine.py
from going_modular import engine

# Use train() by calling it from engine.py
engine.train(...)
```

5.10 5. Creating a function to save the model (`utils.py`)

Often you'll want to save a model whilst it's training or after training.

Since we've written the code to save a model a few times now in previous notebooks, it makes sense to turn it into a function and save it to file.

It's common practice to store helper functions in a file called `utils.py` (short for utilities).

Let's save our `save_model()` function to a file called `utils.py` with the line `%%writefile going_modular/utils.py`:

utils.py

```
%%writefile going_modular/utils.py
"""
Contains various utility functions for PyTorch model training and saving.
"""
import torch
from pathlib import Path

def save_model(model: torch.nn.Module,
               target_dir: str,
               model_name: str):
  """Saves a PyTorch model to a target directory.

  Args:
    model: A target PyTorch model to save.
    target_dir: A directory for saving the model to.
    model_name: A filename for the saved model. Should include
      either ".pth" or ".pt" as the file extension.

  Example usage:
    save_model(model=model_0,
               target_dir="models",
               model_name="05_going_modular_tingvgg_model.pth")
  """
  # Create target directory
  target_dir_path = Path(target_dir)
  target_dir_path.mkdir(parents=True,
                        exist_ok=True)
```

```
  # Create model save path
  assert model_name.endswith(".pth") or model_name.endswith(".pt"), "model_name should end with '.pt' or '.pth'"
  model_save_path = target_dir_path / model_name

  # Save the model state_dict()
  print(f"[INFO] Saving model to: {model_save_path}")
  torch.save(obj=model.state_dict(),
             f=model_save_path)
```

Now if we wanted to use our `save_model()` function, instead of writing it all over again, we can import it and use it via:

```
# Import utils.py
from going_modular import utils

# Save a model to file
save_model(model=...
           target_dir=...,
           model_name=...)
```

5.11 6. Train, evaluate and save the model (`train.py`)

As previously discussed, you'll often come across PyTorch repositories that combine all of their functionality together in a `train.py` file.

This file is essentially saying "train the model using whatever data is available".

In our `train.py` file, we'll combine all of the functionality of the other Python scripts we've created and use it to train a model.

This way we can train a PyTorch model using a single line of code on the command line:

```
python train.py
```

To create `train.py` we'll go through the following steps:

1. Import the various dependencies, namely `torch`, `os`, `torchvision.transforms` and all of the scripts from the `going_modular` directory, `data_setup`, `engine`, `model_builder`, `utils`.
2. **Note:** Since `train.py` will be *inside* the `going_modular` directory, we can import the other modules via `import ...` rather than `from going_modular import ...`.
3. Setup various hyperparameters such as batch size, number of epochs, learning rate and number of hidden units (these could be set in the future via Python's `argparse`).
4. Setup the training and test directories.
5. Setup device-agnostic code.
6. Create the necessary data transforms.
7. Create the DataLoaders using `data_setup.py`.
8. Create the model using `model_builder.py`.
9. Setup the loss function and optimizer.
10. Train the model using `engine.py`.
11. Save the model using `utils.py`.

And we can create the file from a notebook cell using the line `%%writefile going_modular/train.py`:

train.py

```
%%writefile going_modular/train.py
"""
Trains a PyTorch image classification model using device-agnostic code.
"""

import os
import torch
import data_setup, engine, model_builder, utils

from torchvision import transforms

# Setup hyperparameters
NUM_EPOCHS = 5
BATCH_SIZE = 32
HIDDEN_UNITS = 10
```

```
LEARNING_RATE = 0.001

# Setup directories
train_dir = "data/pizza_steak_sushi/train"
test_dir = "data/pizza_steak_sushi/test"

# Setup target device
device = "cuda" if torch.cuda.is_available() else "cpu"

# Create transforms
data_transform = transforms.Compose([
  transforms.Resize((64, 64)),
  transforms.ToTensor()
])

# Create DataLoaders with help from data_setup.py
train_dataloader, test_dataloader, class_names = data_setup.create_dataloaders(
    train_dir=train_dir,
    test_dir=test_dir,
    transform=data_transform,
    batch_size=BATCH_SIZE
)

# Create model with help from model_builder.py
model = model_builder.TinyVGG(
    input_shape=3,
    hidden_units=HIDDEN_UNITS,
    output_shape=len(class_names)
).to(device)

# Set loss and optimizer
loss_fn = torch.nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(),
                             lr=LEARNING_RATE)

# Start training with help from engine.py
engine.train(model=model,
             train_dataloader=train_dataloader,
             test_dataloader=test_dataloader,
             loss_fn=loss_fn,
             optimizer=optimizer,
             epochs=NUM_EPOCHS,
             device=device)

# Save the model with help from utils.py
utils.save_model(model=model,
                 target_dir="models",
                 model_name="05_going_modular_script_mode_tinyvgg_model.pth")
```

Woohoo!

Now we can train a PyTorch model by running the following line on the command line:

```
python train.py
```

Doing this will leverage all of the other code scripts we've created.

And if we wanted to, we could adjust our `train.py` file to use argument flag inputs with Python's `argparse` module, this would allow us to provide different hyperparameter settings like previously discussed:

```
python train.py --model MODEL_NAME --batch_size BATCH_SIZE --lr LEARNING_RATE --num_epochs NUM_EPOCHS
```

5.12 Exercises

Resources:

- Exercise template notebook for 05
- Example solutions notebook for 05
- Live coding run through of solutions notebook for 05 on YouTube

Exercises:

1. Turn the code to get the data (from section 1. Get Data above) into a Python script, such as `get_data.py`.
   - When you run the script using `python get_data.py` it should check if the data already exists and skip downloading if it does.
   - If the data download is successful, you should be able to access the `pizza_steak_sushi` images from the `data` directory.
2. Use Python's `argparse` module to be able to send the `train.py` custom hyperparameter values for training procedures.
   - Add an argument for using a different:
   - Training/testing directory
   - Learning rate
   - Batch size
   - Number of epochs to train for
   - Number of hidden units in the TinyVGG model
   - Keep the default values for each of the above arguments as what they already are (as in notebook 05).
   - For example, you should be able to run something similar to the following line to train a TinyVGG model with a learning rate of 0.003 and a batch size of 64 for 20 epochs: `python train.py --learning_rate 0.003 --batch_size 64 --num_epochs 20`.
   - **Note:** Since `train.py` leverages the other scripts we created in section 05, such as, `model_builder.py`, `utils.py` and `engine.py`, you'll have to make sure they're available to use too. You can find these in the `going_modular` folder on the course GitHub.
3. Create a script to predict (such as `predict.py`) on a target image given a file path with a saved model.
   - For example, you should be able to run the command `python predict.py some_image.jpeg` and have a trained PyTorch model predict on the image and return its prediction.
   - To see example prediction code, check out the predicting on a custom image section in notebook 04.
   - You may also have to write code to load in a trained model.

5.13 Extra-curriculum

- To learn more about structuring a Python project, check out Real Python's guide on Python Application Layouts.
- For ideas on styling your PyTorch code, check out the PyTorch style guide by Igor Susmelj (much of styling in this chapter is based off this guide + various similar PyTorch repositories).
- For an example `train.py` script and various other PyTorch scripts written by the PyTorch team to train state-of-the-art image classification models, check out their `classification` repository on GitHub.