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:

```
last_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 are strings.
- 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):

        se1f.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):

        se1f.attribute_1 = param_1

    def add_20(se1f):

        se1f.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, where sequential steps are executed.

- An **object** is an entity that stores data.
- A **class** describes an object's type. It defines:
    - 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.
- Attributes and methods are accessed 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`.
    - When an object is created, it is known as **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 an object is instantiated.
    - 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 aren't intended 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` (5a.m.) `15` (3p.m.)
`%p`	a.m. or p.m.[2]	`AM`
`%I`	Hour in 12 hour time as zero-padded number[1]	`05` (5a.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 'febrerd' (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 70-99 interpreted as 1970-1999.

- Operations between timedelta, datetime, and time objects (datetime can be substituted 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