Programming Techniques in Computational Linguistics II – FS24

Lecture 5: Functional Programming

Topics

- Advanced Programming with Functions
- Functional Programming
- Iterables, Iterators and Generators

Learning Objectives

- You know how to use higher order functions, anonymous functions and decorators.
- You can avoid side effects by using built in functions and list comprehensions.
- You can use the `__iter__` special method to make custom classes iterable.
- You know generator functions and their use cases.

Advanced Programming with Functions

Functions as Objects

Functions in Python are **objects**.

Functions in Python are **objects**.

In programming language theory, (first-class) objects are:

- Created at runtime
- Assigned to a variable or element in a datastructure
- Passed as an argument to a function
- Returned as the result of a function

In [1]:

```
def count_characters(s: str) -> int:
    '''Returns count of characters in s without whitespace'''
    return len(s) - s.count(' ')
```

In [1]:
```python
def count_characters(s: str) -> int:
    '''Returns count of characters in s without whitespace'''
    return len(s) - s.count(' ')
```

In [2]:
```python
count_characters("Hello World")
```

Out[2]:
```
10
```

In [1]:
```python
def count_characters(s: str) -> int:
    '''Returns count of characters in s without whitespace'''
    return len(s) - s.count(' ')
```

In [2]:
```python
count_characters("Hello World")
```

Out[2]:
```
10
```

In [3]:
```python
print(count_characters)
```

```
<function count_characters at 0x1060b45e0>
```

```
In [4]: cc = count_characters
```

```
In [5]: print(cc)
```

```
<function count_characters at 0x1060b45e0>
```

```
In [6]: print(count_characters)
```

```
<function count_characters at 0x1060b45e0>
```

```
In [7]: cc("Hello World")
```

```
Out[7]: 10
```

Higher-order Functions

A **higher-order function** is a function that takes a function as an argument or that returns a function.

Higher-order Functions

A **higher-order function** is a function that takes a function as an argument or that returns a function.

In [8]:
```python
fruit = ['strawberry', 'apple', 'cherry', 'banana']

sorted(fruit, key=len)
```

Out[8]:
```
['apple', 'cherry', 'banana', 'strawberry']
```

Custom Sort Keys

A sort key can be any function that takes one argument.

In [9]:
```
fruit = ['strawberry', 'apple', 'cherry', 'banana']

def reverse(word):
    return word[::-1]
```

In [10]:
```
reverse("strawberry")
```

Out[10]:
```
'yrrebwarts'
```

In [11]:
```
sorted(fruit, key=reverse)
```

Out[11]:
```
['banana', 'apple', 'strawberry', 'cherry']
```

Anynymous Functions

To use a higher-order function, sometimes it is convenient to create a small, one-off function.

The `lambda` keyword creates an **anonymous function** within a Python expression.

Anynymous Functions

To use a higher-order function, sometimes it is convenient to create a small, one-off function.

The `lambda` keyword creates an **anonymous function** within a Python expression.

In [12]:
```
fruit = ['strawberry', 'apple', 'cherry', 'banana']

sorted(fruit, key=lambda word: word[::-1])
```

Out[12]:
```
['banana', 'apple', 'strawberry', 'cherry']
```

```
In [13]: sorted(fruit, key=lambda word: -word.count('r'))

Out[13]: ['strawberry', 'cherry', 'apple', 'banana']
```

Example: Nested dictionaries

Example: Nested dictionaries

In [14]:
```
from collections import defaultdict

d = defaultdict(lambda: defaultdict(int))

d["SENT_1"]["NOUNS"] = 3
d["SENT_1"]["VERBS"] = 2
d["SENT_2"]["NOUNS"] = 2
```

In [15]:
```
d
```

Out[15]:
```
defaultdict(<function __main__.<lambda>()>,
            {'SENT_1': defaultdict(int, {'NOUNS': 3, 'VERBS': 2}),
             'SENT_2': defaultdict(int, {'NOUNS': 2})})
```

Callable Objects

The **call operator** `()` may be applied to other objects beyond standard functions. To determine whether an object is **callable**, use the `callable()` built-in function.

Callable Objects

The **call operator** `()` may be applied to other objects beyond standard functions. To determine whether an object is **callable**, use the `callable()` built-in function.

In [16]:
```
callable(print)
```

Out[16]:
```
True
```

Callable Objects

The **call operator** `()` may be applied to other objects beyond standard functions. To determine whether an object is **callable**, use the `callable()` built-in function.

In [16]:
```python
callable(print)
```

Out[16]:
```
True
```

In [17]:
```python
callable(lambda x: x.a)
```

Out[17]:
```
True
```

Callable Objects

The **call operator** `()` may be applied to other objects beyond standard functions. To determine whether an object is **callable**, use the `callable()` built-in function.

In [16]:
```
callable(print)
```
Out[16]:
```
True
```

In [17]:
```
callable(lambda x: x.a)
```
Out[17]:
```
True
```

In [18]:
```
callable(12)
```
Out[18]:
```
False
```

Types of callables

- **User-defined functions**. Created with `def` statements or `lambda` expressions.
- **Built-in functions**. A function implemented in C like `len` or `time.sleep`.
- **Methods**. Functions defined in the body of a class.
- **Built-in methods**. Methods implemented in C, like `dict.get`.

- **Classes**. When invoked, a class runs its `__new__` method to create an instance, then `__init__` to initialize it, and finally the instance is returned to the caller.
- **Class instances**. If a class defines a `__call__` method, then its instances may be invoked as functions.

In [19]:
```python
class Word:
    def __init__(self, text: str):
        self.text = text

    def __call__(self):
        print(self.text)

callable(Word)
```

Out[19]:
```
True
```

In [19]:
```python
class Word:
    def __init__(self, text: str):
        self.text = text

    def __call__(self):
        print(self.text)

callable(Word)
```

Out[19]:
```
True
```

In [20]:
```python
word = Word("Hello")
word()
```

```
Hello
```

Decorators

A **decorator** takes a callable as argument and adds behavior without explicitly modifying the callable.

Decorators from previous lectures:

```
@classmethod
def introduce(self):
    ...

@abstractmethod
def do_stuff(self):
    ...
```

Applying a decorator:

```python
@mydecorator
def myfunction():
    print('Running myfunction')
```

is equivalent to:

```python
def myfunction():
    print('Running myfunction')

myfunction = mydecorator(myfunction)
```

In [21]:

```python
def mydecorator(func: callable) -> callable:
    def inner():
        print("Before function call")
        func()
        print("After function call")
    return inner
```

In [22]:
```
def myfunction():
    print("Running myfunction")
```

In [23]:
```
decorated_function = mydecorator(myfunction)
decorated_function()
```

```
Before function call
Running myfunction
After function call
```

In [24]:
```
@mydecorator
def myfunction():
    print("Running myfunction")
```

In [25]:
```
myfunction()
```

```
Before function call
Running myfunction
After function call
```

In [1]:
```
import time
def timer(func: callable) -> callable:
    def inner(*args, **kwargs):
        start_time = time.perf_counter()
        f = func(*args, **kwargs)
        run_time = time.perf_counter() - start_time
        print(f"Finished {func.__name__}() in {run_time:.4f} secs")
        return f
    return inner
```

In [2]:
```
@timer
def myfunction2(n: int) -> int:
    for i in range(n):
        p = i*i-i
    print("Finished for-loop")
    return p
```

In [1]:
```python
import time
def timer(func: callable) -> callable:
    def inner(*args, **kwargs):
        start_time = time.perf_counter()
        f = func(*args, **kwargs)
        run_time = time.perf_counter() - start_time
        print(f"Finished {func.__name__}() in {run_time:.4f} secs")
        return f
    return inner
```

In [2]:
```python
@timer
def myfunction2(n: int) -> int:
    for i in range(n):
        p = i*i-i
    print("Finished for-loop")
    return p
```

In [4]:
```python
myfunction2(100000000)
```

```
Finished for-loop
Finished myfunction2() in 3.5708 secs
```

Out[4]:
```
9999999700000002
```

Klicker Quiz

https://pwa.klicker.uzh.ch/join/lfische

Functional Programming

What is Functional Programming?

Theoretically:

- avoiding side effects
- avoiding mutable data
- preferring expressions over statements
- using pure functions

Practically in Python: a frequent use of:

- the `lambda` operator
- `zip()` and `enumerate()`
- `map()` and `filter()`
- list comprehensions

Pure Functions

A function is considered pure if

- it has no side effects
    - it does not modify or interact with any data outside of its scope
    - it does not print anything
- always provides the same output for the same inputs

Example: Side Effects

In [5]:
```
from typing import Iterable

tokens = ['This', 'is', 'a', 'Test', '.']

def lowercase1(tokens: Iterable[str]) -> Iterable[str]:
    for i in range(len(tokens)):
        tokens[i] = tokens[i].lower()
    return tokens
```

Example: Side Effects

In [5]:

```
from typing import Iterable

tokens = ['This', 'is', 'a', 'Test', '.']

def lowercase1(tokens: Iterable[str]) -> Iterable[str]:
    for i in range(len(tokens)):
        tokens[i] = tokens[i].lower()
    return tokens
```

In [6]:

```
lowercase1(tokens)
```

Out[6]:

```
['this', 'is', 'a', 'test', '.']
```

Example: Side Effects

In [5]:
```
from typing import Iterable

tokens = ['This', 'is', 'a', 'Test', '.']

def lowercase1(tokens: Iterable[str]) -> Iterable[str]:
    for i in range(len(tokens)):
        tokens[i] = tokens[i].lower()
    return tokens
```

In [6]:
```
lowercase1(tokens)
```

Out[6]:
```
['this', 'is', 'a', 'test', '.']
```

In [7]:
```
print(tokens)
```

```
['this', 'is', 'a', 'test', '.']
```

Without side effects:

In [8]:

```python
tokens = ['This', 'is', 'a', 'Test', '.']

def lowercase2(tokens: Iterable[str]) -> Iterable[str]:
    new_tokens = []
    for token in tokens:
        new_tokens.append(token.lower())
    return new_tokens
```

Without side effects:

In [8]:

```
tokens = ['This', 'is', 'a', 'Test', '.']

def lowercase2(tokens: Iterable[str]) -> Iterable[str]:
    new_tokens = []
    for token in tokens:
        new_tokens.append(token.lower())
    return new_tokens
```

In [9]:

```
print(lowercase2(tokens))
```

```
['this', 'is', 'a', 'test', '.']
```

Without side effects:

In [8]:
```
tokens = ['This', 'is', 'a', 'Test', '.']

def lowercase2(tokens: Iterable[str]) -> Iterable[str]:
    new_tokens = []
    for token in tokens:
        new_tokens.append(token.lower())
    return new_tokens
```

In [9]:
```
print(lowercase2(tokens))
```

```
['this', 'is', 'a', 'test', '.']
```

In [10]:
```
print(tokens)
```

```
['This', 'is', 'a', 'Test', '.']
```

More elegant solution:

```
In [11]: tokens = ['This', 'is', 'a', 'Test', '.']

         def lowercase3(tokens: Iterable[str]) -> Iterable[str]:
             return [token.lower() for token in tokens]

         print(lowercase3(tokens))
         print(tokens)
```

```
['this', 'is', 'a', 'test', '.']
['This', 'is', 'a', 'Test', '.']
```

Klicker-Quiz: "Is this a pure function?"

https://pwa.klicker.uzh.ch/join/lfische

Built-in Function: `enumerate()`

`enumerate()` generates index-element pairs from an iterable

```
In [12]: l = ['Fred', 'fed', 'Ted', 'bread', 'and', 'Ted', 'fed', 'Fred', 'bread']
         >>> list(enumerate(l))

Out[12]: [(0, 'Fred'),
          (1, 'fed'),
          (2, 'Ted'),
          (3, 'bread'),
          (4, 'and'),
          (5, 'Ted'),
          (6, 'fed'),
          (7, 'Fred'),
          (8, 'bread')]
```

This function is useful when processing a file line by line:

```
with open('lyrics.txt', 'r') as f:
    for i, line in enumerate(f):
        print(i, line)
```

Built-in Function: `zip()`

`zip(iterA, iterB, ...)` takes one element from each iterable and returns them in a tuple:

In [19]:
```python
l1 = ['Italy', 'France', 'Switzerland']
l2 = ['Rome', 'Bern']

for e1, e2 in zip(l1, l2, strict=True):
    print(e1, e2)
```

```
Italy Rome
France Bern

---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[19], line 4
      1 l1 = ['Italy', 'France', 'Switzerland']
      2 l2 = ['Rome', 'Bern']
----> 4 for e1, e2 in zip(l1, l2, strict=True):
      5     print(e1, e2)

ValueError: zip() argument 2 is shorter than argument 1
```

`dict` from `zip()`

The `dict()` constructor can accept any iterator that returns a finite stream of (key, value) tuples:

In [16]:
```python
l1 = ['Italy', 'France', 'Switzerland']
l2 = ['Rome', 'Paris', 'Bern']

d = dict(zip(l1, l2))
print(d)
```

```
{'Italy': 'Rome', 'France': 'Paris', 'Switzerland': 'Bern'}
```

Built-in Function: `filter()`

`filter(predicate, iter)` returns an iterator over all the sequence elements that meet a certain condition. Its predicate is a function that returns the truth value of some condition.

In [20]:
```
# isupper() returns True if all the characters
# are in upper case, otherwise False.
l = ['Hello', 'WORLD', '!!!']
list(filter(str.isupper, l))
```

Out[20]:
```
['WORLD']
```

You can also use an anonymous function as predicate:

In [21]:
```
l = ['Hello', 'WORLD', '!!!']

for x in (filter(lambda s: s != '!!!', l)):
    print(x)
```

```
Hello
WORLD
```

Built-in Function: `map()`

The function passed into `map(func, iter)` is applied to each item in the iterable.

```
In [22]: list(map(lambda x: x.upper(), ['foo', 'bar']))

Out[22]: ['FOO', 'BAR']
```

In [23]:
```
list(map(str, ['hello', 123, {'France': 'Paris'}]))
```

Out[23]:
```
['hello', '123', "{'France': 'Paris'}"]
```

List Comprehensions

Both `filter()` and `map()` can be replaced with list comprehensions:

In [24]:
```
l = ['Hello', 'WORLD', '!!!']
list(filter(str.isupper, l))
```

Out[24]:
```
['WORLD']
```

List Comprehensions

Both `filter()` and `map()` can be replaced with list comprehensions:

In [24]:
```
l = ['Hello', 'WORLD', '!!!']
list(filter(str.isupper, l))
```

Out[24]:
```
['WORLD']
```

In [25]:
```
[word for word in l if word.isupper()]
```

Out[25]:
```
['WORLD']
```

In [26]:

```python
list(map(lambda x: x.upper(), ['foo', 'bar']))
```

Out[26]:

```
['FOO', 'BAR']
```

In [26]:
```python
list(map(lambda x: x.upper(), ['foo', 'bar']))
```

Out[26]:
```
['FOO', 'BAR']
```

In [27]:
```python
[word.upper() for word in ['foo', 'bar']]
```

Out[27]:
```
['FOO', 'BAR']
```

Dictionary Comprehensions

In [28]:

```python
DIAL_CODES = [
(86, 'China'),
(91, 'India'),
(1, 'United States'),
(41, 'Switzerland'),
(880, 'Bangladesh'),
]
```

In [28]:
```
DIAL_CODES = [
(86, 'China'),
(91, 'India'),
(1, 'United States'),
(41, 'Switzerland'),
(880, 'Bangladesh'),
]
```

In [29]:
```
d = {country: code
     for code, country in DIAL_CODES}
print(d)
```

```
{'China': 86, 'India': 91, 'United States': 1, 'Switzerland': 41, 'Bangladesh': 880}
```

In [30]:
```
d
```

Out[30]:
```
{'China': 86,
 'India': 91,
 'United States': 1,
 'Switzerland': 41,
 'Bangladesh': 880}
```

In [31]:
```
{country.upper(): code
     for country, code in d.items()
     if code < 42}
```

Out[31]:
```
{'UNITED STATES': 1, 'SWITZERLAND': 41}
```

Exercise

Answer three questions given the sentence:

In [32]:
```
s = 'Time flies like an arrow , fruit flies like a banana .'
tokens = s.split()
```

1. Use `map()` to reverse each token individually. Then convert the expression to a **list comprehension**.
2. Use `filter()` to filter out commas and periods. Then convert the expression to a **list comprehension**.

3. Which syntax do you prefer?

In [41]:
```
map(lambda x: x[::-1], tokens)
```

Out[41]:
```
<map at 0x10aa1ac80>
```

In [42]:
```
(word[::-1] for word in tokens)
```

Out[42]:
```
<generator object <genexpr> at 0x10b059f50>
```

In [37]:
```
list(filter(str.isalpha, tokens))
```

Built-in Functions: `min()`, `max()`

In [43]:
```
min(['a', 'b', 'c'])
```
Out[43]:
```
'a'
```

In [44]:
```
max(['a', 'b', 'c'])
```
Out[44]:
```
'c'
```

Built-in Functions: `min()`, `max()`

In [43]:
```
min(['a', 'b', 'c'])
```
Out[43]:
```
'a'
```

In [44]:
```
max(['a', 'b', 'c'])
```
Out[44]:
```
'c'
```

Functions like `min()` or `max()` are called *reducing*, *folding* or *accumulating* functions.

`sum()` and `str.join()`

In [45]:
```
s = 'Time flies like an arrow , fruit flies like a banana .'
tokens = s.split()
```

`sum()` and `str.join()`

In [45]:
```
s = 'Time flies like an arrow , fruit flies like a banana .'
tokens = s.split()
```

In [46]:
```
sum([len(token) for token in tokens])
```

Out[46]:
```
43
```

`sum()` and `str.join()`

In [45]:
```
s = 'Time flies like an arrow , fruit flies like a banana .'
tokens = s.split()
```

In [46]:
```
sum([len(token) for token in tokens])
```

Out[46]:
```
43
```

In [47]:
```
'\n'.join(tokens)
```

Out[47]:
```
'Time\nflies\nlike\nan\narrow\n,\nfruit\nflies\nlike\na\nbanana\n.'
```

Built-in Functions: `any()` and `all()`

`any()` returns `True` if any element in the iterable is a true value.

`all()` returns `True` if all of the elements are true values.

In [48]:
```
def may_be_german(text: str) -> bool:
    return any([char.lower() in 'äöü' for char in text])
```

In [48]:
```
def may_be_german(text: str) -> bool:
    return any([char.lower() in 'äöü' for char in text])
```

In [49]:
```
may_be_german("aou")
```

Out[49]:
```
False
```

In [48]:
```
def may_be_german(text: str) -> bool:
    return any([char.lower() in 'äöü' for char in text])
```

In [49]:
```
may_be_german("aou")
```

Out[49]:
```
False
```

In [50]:
```
may_be_german("äou")
```

Out[50]:
```
True
```

In [48]:
```
def may_be_german(text: str) -> bool:
    return any([char.lower() in 'äöü' for char in text])
```

In [49]:
```
may_be_german("aou")
```

Out[49]:
```
False
```

In [50]:
```
may_be_german("äou")
```

Out[50]:
```
True
```

In [51]:
```
may_be_german("äöü")
```

Out[51]:
```
True
```

In [52]:

```python
def may_be_german(text: str) -> bool:
    return all([char.lower() in 'äöü' for char in text])
```

In [52]:
```
def may_be_german(text: str) -> bool:
    return all([char.lower() in 'äöü' for char in text])
```

In [53]:
```
may_be_german("aou")
```

Out[53]:
```
False
```

In [52]:
```python
def may_be_german(text: str) -> bool:
    return all([char.lower() in 'äöü' for char in text])
```

In [53]:
```python
may_be_german("aou")
```

Out[53]:
```
False
```

In [55]:
```python
may_be_german("äou")
```

Out[55]:
```
False
```

In [52]:
```python
def may_be_german(text: str) -> bool:
    return all([char.lower() in 'äöü' for char in text])
```

In [53]:
```python
may_be_german("aou")
```

Out[53]:
```
False
```

In [55]:
```python
may_be_german("äou")
```

Out[55]:
```
False
```

In [56]:
```python
may_be_german("äöü")
```

Out[56]:
```
True
```

Iterables, Iterators and Generators

Iterators

In [57]:
```
fruits = ['strawberry', 'apple', 'cherry']
fruits_it = iter(fruits)
fruits_it
```

Out[57]:
```
<list_iterator at 0x10aa1b790>
```

In [58]:
```
fruits_it.__next__()
```

Out[58]:
```
'strawberry'
```

In [61]:
```
next(fruits_it)
```

```
---------------------------------------------------------------------------
StopIteration                             Traceback (most recent call last)
Cell In[61], line 1
----> 1 next(fruits_it)

StopIteration: 
```

Calling iter() on objects of different types:

In [62]:
```
iter({'n': .58, 'v': .37, 'a': .05})
```

Out[62]:
```
<dict_keyiterator at 0x10a0dc4a0>
```

Calling iter() on objects of different types:

In [62]:
```
iter({'n': .58, 'v': .37, 'a': .05})
```
Out[62]:
```
<dict_keyiterator at 0x10a0dc4a0>
```

In [63]:
```
iter('TACTTAATAAAAATAAAGCATATTACATTATTCTCACATGGACTAT')
```
Out[63]:
```
<str_iterator at 0x10aa1afb0>
```

In [64]:
```python
iter(open('very_large_file.txt'))
```

Out[64]:
```
<_io.TextIOWrapper name='very_large_file.txt' mode='r' encoding='UTF-8'>
```

In [65]:
```python
iter(5)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[65], line 1
----> 1 iter(5)

TypeError: 'int' object is not iterable
```

The `iter()` Function

Whenever the interpreter needs to **iterate** over an object x, it automatically calls `iter(x)`.

The `iter()` built-in function:

The `iter()` built-in function:

1. Checks whether the object implements `__iter__()`, and calls that to obtain an iterator.

The `iter()` built-in function:

1. Checks whether the object implements `__iter__()`, and calls that to obtain an iterator.
2. If `__iter__()` is not implemented but `__getitem__()` is implemented, Python creates an iterator that attempts to fetch items in order, starting from index 0.

The `iter()` built-in function:

1. Checks whether the object implements `__iter__()`, and calls that to obtain an iterator.
2. If `__iter__()` is not implemented but `__getitem__()` is implemented, Python creates an iterator that attempts to fetch items in order, starting from index 0.
3. If that fails, Python raises `TypeError`, usually saying "object is not iterable".

Iterable Objects

In [66]:

```
class Sentence:

    def __init__(self, text: str):
        self.text = text
        self.tokens = text.split()

    def __getitem__(self, index):
        return self.tokens[index]
```

In [67]:

```python
s = Sentence('Hello , World !')
```

In [68]:

```python
for token in s:
    print(token)
```

```
Hello
,
World
!
```

Generators

In [ ]:
```
class Sentence:

    def __init__(self, text: str):
        self.text = text
        self.tokens = text.split()

    def __iter__(self):
        for token in self.tokens:
            yield token
```

In [ ]:

```python
class Sentence:

    def __init__(self, text: str):
        self.text = text
        self.tokens = text.split()

    def __iter__(self):
        for token in self.tokens:
            yield token
```

In [69]:

```python
s = Sentence('Hello , World !')
for token in s:
    print(token)
```

```
Hello
,
World
!
```

Generators are itertators.

`yield` is a keyword used to create a generator function.

It is similar to `return`, but where `return` will terminate the function, `yield` will only pause it.

`yield` is highly memory efficient, as the function is only executed when caller iterates over the object. But it must be handled properly.

Lazy Iteration through Generators

In [70]:

```python
class Sentence:

    def __init__(self, text: str):
        self.text = text

    def __iter__(self):
        while ' ' in self.text:
            token, self.text = self.text.split(maxsplit=1)
            yield token
        yield self.text
```

```
In [70]: class Sentence:

             def __init__(self, text: str):
                 self.text = text

             def __iter__(self):
                 while ' ' in self.text:
                     token, self.text = self.text.split(maxsplit=1)
                     yield token
                 yield self.text
```

```
In [71]: s = Sentence('Hello , World !')
         for token in s:
             print(token)
```

```
Hello
,
World
!
```

Generator Expressions

Generator expressions are an alternative to list comprehensions in two cases:

- When the iterator returns an infinite stream (e.g. prime numbers)
- When the iterator handles a large amount of data.

Generator Expressions

Generator expressions are an alternative to list comprehensions in two cases:

- When the iterator returns an infinite stream (e.g. prime numbers)
- When the iterator handles a large amount of data.

Generator expressions are surrounded by parentheses (`()`) while list comprehensions are surrounded by square brackets (`[]`).

In [87]:
```python
infile = open("very_large_file.txt")
generator = (line.strip().upper() for line in infile if line != "\n")
```

Read and uppercase lines one by one:

In [88]:
```python
next(generator)
```

Out[88]:
```
'THE PROJECT GUTENBERG EBOOK OF ULYSSES, BY JAMES JOYCE'
```

In [89]:
```python
from typing import Iterable

def fibonacci() -> Iterable[int]:
    x, y = 0, 1
    while True:
        yield y
        x, y = y, x + y
```

In [90]:
```python
generator = fibonacci()
```

In [105]:
```python
next(generator)
```

Out[105]:
```
610
```

Klicker Quiz

https://pwa.klicker.uzh.ch/join/lfische

Take-home messages

- Higher-order functions have a function as an argument or return value.
- Decorators are higher-order functions used to modify the behaviour of a function.
- The `lambda` keyword creates anonymous functions.
- Functional programming in python means using pure functions, list comprehensions and built in functions like `map`, `filter`, `any`, `zip`, etc.
- Iterables are objects that can be iterated over, Iterators are objects created to handle the iteration.
- The `yield` keyword creates a generator function, a memory efficient way to iterate over a large dataset.