Reading 5

☆ Python Requests Library

- Requests package is the go-to package for making HTTP requests.
  - ↳ It is not built in, it is a 3rd party library.

- A GET request retrieves data from a specified resource.

- Ex : Making a get request :

```python
>>> import requests
>>> response = requests.get("https://api.github.com")
```

- response is an object that contains the results of your request.
- Attributes :
  - status code : the status of your request

```python
if response.status_code == 200:
    print("Success!")
elif response.status_code == 404:
    print("Not Found.")
```

↳ Also if you use response in a Boolean context it evaluates to :
- ↳ True if status code less than 400
- ↳ False if otherwise

- raise_for_status()

Python raise_error.py

```python
import requests
from requests.exceptions import HTTPError

URLS = ["https://api.github.com", "https://api.github.com/invalid"]

for url in URLS:
    try:
        response = requests.get(url)
        response.raise_for_status()
    except HTTPError as http_err:
        print(f"HTTP error occurred: {http_err}")
    except Exception as err:
        print(f"Other error occurred: {err}")
    else:
        print("Success!")
```

← raises an HTTPError for status codes between 400 and 600.

- **.content()**
  - ↳ the responses content in bytes

- **.text()**
  - ↳ the response content as a string
    - ↳ It is serialized JSON content.

- **.json()**
  - ↳ automatically converts response object into a python dictionary.

- **.headers()**
  - ↳ returns a dictionary like object, allowing you to access headers by key.
  - ↳ headers are case insensitive so you can access them without worrying about capitilization

Adding Query String Parameters

- Makes your GET request customizable.
- Ex: Using Github's repository search API to look for popular python repo's:

Python search_popular_repos.py

```python
import requests

response = requests.get(
    "https://api.github.com/search/repositories",
    params={"q": "language:python", "sort": "stars", "order": "desc"},
)

json_response = response.json()
popular_repositories = json_response["items"]
for repo in popular_repositories[:3]:
    print(f"Name: {repo['name']}")
    print(f"Description: {repo['description']}")
    print(f"Stars: {repo['stargazers_count']}\n")
```

you can pass params either as a dictionary or as a list of tuples.

Customizing Request Headers:

```python
import requests

response = requests.get(
    "https://api.github.com/search/repositories",
    params={"q": '"real python"'},
    headers={"Accept": "application/vnd.github.text-match+json"},
)

json_response = response.json()
first_repository = json_response["items"][0]
print(first_repository["text_matches"][0]["matches"])
```

- The "Accept" header tells the server what content types your application can handle.

Example of Using Other HTTP Methods:

```python
>>> import requests

>>> requests.get("https://httpbin.org/get")
<Response [200]>
>>> requests.post("https://httpbin.org/post", data={"key": "value"})
<Response [200]>
>>> requests.put("https://httpbin.org/put", data={"key": "value"})
<Response [200]>
>>> requests.delete("https://httpbin.org/delete")
<Response [200]>
>>> requests.head("https://httpbin.org/get")
<Response [200]>
>>> requests.patch("https://httpbin.org/patch", data={"key": "value"})
<Response [200]>
>>> requests.options("https://httpbin.org/get")
<Response [200]>
```

can also be passed as list of tuples

```python
>>> import requests

>>> requests.post("https://httpbin.org/post", data={"key": "value"})
<Response [200]>
```

sending JSON data

```python
>>> response = requests.post("https://httpbin.org/post", json={"key": "value"}
>>> json_response = response.json()
>>> json_response["data"]
'{"key": "value"}'
>>> json_response["headers"]["Content-Type"]
'application/json'
```

Inspecting Prepared Request

- You can view the PreparedRequest object by accessing .request on a Response object:

```python
>>> import requests

>>> response = requests.post("https://httpbin.org/post", json={"key":"value"})

>>> response.request
<PreparedRequest [POST]>

>>> response.request.headers["Content-Type"]
'application/json'

>>> response.request.url
'https://httpbin.org/post'

>>> response.request.body
b'{"key": "value"}'
```

Use Authentication

Ex: Using auth parameter to pass credentials:

```python
>>> import requests

>>> response = requests.get(
...     "https://httpbin.org/basic-auth/user/passwd",
...     auth=("user", "passwd")
... )

>>> response.status_code
200
>>> response.request.headers["Authorization"]
'Basic dXNlcjpwYXNzd2Q='
```

- Ex: Creating a subclass of AuthBase for token based authentication:

custom_token_auth.py

```python
from requests.auth import AuthBase

class TokenAuth(AuthBase):
    """Implements a token authentication scheme."""

    def __init__(self, token):
        self.token = token

    def __call__(self, request):
        """Attach an API token to the Authorization header."""
        request.headers["Authorization"] = f"Bearer {self.token}"
        return request
```

```python
>>> import requests
>>> from custom_token_auth import TokenAuth

>>> token = "<YOUR_GITHUB_PA_TOKEN>"
>>> response = requests.get(
...     "https://api.github.com/user",
...     auth=TokenAuth(token)
... )

>>> response.status_code
200
>>> response.request.headers["Authorization"]
'Bearer ghp_b...Tx'
```

Communicating Effectively with Servers:

- The way of communicating with secure sites over HTTP is by establishing an encrypted connection using Transport Layer Security (TLS)
  ↳ TLS is successor of Secure Sockets Layer (SSL)

- Usually you don't need to make adjustments for verifying the digital certificate of servers.
- Ex: Providing your own certificate bundle!

Python
```python
>>> import requests

>>> requests.get(
...     "https://internal-api.company.com",
...     verify="/path/to/company-ca.pem"
... )
<Response [200]>
```

Improve Performance

- Set Request Timeouts

  - By default requests will wait indefinetly on the response, so you should always specify a timeout duration.

Python
```python
>>> requests.get("https://api.github.com", timeout=1)
<Response [200]>

>>> requests.get("https://api.github.com", timeout=0.01)
Traceback (most recent call last):
  ...
requests.exceptions.ConnectTimeout
↳ HTTPSConnectionPool(host='api.github.com', port=443):
↳ Max retries exceeded with url: / (Caused by ConnectTimeoutError(...))
```

- You can also pass a tuple to timeout with the following two elements:
  1. Connection Timeout: The amount of time it allows the client to establish a connection to the server.
  2. Read Timeout: The time it'll wait for a response once the client has established a connection.

- Ex

Python — timeout_catcher.py

```python
import requests
from requests.exceptions import Timeout

try:
    response = requests.get("https://api.github.com", timeout=(3.05, 5))
except Timeout:
    print("The request timed out")
else:
    print("The request did not time out")
```

seconds (both 3.05 and 5)

Reuse Connections With Session Objects.

- Sessions are used to persist parameters across queries.
- Ex: Using same authentication across multiple requests:

Python — persist_info_with_session.py

```python
import requests
from custom_token_auth import TokenAuth

TOKEN = "<YOUR_GITHUB_PA_TOKEN>"

with requests.Session() as session:
    session.auth = TokenAuth(TOKEN)

    first_response = session.get("https://api.github.com/user")
    second_response = session.get("https://api.github.com/user")

print(first_response.headers)
print(second_response.json())
```

- Primary performance optimization is persistent connections.
  - ↳ When using a session, any connections are kept in a connection pool.
  - ↳ When your app wants to connect to the same server again, it'll reuse the connection rather than establishing a new one.

Retry Failed Requests

- When a request fails, you might want your application to retry the same request.
  - ↳ However, requests won't do this by default.
  - ↳ So you need to implement a custom transport adapter:

Python retry_twice.py

```python
import requests
from requests.adapters import HTTPAdapter
from requests.exceptions import RetryError
from urllib3.util.retry import Retry

retry_strategy = Retry(
    total=2,
    status_forcelist=[429, 500, 502, 503, 504]
)
github_adapter = HTTPAdapter(max_retries=retry_strategy)

with requests.Session() as session:
    session.mount("https://api.github.com", github_adapter)
    try:
        response = session.get("https://api.github.com/")
    except RetryError as err:
        print(f"Error: {err}")
```

(Handwritten annotations: `total=2` ← How many times to try; `status_forcelist` ← What status codes to retry for)

Reading 5 Page 2

✪ Reading and Writing CSV Files In Python

- CSV files are plain text files that use delimeters to create tabular data.
  - ↳ delimeters are often comma, tab (\t), colon, semicolon, and more

- Ex Using csv library:

CSV

```
name,department,birthday month
John Smith,Accounting,November
Erica Meyers,IT,March
```

Here's code to read it:

Python

```python
import csv

with open('employee_birthday.csv') as csv_file:
    csv_reader = csv.reader(csv_file, delimiter=',')
    line_count = 0
    for row in csv_reader:
        if line_count == 0:
            print(f'Column names are {", ".join(row)}')
            line_count += 1
        else:
            print(f'\t{row[0]} works in the {row[1]} department, and was born in {row[2]}.')
            line_count += 1
    print(f'Processed {line_count} lines.')
```

This results in the following output:

Shell

```
Column names are name, department, birthday month
    John Smith works in the Accounting department, and was born in November.
    Erica Meyers works in the IT department, and was born in March.
Processed 3 lines.
```

- Ex Same example but reading csv to dictionary:

CSV

```
name,department,birthday month
John Smith,Accounting,November
Erica Meyers,IT,March
```

Here's the code to read it in as a dictionary this time:

Python

```python
import csv

with open('employee_birthday.csv', mode='r') as csv_file:
    csv_reader = csv.DictReader(csv_file)
    line_count = 0
    for row in csv_reader:
        if line_count == 0:
            print(f'Column names are {", ".join(row)}')
            line_count += 1
        print(f'\t{row["name"]} works in the {row["department"]} department, a
        line_count += 1
    print(f'Processed {line_count} lines.')
```

- The first line of the csv is assumed to have the dictionary keys, unless specified otherwise.

This results in the same output as before:

Shell

```
Column names are name, department, birthday month
    John Smith works in the Accounting department, and was born in November.
    Erica Meyers works in the IT department, and was born in March.
Processed 3 lines.
```

Optional Python CSV reader Parameters:

- delimeter
- quotechar : Specifies the character used to surround fields that contain the delimeter character. Default is double quotes.
- escapechar: Specifies char used to escape the delimeter character, in case quotes aren't used.

Writing CSV Files with csv

```
import csv

with open('employee_file.csv', mode='w') as employee_file:
    employee_writer = csv.writer(employee_file, delimiter=',', quotechar='"', quoting=csv.QUOTE_MINIMAL)

    employee_writer.writerow(['John Smith', 'Accounting', 'November'])
    employee_writer.writerow(['Erica Meyers', 'IT', 'March'])
```

quoting parameter:

= csv.QUOTE_MINIMAL, then .writerow() only quotes fields if they contain delimeter or quotechar

= csv.QUOTE_ALL, then .writerow() will quote all fields

= csv.QUOTE_NONNUMERIC, then .writerow() quotes all fields contain text, all numeric fields converted to floats.

= csv.QUOTE_NONE, then nothing quoted, delimeter escaped by specified escapechar.

Writing CSV file from dictionary

```
Python
import csv

with open('employee_file2.csv', mode='w') as csv_file:
    fieldnames = ['emp_name', 'dept', 'birth_month']
    writer = csv.DictWriter(csv_file, fieldnames=fieldnames)

    writer.writeheader()
    writer.writerow({'emp_name': 'John Smith', 'dept': 'Accounting', 'bir
    writer.writerow({'emp_name': 'Erica Meyers', 'dept': 'IT', 'birth_mon
```

Parsing CSV with Pandas

Ex

CSV

```
Name,Hire Date,Salary,Sick Days remaining
Graham Chapman,03/15/14,50000.00,10
John Cleese,06/01/15,65000.00,8
Eric Idle,05/12/14,45000.00,10
Terry Jones,11/01/13,70000.00,3
Terry Gilliam,08/12/14,48000.00,7
Michael Palin,05/23/13,66000.00,8
```

Python

```python
import pandas
df = pandas.read_csv('hrdata.csv', index_col='Name')
print(df)
```

Now the `Name` field is our `DataFrame` index:

Shell

```
               Hire Date   Salary  Sick Days remaining
Name
Graham Chapman  03/15/14  50000.0                   10
John Cleese     06/01/15  65000.0                    8
Eric Idle       05/12/14  45000.0                   10
Terry Jones     11/01/13  70000.0                    3
Terry Gilliam   08/12/14  48000.0                    7
Michael Palin   05/23/13  66000.0                    8
```

- Now changing Hire Date to Date from string:

Python

```python
import pandas
df = pandas.read_csv('hrdata.csv', index_col='Name', parse_dates=['Hire Date'])
print(df)
```

Notice the difference in the output:

Shell

```
                Hire Date   Salary  Sick Days remaining
Name
Graham Chapman 2014-03-15  50000.0                   10
John Cleese    2015-06-01  65000.0                    8
Eric Idle      2014-05-12  45000.0                   10
Terry Jones    2013-11-01  70000.0                    3
Terry Gilliam  2014-08-12  48000.0                    7
Michael Palin  2013-05-23  66000.0                    8
```

- Changing Column Names

Python

```python
import pandas
df = pandas.read_csv('hrdata.csv',
            index_col='Employee',
            parse_dates=['Hired'],
            header=0,  ← "ignore 1st row of csv specifying column names)
            names=['Employee', 'Hired','Salary', 'Sick Days'])← new column names
print(df)
```

Notice that, since the column names changed, the columns specified in the `index_col` and `parse_dates` optional parameters must also be changed. This now results in the following output:

Shell

```
                    Hired   Salary  Sick Days
Employee
Graham Chapman 2014-03-15  50000.0         10
John Cleese    2015-06-01  65000.0          8
Eric Idle      2014-05-12  45000.0         10
Terry Jones    2013-11-01  70000.0          3
Terry Gilliam  2014-08-12  48000.0          7
Michael Palin  2013-05-23  66000.0          8
```

- To write a csv from a dataFrame:

```python
df.to_csv('my.csv')
```

☆ How to Use sorted() and .sort() in python.

- sorted() returns a new sorted list from the elements of any iterable, without modifying original.
- .sort() modifies a list in place and does not return a value.

Sorted()

Python

```
>>> numbers = [6, 9, 3, 1]
>>> numbers_sorted = sorted(numbers)
>>> numbers_sorted
[1, 3, 6, 9]
>>> numbers
[6, 9, 3, 1]
```

- Sorted() returns a list, but you can cast to other iterables!

Python

```
>>> numbers_tuple = (6, 9, 3, 1)
>>> numbers_set = {5, 10, 1, 0}
>>> numbers_tuple_sorted = sorted(numbers_tuple)
>>> numbers_set_sorted = sorted(numbers_set)
>>> numbers_tuple_sorted
[1, 3, 6, 9]
>>> numbers_set_sorted
[0, 1, 5, 10]
>>> tuple(numbers_tuple_sorted)
(1, 3, 6, 9)
>>> set(numbers_set_sorted)
{0, 1, 10, 5}
```

- Sorted() on strings!

Python

```
>>> string_number_value = "34521"
>>> sorted(string_number_value)
['1', '2', '3', '4', '5']

>>> string_value = "I like to sort"
>>> sorted(string_value)
[' ', ' ', ' ', 'I', 'e', 'i', 'k', 'l', 'o', 'o', 'r', 's', 't', 't']
```

- Sorted() with .split() to sort words

Python

```
>>> string_value = "I like to sort"
>>> sorted_string = sorted(string_value.split())
>>> sorted_string
['I', 'like', 'sort', 'to']
```

- Limitations of sorted()

  - returns error if you attempt to sort non-comparable data.
    - ↳ throws a TypeError

  - Sort stability: when you sort equal values, they'll retain their original order in the output.
    - ↳ not really a limitation but good to know

  - Ex: shorter strings sorted first if "identical"

```python
>>> sorted(["aaa", "ab", "a"])
['a', 'aaa', 'ab']
```

  - ord() function returns a characters unicode.
    - ↳ Ex ord("b") returns 98

- Customizing sorted() with keyword arguments:

  - reverse = True to sort in descending order

  - Argument "key" (key = len)
    - ↳ expects a function to be passed to it (function MUST only take one argument)
    - ↳ function is used to determine the resulting order
    - ↳ you can do key=str.lower to sort strings case insensitive

```python
>>> def reverse_word(word):
...     return word[::-1]
...
>>> words = ["cookie", "banana", "donut", "pizza"]
>>> sorted(words, key=reverse_word)
['banana', 'pizza', 'cookie', 'donut']
```

Anonymous function being used ↓

```python
>>> words = ["cookie", "banana", "donut", "pizza"]
>>> sorted(words, key=lambda word: word[::-1])
['banana', 'pizza', 'cookie', 'donut']
```

Ordering Values with .sort()

- Sort() is method of list class
  - ↳ so it can only be used on lists
- Sort() returns None
- Sort() modifies list in place

- Ex

```python
>>> tuple_val = (5, 1, 3, 5)
>>> tuple_val.sort()
Traceback (most recent call last):
  ...
AttributeError: 'tuple' object has no attribute 'sort'

>>> values_to_sort = list(tuple_val)
>>> returned_from_sort = values_to_sort.sort()
>>> print(returned_from_sort)
None

>>> print(values_to_sort)
[1, 3, 5, 5]
```

- you can use the same keyword arguments as sorted()

Reading 5 continued

★ Functional Programming in Python: When and How to use it

- A pure function is a function whose output value follows solely from its input values without any observable side effects.

- In functional programming, a program consists primarily of the evaluation of pure functions.
  ↳ Computation proceeds by nested or composed function calls without changes to state or mutable data.

- In Python everything is an object.
- In Python, functions are first-class citizens.
  ↳ This means functions have same characteristics as values like strings and numbers.
  ↳ Ex of such behavior:

Python
```python
>>> def func():
...     print("I am function func()!")
...

>>> func()
I am function func()!

>>> another_name = func
>>> another_name()
I am function func()!
```

Python
```python
>>> def func():
...     print("I am function func()!")
...

>>> print("cat", func, 42)
cat <function func at 0x7f81b4d29bf8> 42

>>> objects = ["cat", func, 42]
>>> objects[1]
<function func at 0x7f81b4d29bf8>
>>> objects[1]()
I am function func()!

>>> d = {"cat": 1, func: 2, 42: 3}
>>> d[func]
2
```

↳ Ex function composition: [passing function (not return value) to another function]

Python
```python
>>> def inner():
...     print("I am function inner()!")
...

>>> def outer(function):
...     function()
...

>>> outer(inner)
I am function inner()!
```

- Sometimes the function being passed in is referred to as a callback because a "call back" to the inner function can modify the outer function's behavior.

Ex function defining a function and returning the function:

Python

```
1  >>> def outer():
2  ...     def inner():
3  ...         print("I am function inner()!")
4  ...     # Function outer() returns function inner()
5  ...     return inner
6  ...
7
8  >>> function = outer()
9  >>> function
10 <function outer.<locals>.inner at 0x7f18bc85faf0>
11 >>> function()
12 I am function inner()!
13
14 >>> outer()()
15 I am function inner()!
```

• Defining Anonymous (lambda) functions:

General Syntax:

optional comma separated list of parameter names

an expression which represents the lambda function's return value

Python

```
lambda <parameter_list>: <expression>
```

Ex assigning lambda function to variable (equivalent to normal function def)

Python

```
1  >>> def reverse(s):
2  ...     return s[::-1]
3  ...
4  >>> reverse("I am a string")
5  'gnirts a ma I'
6
7  >>> reverse = lambda s: s[::-1]
8  >>> reverse("I am a string")
9  'gnirts a ma I'
```

← in general this is NOT good practice

Ex defining and calling lambda function:

Python

```
>>> (lambda x1, x2, x3: (x1 + x2 + x3) / 3)(9, 6, 6)
7.0
>>> (lambda x1, x2, x3: (x1 + x2 + x3) / 3)(1.4, 1.1, 0.5)
1.0
```

} again, NOT good practice

Ex good use case of lambda function:

Python

```
>>> animals = ["ferret", "vole", "dog", "gecko"]
>>> sorted(animals, key=lambda s: -len(s))
['ferret', 'gecko', 'vole', 'dog']
```

Ex More complex lambda function with conditional expression:

```python
>>> (lambda x: "even" if x % 2 == 0 else "odd")(2)
'even'
>>> (lambda x: "even" if x % 2 == 0 else "odd")(3)
'odd'
```

Ex Returning tuple, list, and dict from lambda function:

```python
>>> (lambda x: (x, x ** 2, x ** 3))(3)
(3, 9, 27)

>>> (lambda x: [x, x ** 2, x ** 3])(3)
[3, 9, 27]

>>> (lambda x: {1: x, 2: x ** 2, 3: x ** 3})(3)
{1: 3, 2: 9, 3: 27}
```

- If you use a lambda function in an fstring, you must explicitly enclose it in parentheses.

Applying Function to an Iterable with map():

- With map() you can apply a function to each element in an iterable in turn.
  - ↳ returns an iterator that yields the results.
  - ↳ allows for concise code because it can take the place of an explicit loop.

- Syntax:

```python
map(<f>, <iterable>)
```

↳ returns iterator that yields the results of applying function <f> to each element of <iterable>.

- Ex

```python
>>> animals = ["cat", "dog", "hedgehog", "gecko"]
>>> map(reverse, animals)
<map object at 0x7fd3558cbef0>
```

- Remember map() returns a map object, which is an iterator NOT a list.
- So you can get desired output like this →

```python
>>> iterator = map(reverse, animals)
>>> for animal in iterator:
...     print(animal)
...
tac
god
gohegdeh
okceg

>>> iterator = map(reverse, animals)
>>> list(iterator)
['tac', 'god', 'gohegdeh', 'okceg']
```

Ex using map to convert data types instead of loop:

```python
>>> "+".join(map(str, [1, 2, 3, 4, 5]))
'1+2+3+4+5'
```

- Syntax for calling map with multible iterables:

```python
map(<f>, <iterable_1>, <iterable_2>, ..., <iterable_n>)
```

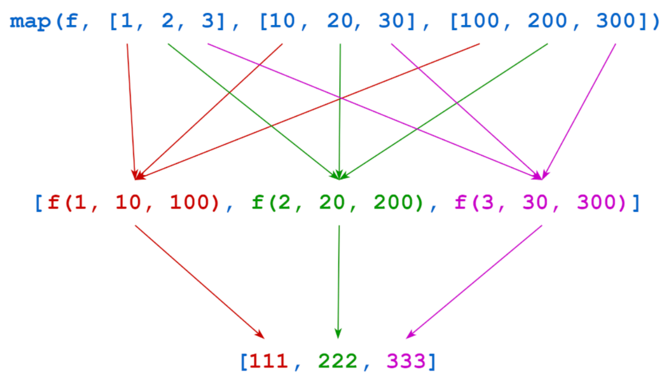
↳ applies <f> to the elements in each <iterable_i> in <u>parallel</u> and returns iterator that yields the results.

Ex

```python
>>> def add_three(a, b, c):
...     return a + b + c
...

>>> list(map(add_three, [1, 2, 3], [10, 20, 30], [100, 200, 300]))
[111, 222, 333]
```

- **Note:** It's important you pass as many iterables as your function takes inputs.



Ex Same concept using lambda:

```python
>>> list(
...     map(
...         lambda a, b, c: a + b + c,
...         [1, 2, 3, 4],
...         [10, 20, 30, 40],
...         [100, 200, 300, 400],
...     )
... )
[111, 222, 333, 444]
```

Selecting elements from an iterable with filter()

- Syntax:

```python
filter(<f>, <iterable>)
```

- Ex

```python
>>> list(filter(lambda x: x > 100, [1, 111, 2, 222, 3, 333]))
[111, 222, 333]
```

- Ex

```python
>>> animals = ["cat", "Cat", "CAT", "dog", "Dog", "DOG", "emu", "Emu", "EM

>>> def all_caps(s):
...     return s.isupper()
...
>>> list(filter(all_caps, animals))
['CAT', 'DOG', 'EMU']

>>> list(filter(lambda s: s.isupper(), animals))
['CAT', 'DOG', 'EMU']
```

- Ex Removing empty strings (empty strings are falsy in Python)

```python
>>> animals_and_empty_strings = ["", "cat", "dog", "", ""]

>>> list(filter(lambda s: s, animals_and_empty_strings))
['cat', 'dog']
```

Reducing an iterable to a Single Value with reduce():

- reduce() is no longer built in
  - ↳ have to "from functools import reduce"

- Syntax:

```python
reduce(<f>, <iterable>)
```

- Ex

```python
>>> def f(x, y):
...     return x + y
...

>>> from functools import reduce
>>> reduce(f, [1, 2, 3, 4, 5])
15
```

This call to `reduce()` produces the result 15 from the list `[1, 2, 3, 4, 5]` as follows:

```
1 + 2
f(1, 2) = 3
          ↓   3 + 3
        f(3, 3) = 6
                  ↓   6 + 4
                f(6, 4) = 10
                          ↓   10 + 5
                        f(10, 5) = 15
```

reduce(f, [1, 2, 3, 4, 5])

• Ex Using reduce to find max!

Python

```
>>> max([23, 49, 6, 32])
49

>>> def greater(x, y):
...     return x if x > y else y
...

>>> from functools import reduce
>>> reduce(greater, [23, 49, 6, 32])
49
```

• Calling reduce() with an initial value:

• Syntax:

Python

```
reduce(<f>, <iterable>, <initializer>)
```

• Ex

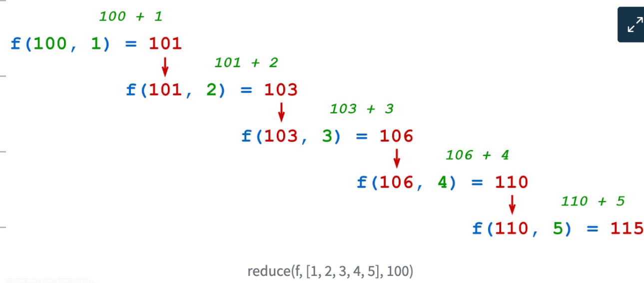
Python

```
>>> def f(x, y):
...     return x + y
...

>>> from functools import reduce
>>> reduce(f, [1, 2, 3, 4, 5], 100)  # (100 + 1 + 2 + 3 + 4 + 5)
115

>>> # Using lambda:
>>> reduce(lambda x, y: x + y, [1, 2, 3, 4, 5], 100)
115
```

Consider this diagram to better understand the sequence of function calls that Python goes through when you call `reduce()` with an initializer:



reduce(f, [1, 2, 3, 4, 5], 100)

☆ When to use a list comprehension in Python

- A list comprehension in Python is a tool for creating lists by iterating over an iterable and optionally applying a condition.

- A list comprehension can be faster than a for loop.
  - ↳ list comprehensions create lists.

- 3 ways to transform lists in python:
  1. for loops 2. map() 3. list comprehensions

- Syntax:

Python Syntax

```
new_list = [expression for member in iterable]
```

↑ can be the member itself, call to a method, or any valid expression that returns a value.

- Ex list comprehension

Python

```
>>> squares = [number * number for number in range(10)]
>>> squares
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

- Ex

Python

```
>>> prices = [1.09, 23.56, 57.84, 4.56, 6.78]
>>> TAX_RATE = .08
>>> def get_price_with_tax(price):
...     return price * (1 + TAX_RATE)
...

>>> final_prices = [get_price_with_tax(price) for price in prices]
>>> final_prices
[1.1772000000000002, 25.4448, 62.467200000000005, 4.9248, 7.322400000000001]
```

• More complete syntax!

Python Syntax

```
new_list = [expression for member in iterable if conditional]
```

• Ex

Python

```
>>> sentence = (
...     "The rocket, who was named Ted, came back "
...     "from Mars because he missed his friends."
... )
>>> def is_consonant(letter):
...     vowels = "aeiou"
...     return letter.isalpha() and letter.lower() not in vowels
...

>>> [char for char in sentence if is_consonant(char)]
['T', 'h', 'r', 'c', 'k', 't', 'w', 'h', 'w', 's', 'n', 'm', 'd',
 'T', 'd', 'c', 'm', 'b', 'c', 'k', 'f', 'r', 'm', 'M', 'r', 's', 'b',
 'c', 's', 'h', 'm', 's', 's', 'd', 'h', 's', 'f', 'r', 'n', 'd', 's']
```

• Syntax for changing a member value instead of filtering it out:

Python Syntax

```
new_list = [true_expr if conditional else false_expr for member in iterable]
```

• Ex

Python

```
>>> original_prices = [1.25, -9.45, 10.22, 3.78, -5.92, 1.16]
>>> [price if price > 0 else 0 for price in original_prices]
[1.25, 0, 10.22, 3.78, 0, 1.16]
```

or

Python

```
>>> def get_price(price):
...     return price if price > 0 else 0
...

>>> [get_price(price) for price in original_prices]
[1.25, 0, 10.22, 3.78, 0, 1.16]
```

Removing Duplicates With Set and Dictionary Comprehensions

• A set comprehension is just like list comprehension but uses curly braces

↳ the output contains no duplicates

↳ Ex

Python

```
>>> quote = "life, uh, finds a way"
>>> {char for char in quote if char in "aeiou"}
{'a', 'e', 'u', 'i'}
```

• Dictionary Comprehensions are similar, you just need to define a key:

```python
>>> {number: number * number for number in range(10)}
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25, 6: 36, 7: 49, 8: 64, 9: 81}
```

• Ex Using Walrus operator to assign values in list comprehension:

```python
>>> import random
>>> def get_weather_data():
...     return random.randrange(90, 110)
...

>>> [temp for _ in range(20) if (temp := get_weather_data()) >= 100]
[107, 102, 109, 104, 107, 109, 108, 101, 104]
```

walrus operator

Ex Creating dictionary via nested comprehension:

```python
>>> cities = ["Austin", "Tacoma", "Topeka", "Sacramento", "Charlotte"]
>>> {city: [0 for _ in range(7)] for city in cities}
{
    'Austin': [0, 0, 0, 0, 0, 0, 0],
    'Tacoma': [0, 0, 0, 0, 0, 0, 0],
    'Topeka': [0, 0, 0, 0, 0, 0, 0],
    'Sacramento': [0, 0, 0, 0, 0, 0, 0],
    'Charlotte': [0, 0, 0, 0, 0, 0, 0]
}
```

Ex Creating Matrix via nested comprehension:

```python
>>> [[number for number in range(5)] for _ in range(6)]
[
    [0, 1, 2, 3, 4],
    [0, 1, 2, 3, 4],
    [0, 1, 2, 3, 4],
    [0, 1, 2, 3, 4],
    [0, 1, 2, 3, 4],
    [0, 1, 2, 3, 4]
]
```

- When the size of a list becomes problematic, its often helpful to use a <u>generator</u> instead of list comprehension in python.

- A <u>generator</u> doesn't create a single, large data structure in memory, but instead returns an iterable.
  - ↳ A generator uses <u>lazy evaluation</u>, meaning your code only calculates values when they're explicitly requested.
  - ↳ generators are optionally inside parentheses.
    - ↳ Ex

Python

```python
>>> sum(number * number for number in range(1_000_000_000))
333333332833333333500000000
```

Reading 5 continued

☆ How to use Generators and yield in Python

- Generator functions are a special kind of function that return a lazy iterator.
  - ↳ you can loop over these objects like lists, but iterators do not store their contents in memory.

- Ex reading in large .csv files!

```python
def csv_reader(file_name):
    for row in open(file_name, "r"):
        yield row
```

```python
csv_gen = (row for row in open(file_name))
```

- Using yield will result in a generator object
- Using return will result in the first line of the file only.

- Ex Generating infinite sequence and testing it in console!

```python
def infinite_sequence():
    num = 0
    while True:
        yield num
        num += 1
```

```python
>>> gen = infinite_sequence()
>>> next(gen)
0
>>> next(gen)
1
>>> next(gen)
2
>>> next(gen)
3
```

- Generator functions use yield instead of return
  - ↳ yield indicates where a value is sent back to the caller, but unlike return, you don't exit the function.
    - ↳ Instead the state of the function is always remembered.

• Generator comprehension vs list comprehension:

Python
```
>>> nums_squared_lc = [num**2 for num in range(5)]
>>> nums_squared_gc = (num**2 for num in range(5))
```

Python
```
>>> nums_squared_lc
[0, 1, 4, 9, 16]
>>> nums_squared_gc
<generator object <genexpr> at 0x107fbbc78>
```

Python
```
>>> import sys
>>> nums_squared_lc = [i ** 2 for i in range(10000)]
>>> sys.getsizeof(nums_squared_lc)
87624
>>> nums_squared_gc = (i ** 2 for i in range(10000))
>>> print(sys.getsizeof(nums_squared_gc))
120
```

list comprehension is usually faster than generator comprehension if memory is not an issue.

Understanding the Python Yield Statement

• When you call special methods on the generator, such as next(), the code within the function is executed up to yield.

↳ Then the execution is suspended, but the state of that function is saved

• Ex

Python
```
>>> def multi_yield():
...     yield_str = "This will print the first string"
...     yield yield_str
...     yield_str = "This will print the second string"
...     yield yield_str
...
>>> multi_obj = multi_yield()
>>> print(next(multi_obj))
This will print the first string
>>> print(next(multi_obj))
This will print the second string
>>> print(next(multi_obj))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

← because generators can also be exhausted like any other iterator.

Using Advanced Generator Methods:

i = (yield num) ← i takes the value that is yielded

Ex .send()

Python

```python
pal_gen = infinite_palindromes()
for i in pal_gen:
    digits = len(str(i))
    pal_gen.send(10 ** (digits))
```

← brings execution back into the generator and assigns 10** digits to i.

- This is a coroutine, or a generator function into which you can pass data.

Ex .throw(), allows you to throw errors!

Python

```python
1 pal_gen = infinite_palindromes()
2 for i in pal_gen:
3     print(i)
4     digits = len(str(i))
5     if digits == 5:
6         pal_gen.throw(ValueError("We don't like large palindromes"))
7     pal_gen.send(10 ** (digits))
```

Python

```
11
111
1111
10101
Traceback (most recent call last):
  File "advanced_gen.py", line 47, in <module>
    main()
  File "advanced_gen.py", line 41, in main
    pal_gen.throw(ValueError("We don't like large palindromes"))
  File "advanced_gen.py", line 26, in infinite_palindromes
    i = (yield num)
ValueError: We don't like large palindromes
```

Ex .close(), raises a StopIteration, allows you to stop the generator:

```python
pal_gen = infinite_palindromes()
for i in pal_gen:
    print(i)
    digits = len(str(i))
    if digits == 5:
        pal_gen.close()
    pal_gen.send(10 ** (digits))
```

```
11
111
1111
10101
Traceback (most recent call last):
  File "advanced_gen.py", line 46, in <module>
    main()
  File "advanced_gen.py", line 42, in main
    pal_gen.send(10 ** (digits))
StopIteration
```

Ex Building Data Pipeline with Generators:

Starting with this CSV file:

```csv
permalink,company,numEmps,category,city,state,fundedDate,raisedAmt,raisedCurre
digg,Digg,60,web,San Francisco,CA,1-Dec-06,8500000,USD,b
digg,Digg,60,web,San Francisco,CA,1-Oct-05,2800000,USD,a
facebook,Facebook,450,web,Palo Alto,CA,1-Sep-04,500000,USD,angel
facebook,Facebook,450,web,Palo Alto,CA,1-May-05,12700000,USD,a
photobucket,Photobucket,60,web,Palo Alto,CA,1-Mar-05,3000000,USD,a
```

Pipeline:

```python
file_name = "techcrunch.csv"
lines = (line for line in open(file_name))
list_line = (s.rstrip().split(",") for s in lines)
cols = next(list_line)
company_dicts = (dict(zip(cols, data)) for data in list_line)
funding = (
    int(company_dict["raisedAmt"])
    for company_dict in company_dicts
    if company_dict["round"] == "a"
)
total_series_a = sum(funding)
print(f"Total series A fundraising: ${total_series_a}")
```

Breakdown:

- **Line 2** reads in each line of the file.
- **Line 3** splits each line into values and puts the values into a list.
- **Line 4** uses `next()` to store the column names in a list.
- **Line 5** creates dictionaries and unites them with a `zip()` call:
    - **The keys** are the column names `cols` from line 4.
    - **The values** are the rows in list form, created in line 3.
- **Line 6** gets each company's series A funding amounts. It also filters out any other raised amount.
- **Line 11** begins the iteration process by calling `sum()` to get the total amount of series A funding found in the CSV.