Python Programming: Takeaways

Syntax

- Displaying the output of a computer program:

```
print(1 + 2)
print(5 * 10)
```

- Ignoring certain lines of code by using code comments:

```
# print(1 + 2)
print(5 * 10)
# This program will only print 50
```

- Performing arithmetical operations:

```
1 + 2
4 - 5
30 * 1
20 / 3
4 ** 3
(4 * 18) ** 2 / 10
```

Concepts

- When we give a computer a set of instructions, we say that we're **programming** it. To program a computer, we need to write the instructions in a special language, which we call a **programming language**.
- Python has **syntax** rules, and each line of instruction has to comply with these rules. For example, `print(23 + 7) print(10 - 6) print(12 + 38)` doesn't comply with Python's syntax rules, and it causes a **syntax error**.
- We call the instructions we send to the computer **code**. We call each line of instruction a **line of code**.
- When we write code, we *program* the computer to do something. For this reason, we also call the code we write a **computer program** (or a **program**).
- The code we write serves as **input** to the computer. We call the result of executing the code **output**.
- We call the sequence of characters that follows the `#` symbol a **code comment**. We can use code comments to prevent the computer from executing a line of code or to add information about the code we write.

Programming Python Variables: Takeaways

Syntax

- Storing values to variables:

```
twenty = 20
result = 43 + 2**5
currency = 'USD'
```

- Updating the value stored in a variable:

```
x = 30
x += 10 # this is the same as x = x + 10
```

- Syntax shortcuts:

```
x += 2  # Addition
x -= 2  # Subtraction
x *= 2  # Multiplication
x /= 2  # Division
x **= 2 # Exponentiation
```

Concepts

- We can store values in the computer memory. We call each storage location in the computer's memory a **variable**.
- There are two syntax rules we need to follow when we're naming variables:
    - We must use only letters, numbers, or underscores (we can't use apostrophes, hyphens, spaces, etc.).
    - Variable names can't begin with a number.
- Whenever the syntax is correct, but the computer still returns an error, we call this a **runtime error**.
- In Python, the `=` operator tells us that the value on the right is **assigned** to the variable on the left. It doesn't tell us anything about equality. We call `=` an **assignment operator**, and we read code like `x = 5` as "five is assigned to x" or "x is assigned five," not "x equals five."
- In computer programming, we classify values into different **types** — or **data types**. A value's type offers the computer the information necessary to process that value. Depending on the type, the computer will know how to store a value in memory, or which operations it can perform on a value.

Python Data Types: Integers, Floats, Strings: Takeaways

Syntax

- Rounding a number:

```
round(4.99) # the output will be 5
```

- Using quotation marks to create a string:

```
app_name = "Clash of Clans"
app_rating = '3.5'
```

- Concatenating two or more strings:

```
print('a' + 'b') # prints 'ab'
print('a' + 'b' + 'c') # prints 'abc'
```

- Converting between types of variables:

```
int('4')
str(4)
float('4.3')
str(4.3)
```

- Finding the type of a value:

```
type(4)
type('4')
```

Concepts

- In this lesson, we learned about three data types: integers, floats, and strings.
- We call the process of linking two or more strings together **concatenation**.

Resources

- More on Strings in Python.

Python Lists: Takeaways

Syntax

- Creating a list of data points:

```
row_1 = ['Facebook', 0.0, 'USD', 2974676, 3.5]
row_2 = ['Instagram', 0.0, 'USD', 2161558, 4.5]
```

- Creating a list of lists:

```
data = [row_1, row_2]
```

- Retrieving an element of a list:

```
first_row = data[0]
first_element_in_first_row = first_row[0]
first_element_in_first_row = data[0][0]
last_element_in_first_row = first_row[-1]
last_element_in_first_row = data[0][-1]
```

- Retrieving multiple list elements and creating a new list:

```
row_1 = ['Facebook', 0.0, 'USD', 2974676, 3.5]
rating_data_only = [row_1[3], row_1[4]]
```

- Performing list slicing:

```
python
row_1 = ['Facebook', 0.0, 'USD', 2974676, 3.5]
second_to_fourth_element = row_1[1:4]
```

Concepts

- A **data point** is a value that offers us some information.
- A set of data points make up a **dataset**. A table is an example of a dataset.
- **Lists** are data types that can store datasets.

Resources

- Python Lists
- More on CSV files

Python For Loops: Takeaways

by Dataquest Labs, Inc. - All rights reserved © 2022

Syntax

- Repeating a process using a for loop:

```
row_1 = ['Facebook', 0.0, 'USD', 2974676, 3.5]
for element in row_1:
    print(element)
```

- Appending values to a list:

```
a_list = [1, 2]
a_list.append(3)
```

- Opening a dataset file and using it to create a list of lists:

```
opened_file = open('AppleStore.csv')
from csv import reader
read_file = reader(opened_file)
apps_data = list(read_file)
```

- Converting a string to a float:

```
rating_sum = 0
for row in apps_data[1:]:
    rating = float(row[7])
    rating_sum = rating_sum + rating
```

Concepts

- We can automate repetitive processes using **for loops**.
- We always start a for loop with `for` (like in `for element in app_ratings:`).
- The indented code in the **body** gets executed the same number of times as elements in the **iterable variable**. If the iterable variable is a list containing three elements, the indented code in the body gets executed three times. We call each code execution an **iteration**, so there will be three iterations for a list that has three elements. For each iteration, the **iteration variable** will take a different value.

Resources

- Python For Loops
- A list of keywords in Python — `for` and `in` are examples of keywords (we used `for` and `in` to write for loops)

Python If, Else, Elif Statements: Takeaways

Syntax

- Append values with each iteration of a for loop:

```
apps_names = []
for row in apps_data[1:]:
    name = row[1]
    apps_names.append(name)
print(apps_names[:5])
```

- Use an if statement to control your code:

```
if True:
    print(1)
if 1 == 1:
    print(2)
    print(3)
```

- Return boolean values:

```
price = 0
print(price == 0)
print(price == 2)
```

- Execute code only when True follows if:

```
if True:
    print('First Output')
if False:
    print('Second Output')
if True:
    print('Third Output')
```

- Using the == and != operators with strings or lists:

```
print('Games' == 'Music')
print('Games' != 'Music')
print([1,2,3] == [1,2,3])
print([1,2,3] == [1,2,3,4])
```

Concepts

- We can use an `if` **statement** to implement a condition in our code.
- The `if` statement starts with `if`, it continues with a condition such as `price == 0.0`, and it ends with `:`.
- We use the `==` operator to check whether something is **equal** to something else. Don't confuse `==` with `=` (`=` is a variable assignment operator in Python; we use it to assign values to variables — it doesn't tell us anything about equality).

- We indent operations within the body of an `if` statement, such as `list.append(value)` or `print(value)`, four spaces to the right relative to the `if` statement.
- We call `True` and `False` **Boolean values** or **Booleans** — their data type is `bool` ("bool" is an abbreviation for "Boolean").
- Boolean values (`True` and `False`) are necessary parts of any `if` statement. One of the following must always follow `if` : (1) a Boolean value or (2) an expression that evaluates to a Boolean value.
- Indented code *only* executes when `True` follows `if` .
- We can use the `==` and `!=` operators with strings or lists.

Resources

- If Statements in Python

Python If, Else, Elif Statements: Multiple Conditions: Takeaways

Syntax

- Combining multiple conditions:

```
if 3 > 1 and 'data' == 'data':
    print('Both conditions are true!')
if 10 < 20 or 4 <= 5:
    print('At least one condition is true.')
```

- Building more complex if statements:

```
if (20 > 3 and 2 != 1) or 'Games' == 'Games':
    print('At least one condition is true.')
```

- Using the else clause:

```
if False:
    print(1)
else:
    print('The condition above was false.')
```

- Using the elif clause:

```
if False:
    print(1)
elif 30 > 5:
    print('The condition above was false.')
```

Concepts

- We can use an `if` **statement** to implement a condition in our code.
- An `elif` clause executes if the preceding `if` statement (or the other preceding `elif` clauses) resolves to `False` *and* the condition specified after the `elif` keyword evaluates to `True`.
- `True` and `False` are **Boolean values**.
- Python evaluates any combination of Booleans to a single Boolean value.
- `and` and `or` are **logical operators**. They unite two or more Booleans.
- As a general rule, when we combine Booleans using `and`, the resulting Boolean is `True` only if all the Booleans are `True`. If any of the Booleans are `False`, then the resulting Boolean will be `False`.
- We can compare a value `A` to value `B` to determine the following:
    - `A` is **equal** to `B` and vice versa (`B` is equal to `A`) — `==`
    - `A` is **not equal** to `B` and vice versa — `!=`
    - `A` is **greater** than `B` or vice versa — `>`

- `A` is **greater than or equal to** `B` or vice versa — `>=`
- `A` is **less** than `B` or vice versa — `<`
- `A` is **less than or equal to** `B` or vice versa — `<=`

Resources

- If and elif Statements in Python

Python Dictionaries: Takeaways

Syntax

- Create a dictionary:

```
# First way
dictionary = {'key_1': 1, 'key_2': 2}
# Second way
dictionary = {}
dictionary['key_1'] = 1
dictionary['key_2'] = 2
```

- Retrieve individual dictionary values:

```
dictionary = {'key_1': 100, 'key_2': 200}
dictionary['key_1']  # Outputs 100
dictionary['key_2']  # Outputs 200
```

Concepts

- We call the index of a dictionary value a **key**. In `'4+': 4433`, the dictionary key is `'4+'`, and the dictionary value is `4433`. As a whole, `'4+': 4433` is a **key-value pair**.
- Dictionary values can be any data type: strings, integers, floats, Booleans, lists, and even dictionaries. Dictionary keys can be almost any data type, except lists and dictionaries. If we use lists or dictionaries as dictionary keys, we'll get an error.

Resources

- Dictionaries in Python

Python Dictionaries and Frequency Tables: Takeaways

Syntax

- Check if a certain value exists in the dictionary as a key:

```
dictionary = {'key_1': 100, 'key_2': 200}
'key_1' in dictionary  # Outputs True
'key_5' in dictionary  # Outputs False
100 in dictionary  # Outputs False
```

- Use the in operator to check for dictionary membership:

```
content_ratings = {'4+': 4433, '9+': 987, '12+': 1155, '17+': 622}
print('12+' in content_ratings)
```

- Update dictionary values:

```
dictionary = {'key_1': 100, 'key_2': 200}
dictionary['key_1'] += 600  # This will change the value to 700
```

- Create a frequency table for the unique values in a column of a dataset:

```
frequency_table = {}
for row in a_data_set:
    a_data_point = row[5]
    if a_data_point in frequency_table:
        frequency_table[a_data_point] += 1
    else:
        frequency_table[a_data_point] = 1
```

- Loop over a dictionary:

```
content_ratings = {'4+': 4433, '9+': 987, '12+': 1155, '17+': 622}
for iteration_variable in content_ratings:
    print(iteration_variable)
```

- Compute the frequency for defined intervals:

```
data_sizes = {'0 - 10 MB': 0, '10 - 50 MB': 0, '50 - 100 MB': 0,
                    '100 - 500 MB': 0, '500 MB +': 0}

for row in apps_data[1:]:
    data_size = float(row[2])
    if data_size <= 10000000:
        data_sizes['0 - 10 MB'] += 1
    elif 10000000 < data_size <= 50000000:
        data_sizes['10 - 50 MB'] += 1
    elif 50000000 < data_size <= 100000000:
        data_sizes['50 - 100 MB'] += 1
    elif 100000000 < data_size <= 500000000:
        data_sizes['100 - 500 MB'] += 1
    elif data_size > 500000000:
        data_sizes['500 MB +'] += 1
print(data_sizes)
```

Concepts

- We can check if a certain value exists in the dictionary as a key using an `in` operator. An `in` expression always returns a Boolean value.
- We also call the number of times a unique value occurs the **frequency**. We call tables that map unique values to their frequencies **frequency tables**.
- When we iterate over a dictionary with a `for` loop, we loop over the dictionary keys by default.

Resources

- Dictionaries in Python

Python Functions: Using Built-in Functions and Creating Functions: Takeaways

Syntax

- Create a function with a single parameter:

```
def square(number):
    return number**2
```

- Create a function with more than one parameter:

```
def add(x, y):
    return x + y
```

- Directly return the result of an expression:

```
def square(a_number):
    return a_number * a_number
```

Concepts

- Generally, a function displays this pattern:
    - It takes in an input.
    - It processes that input.
    - It returns output.
- In Python, we have **built-in functions** like `sum()` , `max()` , `min()` , `len()` , and `print()` , and functions that we create ourselves.
- Structurally, a function contains a header (which contains the `def` statement), a body, and a `return` statement.
- We call input variables **parameters**, and we call the various values that parameters take **arguments**. In `def square(number)` , the `number` variable is a parameter. In `square(number=6)` , the value `6` is an argument that passes to the parameter `number` .

Resources

- Functions in Python

Python Functions: Arguments, Parameters, and Debugging: Takeaways

Syntax

- Write a single function to generate the frequency tables for any column we want:

```
def freq_table(index):
    frequency_table = {}
    for row in apps_data[1:]:
        value = row[index]
        if value in frequency_table:
            frequency_table[value] += 1
        else:
            frequency_table[value] = 1
    return frequency_table
ratings_ft = freq_table(7)
```

- Define a function with multiple parameters:

```
def add(a, b):
    a_sum = a + b
    return a_sum
print(add(a=9, b=11))
```

- Use named arguments and positional arguments:

```
def subtract(a, b):
    return a - b
print(subtract(a=10, b=7))
print(subtract(b=7, a=10))
print(subtract(10,7))
```

- Reuse functions inside other functions:

```
def find_sum(a_list):
    a_sum = 0
    for element in a_list:
        a_sum += float(element)
    return a_sum

def find_length(a_list):
    length = 0
    for element in a_list:
        length += 1
    return length

def mean(a_list_of_numbers):
    return find_sum(a_list_of_numbers) / find_length(a_list_of_numbers)
```

Concepts

- Python allows us to use multiple parameters for the functions we create.
- We call arguments that we pass by name are called **keyword arguments** (the parameters yield the name). When we use multiple keyword arguments, the order we use doesn't make any practical difference.
- We call arguments that we pass by position **positional arguments**. When we use multiple positional arguments, the order we use matters.
- Positional arguments are often advantageous, because they involve less typing and can speed up our workflow. However, we need to pay extra attention to the order we use to avoid incorrect mappings that can lead to logical errors.
- Reusing functions inside other functions enables us to build complex functions by *abstracting away* function definitions.
- In programming, we call errors bugs. We call the process of fixing an error **debugging**.
- Debugging more complex functions can be more of a challenge, but we can find the **bugs** by reading the **traceback**.

Resources

- Functions in Python

Python Functions: Built-in Functions and Multiple Return Statements: Takeaways

Syntax

- Initiate parameters with **default arguments**:

```
def add_value(x, constant=3.14):
    return x + constant
```

- Use **multiple return statements**:

```
def sum_or_difference(a, b, return_sum):
    if return_sum:
        return a + b
    return a - b
```

- Not using the else clause:

```
def sum_or_difference(a, b, return_sum=True):
    if return_sum:
        return a + b
    return a - b
```

Concepts

- We need to avoid using the name of a built-in function to name a function or a variable because this overwrites the built-in function. Also avoid naming variables using the names of the built-in functions because this also causes unwanted interference.
- Virtually every code editor highlights built-in functions.
- Each built-in function is well documented in the official Python **documentation**.
- It's possible to use **multiple `return` statements**. Combining `return` with an `if` statement and an `else` clause, for example.

Resources

- Python official documentation
- Style guide for Python code

Python Functions: Returning Multiple Variables and Function Scopes: Takeaways

Syntax

- Return **multiple variables**:

```
def sum_and_difference(a, b):
    a_sum = a + b
    difference = a - b
    return a_sum, difference
sum_1, diff_1 = sum_and_difference(15, 10)
```

- Lists versus tuples:

```
a_list = [1, 'a', 10.5]
a_tuple = (1, 'a', 10.5)
```

- Use return statements in function bodies:

```
def price(item, cost):
    return "The " + item + " costs $" + str(cost) + "."

print(price("chair", 40.99))
```

- Use print statements in function bodies:

```
def price(item, cost):
    print("The " + item + " costs $" + str(cost) + ".")

price("chair", 40.99)
```

Concepts

- Parameters and return statements aren't mandatory when we create a function.

```
def print_constant():
    x = 3.14
    print(x)
```

- The code inside a function definition executes only when we call the function.
- When we call a function, the variables defined inside the function definition are saved into a temporary memory that is erased immediately after the function finishes running. The temporary memory associated with a function is isolated from the memory associated with the main program (the main program is the part of the program outside function definitions).
- We call the part of a program where we can access a variable the scope. The variables defined in the main program are in the global scope, while the variables defined inside a function are in the local scope.
- Python searches the global scope if a variable isn't available in the local scope, but the reverse doesn't apply. Python won't search the local scope if it doesn't find a variable in the global scope. Even if it searched the local scope, the memory associated with a function is temporary, so the search would be pointless.

Resources

- Python official documentation
- Style guide for Python code

Project: Learn and Install Jupyter Notebook: Takeaways

Syntax

MARKDOWN SYNTAX

- Add italics and bold text:

```
*Italics*
**Bold**
```

- Add headers (titles) of various sizes:

```
# header one
## header two
```

- Add hyperlinks and images:

```
[Link](http://a.com)
```

- Add block quotes:

```
> Blockquote
```

- Add lists:

```
*
*
*
```

- Add horizontal lines:

```
---
```

- Add inline code:

```
`Inline code with backticks`
```

- Add code blocks

~~~
```
code
```
~~~

JUPYTER NOTEBOOK SPECIAL COMMAND

- Displaying the code execution history:

```
%history -p
```

Concepts

- Jupyter Notebook is much more complex than a code editor. Jupyter allows us to do the following:
  - Type and execute code

    - Add accompanying text to our code (including math equations)
    - Add visualizations
- Jupyter can run in a browser, and we often use it to create compelling data science projects that we can easily share with other people.
- A notebook is a file created using Jupyter notebooks. We can easily share and distribute notebooks so people can view our work.
- Types of modes in Jupyter:
    - Jupyter is in edit mode whenever we type in a cell — a small pencil icon appears to the right of the menu bar.
    - Jupyter is in command mode whenever we press `Esc` or whenever we click outside of the cell — the pencil to the right of the menu bar disappears.
- State refers to what a computer remembers about a program.
- We can convert a code cell to a Markdown cell to add text to explain our code. Markdown syntax allows us to use keyboard symbols to format our text.
- Installing the Anaconda distribution installs both Python and Jupyter on your computer.

Keyboard Shortcuts

- Some of the most useful keyboard shortcuts in command mode include the following:
    - `Ctrl` + `Enter`: run selected cell
    - `Shift` + `Enter`: run cell, select below
    - `Alt` + `Enter`: run cell, insert below
    - `Up`: select cell above
    - `Down`: select cell below
    - `Enter`: enter edit mode
    - `A`: insert cell above
    - `B`: insert cell below
    - `D`, `D` (press D twice): delete selected cell
    - `Z`: undo cell deletion
    - `S`: save and checkpoint
    - `Y`: convert to code cell
    - `M`: convert to Markdown cell
- Some of the most useful keyboard shortcuts in edit mode include the following:
    - `Ctrl` + `Enter`: run selected cell
    - `Shift` + `Enter`: run cell, select below
    - `Alt` + `Enter`: run cell, insert below
    - `Up`: move cursor up
    - `Down`: move cursor down
    - `Esc`: enter command mode
    - `Ctrl` + `A`: select all
    - `Ctrl` + `Z`: undo
    - `Ctrl` + `Y`: redo

    - `Ctrl` + `S`: save and checkpoint
    - `Tab` : indent or code completion
    - `Shift` + `Tab`: tooltip

Resources

- Jupyter Notebook tutorial
- Jupyter Notebook tips and tricks
- Markdown syntax
- Installing Anaconda

Cleaning and Preparing Data in Python: Takeaways

Syntax

TRANSFORMING AND CLEANING STRINGS

- Replace a substring within a string:

```
green_ball = "red ball".replace("red", "green")
```

- Remove a substring:

```
friend_removed = "hello there friend!".replace(" friend", "")
```

- Remove a series of characters from a string:

```
bad_chars = ["'", ",", ".", "!"]
string = "We'll remove apostrophes, commas, periods, and exclamation marks!"
for char in bad_chars:
    string = string.replace(char, "")
```

- Convert a string to title cases:

```
Hello = "hello".title()
```

- Check a string for the existence of a substring:

```
if "car" in "carpet":
    print("The substring was found.")
else:
    print("The substring was not found.")
```

- Split a string into a list of strings:

```
split_on_dash = "1980-12-08".split("-")
```

- Slice characters from a string by position:

```
first_five_chars = "This is a long string."[:5]
```

- Concatenate strings:

```
superman = "Clark" + " " + "Kent"
```

Concepts

- When working with comma-separated value (CSV) data in Python, it's common to have your data in a "list of lists" format, where each item of the internal lists is a string.
- If you have numeric data stored as strings, sometimes you will need to remove and replace certain characters before you can convert the strings to numeric types, like `int` and `float` .
- Strings in Python are made from the same underlying data type as lists, which means you can index and slice specific characters from strings like you can lists.

Resources

- Python Documentation: String Methods

Python Data Analysis Basics: Takeaways

Syntax

STRING FORMATTING AND FORMAT SPECIFICATIONS

- Insert values into a string in order:

```
continents = "France is in {} and China is in {}".format("Europe", "Asia")
```

- Insert values into a string by position:

```
squares = "{0} times {0} equals {1}".format(3,9)
```

- Insert values into a string by name:

```
population = "{name}'s population is {pop} million".format(name="Brazil", pop=209)
```

- Format specification for precision of two decimal places:

```
two_decimal_places = "I own {:.2f}% of the company".format(32.5548651132)
```

- Format specification for comma separator:

```
india_pop = "The approximate population of {} is {:,}".format("India",1324000000)
```

- Order for format specification when using precision and comma separator:

```
balance_string = "Your bank balance is {:,.2f}".format(12345.678)
```

Concepts

- The `str.format()` method allows you to insert values into strings without explicitly converting them.
- The `str.format()` method also accepts optional format specifications, which you can use to format values so they are easier to read.

Resources

- Python Documentation: Format Specifications
- PyFormat: Python String Formatting Reference

Object-Oriented Python: Takeaways

Syntax

- Define an empty class:

```
class MyClass:
    pass
```

- Instantiate an object of a class:

```
class MyClass:
    pass
mc_1 = MyClass()
```

- Define an init function in a class to assign an attribute at instantiation:

```
class MyClass:
    def __init__(self, param_1):
        self.attribute_1 = param_1
mc_2 = MyClass("arg_1")
```

- Define a method inside a class and call it on an instantiated object:

```
class MyClass:
    def __init__(self, param_1):
        self.attribute_1 = param_1
    def add_20(self):
        self.attribute_1 += 20
mc_3 = MyClass(10) # mc_3.attribute is 10
mc_3.add_20()       # mc_3.attribute is 30
```

Concepts

- In **Object-Oriented Programming**, the fundamental building blocks are objects.
  - It differs from **Procedural** programming, which executes sequential steps.
- An **object** is an entity that stores data.
- A **class** describes an object's type. It defines the following:
  - What data is stored in the object, known as attributes
  - What actions the object can do, known as methods
- An **attribute** is a variable that belongs to an instance of a class.
- A **method** is a function that belongs to an instance of a class.
- We access attributes and methods using **dot notation**. Attributes do not use parentheses, whereas methods do.
- An **instance** describes a specific example of a class. For instance, in the code `x = 3` , `x` is an instance of the type `int` .
  - We call creating an object **instantiation**.

- A **class definition** is code that defines how a class behaves, including all methods and attributes.
- The init method is a special method that runs at the moment of an object's instantiation.
    - The init method (`__init__()`) is one of a number of special methods that Python defines.
- All methods must include `self` , representing the object instance, as their first parameter.
- It is convention to start the name of any attributes or methods that we don't intend for external use with an underscore.

Resources

- Python Documentation: Classes

Working with Dates and Times in Python: Takeaways

Syntax

IMPORTING MODULES AND DEFINITIONS

- Importing a whole module:

```
import csv
csv.reader()
```

- Importing a whole module with an alias:

```
import csv as c
c.reader()
```

- Importing a single definition:

```
from csv import reader
reader()
```

- Importing multiple definitions:

```
from csv import reader, writer
reader()
writer()
```

- Importing all definitions:

```
from csv import *
```

WORKING WITH THE `DATETIME` MODULE

- All examples below presume the following import code:

```
import datetime as dt
```

- Creating datetime.datetime object given a month, year, and day:

```
eg_1 = dt.datetime(1985, 3, 13)
```

- Creating a datetime.datetime object from a string:

```
eg_2 = dt.datetime.strptime("24/12/1984", "%d/%m/%Y")
```

- Converting a datetime.datetime object to a string:

```
dt_object = dt.datetime(1984, 12, 24)
dt_string = dt_object.strftime("%d/%m/%Y")
```

- Instantiating a datetime.time object:

```
eg_3 = datetime.time(hour=0, minute=0, second=0, microsecond=0)
```

- Retrieving a part of a date stored in the datetime.datetime object:

```
eg_1.day
```

- Creating a datetime.time object from a datetime.datetime object:

```
d2_dt = dt.datetime(1946, 9, 10)
d2 = d2_dt.time()
```

- Creating a datetime.time object from a string:

```
d3_str = "17 February 1963"
d3_dt = dt.datetime.strptime(d3_str, "%d %B %Y")
d3 = d3_dt.time()
```

- Instantiating a datetime.timedelta object:

```
eg_4 = dt.timedelta(weeks=3)
```

- Adding a time period to a datetime.datetime object:

```
d1 = dt.date(1963, 2, 26)
d1_plus_1wk = d1 + dt.timedelta(weeks=1)
```

Concepts

- The datetime module contains the following classes:
  - `datetime.datetime` : for working with date and time data
  - `datetime.time` : for working with time data only
  - `datetime.timedelta` : for representing time periods
- Time objects behave similarly to datetime objects for the following reasons:
  - They have attributes like `time.hour` and `time.second` that you can use to access individual time components.
  - They have a `time.strftime()` method, which you can use to create a formatted string representation of the object.
- The timedelta type represents a period of time (e.g., 30 minutes or two days).
- Common format codes when working with `datetime.datetime.strptime` :

Strftime Code	Meaning	Examples
`%d`	Day of the month as a zero-padded number[1]	`04`
`%A`	Day of the week as a word[2]	`Monday`
`%m`	Month as a zero-padded number[1]	`09`
`%Y`	Year as a four-digit number	`1901`
`%y`	Year as a two-digit number with zero-padding[1, 3]	`01` (2001) `88` (1988)
`%B`	Month as a word[2]	`September`
`%H`	Hour in 24 hour time as zero-padded number[1]	`05` (5 a.m.) `15` (3 p.m.)
`%p`	a.m. or p.m.[2]	`AM`
`%I`	Hour in 12 hour time as zero-padded number[1]	`05` (5 a.m., or 5 p.m. if `AM` / `PM` indicates otherwise)
`%M`	Minute as a zero-padded number[1]	`07`

1. The strptime parser will parse non-zero padded numbers without raising an error.

2. Date parts containing words will be interpreted using the locale settings on your computer, so strptime won't be able to parse 'febrero' (february in Spanish) if your locale is set to an English language locale.

3. Year values from 00-68 will be interpreted as 2000-2068, with values 69-99 interpreted as 1969-1999.

- Operations between timedelta, datetime, and time objects (we can substitute datetime with time):

Operation	Explanation	Resultant Type
`datetime - datetime`	Calculate the time between two specific dates/times	timedelta
`datetime - timedelta`	Subtract a time period from a date or time.	datetime
`datetime + timedelta`	Add a time period to a date or time.	datetime
`timedelta + timedelta`	Add two periods of time together	timedelta
`timedelta - timedelta`	Calculate the difference between two time periods.	timedelta

Resources

- Python Documentation - Datetime module
- Python Documentation: Strftime/Strptime Codes
- strftime.org