04. More on functions and iterables

October 15, 2025

0.1 1. Function arguments recap

Parameters define the inputs that a function can accept. When defining a function, we specify its parameters in parentheses, and they can be of two types: - required - optional

[1]:
```python
def my_func(first, second=None):
    pass
```

In the function above, `first` is a required parameter and `second` is an optional one, also known as default parameter, because it has a default value which will be used if an actual value is not provided at function call.

Default parameters always follow required ones in a function definition.

When calling the function, arguments can be specified as: - positional arguments - keyword arguments

[2]:
```python
my_func(10, 20)  # call with positional arguments
my_func(10)
```

[3]:
```python
my_func(first=10, second=20)  # call with keyword arguments
my_func(second=20, first=10)  # mapping is done by name, so order is not␣
 ↪important
my_func(first=10)
```

[4]:
```python
my_func(10, second=20)  # call with a mix of positional & keyword arguments
```

0.2 2. Functions with variable-length arguments

Variable-length arguments allow functions to accept an arbitrary number of arguments, providing greater flexibility when defining functions that need to handle varying numbers of inputs. Python supports two types of variable-length arguments: `*args` and `**kwargs`.

0.2.1 `*args`

The `*args` syntax is used to pass a variable number of non-keyword arguments to a function. Inside the function, `args` is treated as a tuple containing all the additional arguments passed to the function.

[5]:
```python
def example_function(*args):
    for arg in args:
```

```
        print(arg)
```

[6]:
```
example_function()
```

[7]:
```
example_function(1)
```

```
1
```

[8]:
```
example_function(1, 2, 3)
```

```
1
2
3
```

0.2.2 **kwargs

The **kwargs syntax allows a function to accept a variable number of keyword arguments. Inside the function, kwargs is treated as a dictionary where the keys are the argument names and the values are the corresponding arguments passed.

[9]:
```
def example_function(**kwargs):
    for key, value in kwargs.items():
        print(f"{key} = {value}")
```

[10]:
```
example_function(a=1, b=2, c=3)
```

```
a = 1
b = 2
c = 3
```

0.2.3 Combining *args and **kwargs

You can combine *args and **kwargs in a single function definition to accept both positional and keyword arguments. When doing so, *args must appear before **kwargs.

[3]:
```
def example_function(*args, **kwargs):
    for arg in args:
        print(arg)
    for key, value in kwargs.items():
        print(f"{key} = {value}")
```

[4]:
```
example_function(1, 2, a=3, b=4)
```

```
1
2
a = 3
b = 4
```

0.2.4 Exercises 2

1. Experiment with `str.format()` method. Call it with positional arguments and with keyword arguments.
2. Write a function called `calculate_average` that takes in any number of arguments and calculates their average. The function should accept integers and floats. If no arguments are provided, the function should return 0. If arguments are provided, the function should calculate their average and return it rounded to two decimal places.
3. Write a function `process_order` that accepts keyword arguments representing items and their quantities. The function should calculate the total number of items ordered and print a summary.

   E.g. `python process_order(apples=5, bananas=3, oranges=2)`

   should output: `You have ordered: apples: 5 bananas: 3 oranges: 2 Total items: 10`

0.3 3. Positional-only and keyword-only arguments

`*args` and `**kwargs` can also be combined with regular arguments:

[6]:
```python
def func(x, y, *args):
    print("Positional arguments:", x, y)
    print("Varargs:", args)
```

For the example above, valid calls will have at least two positional arguments:

[8]:
```python
func(1, 2, 3, 4, 5)
```

```
Positional arguments: 1 2
Varargs: (3, 4, 5)
```

We can define additional arguments after the varargs argument. Since the varargs argument *swallows* all positional arguments, it's impossible to fill these additional arguments with positional values.

[9]:
```python
def func(x, y, *args, flag):
    print("Positional arguments:", x, y)
    print("Varargs:", args)
    print("Flag:", flag)
```

In this case, the `flag` argument therefore needs to be provided by keyword. It is a **keyword-only** required argument.

[11]:
```python
func(1, 2, 3, 4, True)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[11], line 1
----> 1 func(1, 2, 3, 4, True)
```

```
TypeError: func() missing 1 required keyword-only argument: 'flag'
```

[12]:
```
func(1, 2, 3, 4, flag=True)
```

```
Positional arguments: 1 2
Varargs: (3, 4)
Flag: True
```

We might not want to use a varargs argument though. If our function only expects a limited amount of arguments, we can keep the keyword-only behavior while dropping the varargs argument.

[18]:
```
def func(x, y, *, flag):
    print("Positional arguments:", x, y)
    print("Flag:", flag)
```

[19]:
```
func(1, 2, flag=True)
```

```
Positional arguments: 1 2
Flag: True
```

Using keyword-only arguments prevents us from calling functions with all positional arguments. But it still allows calling the function with all keyword arguments:

[20]:
```
func(flag=True, x=1, y=2)
```

```
Positional arguments: 1 2
Flag: True
```

While this is less of a problem, it is still less clear than a combination of both positional and keyword arguments. Positional arguments implicitly provide additional clarity on how the function behaves. They rely on the natural logic of order, which can be understood at a glance. That's why PEP 570 introduced **positional-only arguments**, whose behavior is equivalent but opposite to keyword-only arguments.

[21]:
```
def func(x, y, /, *, flag):
    print("Positional-only arguments:", x, y)
    print("Flag:", flag)
```

Here all arguments left of the / are positional-only arguments.

[22]:
```
func(flag=True, x=1, y=2)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[22], line 1
----> 1 func(flag=True, x=1, y=2)
```

```
TypeError: func() got some positional-only arguments passed as keyword arguments:
↪ 'x, y'
```

[23]:
```
func(1, 2, flag=True)
```

```
Positional-only arguments: 1 2
Flag: True
```

0.3.1 Exercises 3

1. Write a function that takes any number of strings and an integer `min_length` as parameters. `min_length` should be an optional keyword-only parameter. Return the list of strings longer than `min_length`. By default (when `min_length` not given), it should return a list containing all words.
2. Write a function `calculate_total` that accepts positional-only arguments for `price` and `quantity`, and a keyword-only argument `discount` with a default value of 0. The function should return the total price after applying the discount.
3. Find examples of functions (or methods) from Python Standard Library that receive keyword-only and positional-only arguments and try them out.

0.4 4. Variable unpacking

Variable unpacking refers to the process of breaking down data structures, such as lists, tuples, or dictionaries, into individual variables. This feature allows you to assign values from a collection to variables in a concise and readable manner.

0.4.1 Unpacking Lists and Tuples

You can unpack elements of a list or tuple directly into variables.

[24]:
```
my_list = [1, 2, 3]
a, b, c = my_list
print(a, b, c)
```

```
1 2 3
```

[25]:
```
my_tuple = (4, 5, 6)
x, y, z = my_tuple
print(x, y, z)
```

```
4 5 6
```

Unpacking with `*` The `*` operator can be used to unpack multiple elements into a single variable.

[12]:
```
numbers = [1, 2, 3, 3, 2, 4]
first, *middle, last = numbers
```

[13]:
```
first
```

[13]: 1

```
[14]: middle
```

[14]: [2, 3, 3, 2]

```
[15]: last
```

[15]: 4

0.4.2 Unpacking Dictionaries

You can unpack dictionaries using the `**` operator.

```
[26]: def print_info(name, age, city):
          print(f"Name: {name}, Age: {age}, City: {city}")
```

```
[27]: my_dict = {"name": "Alice", "age": 30, "city": "New York"}
      print_info(**my_dict)
      # print_info(name="Alice", age=30, ...)
```

```
Name: Alice, Age: 30, City: New York
```

0.4.3 Nested Unpacking

You can unpack nested structures directly.

```
[29]: nested_list = [1, [2, 3], 4]
      a, (b, c), d = nested_list
      print(a, b, c, d)
```

```
1 2 3 4
```

0.4.4 Ignoring Values During Unpacking

Use the `_` (underscore) variable to ignore certain values.

```
[33]: data = (10, 20, 30, 40)
      _, b, _, d = data
      print(b, d)  # Output: 20 40
```

```
20 40
```

0.4.5 Exercises 4

1. Write a function that takes two numbers as arguments and returns their sum, difference, product, and quotient. Call the function and assign the result to 4 different variables.
2. Using `*` unpacking and `range`, print the numbers 1 to 20, separated by commas. You will have to provide an argument for `print` function's `sep` parameter for this exercise.
3. Modify your code from the previous exercise so that each number prints on a different line. You can only use a single `print` call.

4. Print a sentence using the following dictionary, the `str.format` method and `**` unpacking: `python country = { "name": "Romania", "population": "19 million", "capital": "Bucharest", "currency": "RON" }` E.g. > Romania has a population of 19 million people. The capital is Bucharest and uses RON as currency.

0.5 5. Comprehensions

Comprehensions in Python provide us with a short and concise way to construct new iterables (such as lists, set, dictionary) using iterables which have been already defined. Python supports the following 4 types of comprehensions:

- List Comprehensions
- Dictionary Comprehensions
- Set Comprehensions
- Generator Comprehensions (discussed in a following section)

0.5.1 5.1. List comprehensions

`[expression_containing(x) for x in iterable]`

[34]:
```
[x ** 3 for x in range(5)]
```

[34]:
```
[0, 1, 8, 27, 64]
```

[35]:
```
[s.capitalize() for s in ('sun', 'moon', 'earth')]
```

[35]:
```
['Sun', 'Moon', 'Earth']
```

[36]:
```
[[] for _ in range(10)]
```

[36]:
```
[[], [], [], [], [], [], [], [], [], []]
```

`[expression_containing(x) for x in iterable if condition(x)]`

[37]:
```
[x ** 2 for x in range(20) if x % 3 == 0 and x % 2 != 0]
```

[37]:
```
[9, 81, 225]
```

0.5.2 Exercises 5.1

1. Write a list comprehension that creates a list of numbers from 1 to 20 that are divisible by 3.
2. Write a list comprehension that transforms all strings in a list by making them all lowercase and by replacing `a`s with `*`s.
3. Write a list comprehension that creates a list of all the words in a given string that have more than 3 letters.

0.5.3 5.2. Dictionary comprehensions

They are very similar to list comprehensions, but should build a key-value mapping.

[38]:
```python
{name: len(name) for name in ['Jane', 'Ann', 'George']}
```

[38]:
```
{'Jane': 4, 'Ann': 3, 'George': 6}
```

[39]:
```python
{x[0]: x[1] for x in [(1, 'one'), (2, 'two'), (3, 'three')] if x[0] % 2 == 1}
```

[39]:
```
{1: 'one', 3: 'three'}
```

[40]:
```python
{k: v for k, v in [(1, 'one'), (2, 'two'), (3, 'three')] if k % 2 == 1}
```

[40]:
```
{1: 'one', 3: 'three'}
```

0.5.4 Exercises 5.2

1. Create a dict `{"a": 97, "b": 98, ... }` using comprehension. Use `ord` built-in to obtain ASCII code and `string.ascii_lowercase` to get all letters.
2. Using the dictionary generated above, create another one where you swap keys and values.
3. Filter the above dictionary to contain only even keys.

0.5.5 5.3. Set comprehensions

Again, similar to the two above, use curly brackets, but they are flat.

[41]:
```python
{x*2 for x in [1, 3, 1, 2, 4, 1, 4, 3, 3]}
```

[41]:
```
{2, 4, 6, 8}
```

[42]:
```python
{s for s in 'set comprehension 101' if s.isalpha()}
```

[42]:
```
{'c', 'e', 'h', 'i', 'm', 'n', 'o', 'p', 'r', 's', 't'}
```

0.5.6 Exercises 5.3

1. Write a set comprehension to get all lowercase words in a text.

0.5.7 5.4. Nested comprehensions

Nested comprehensions are a concise way to generate complex lists, sets, or dictionaries by using comprehensions within comprehensions. They allow for the creation of nested structures in a more readable and compact form compared to traditional loops.

Creating a nested structure A basic nested list comprehension involves creating a list of lists or other nested structures.

[43]:
```python
[[i * j for j in range(1, 6)] for i in range(1, 6)]  # multiplication table
```

[43]:
```
[[1, 2, 3, 4, 5],
 [2, 4, 6, 8, 10],
 [3, 6, 9, 12, 15],
 [4, 8, 12, 16, 20],
```

```
[5, 10, 15, 20, 25]]
```

Flattening a nested structure You can also use nested comprehensions to flatten a list of lists into a single list:

[17]:
```
nested_list = [[1, 2, 3], [4, 5], [6, 7, 8, 9]]
```

[18]:
```
flattened_list = [item*2 for sublist in nested_list for item in sublist if item␣
↪% 2 == 0]
```

[19]:
```
flattened_list
```

[19]:
```
[4, 8, 12, 16]
```

0.5.8 Exercises 5.4

1. Write a nested set comprehension to generate a set of all unique pairs `(i, j)` where `i` is from the list `[1, 2, 3]` and `j` is from the list `[1, 1, 2]`.
2. Write a nested set comprehension to flatten a nested list `[[1, 2, 3, 4], [4, 5, 6, 7], [6, 7, 8, 9]] -> {1, 2, 3, 4, 5, 6, 7, 8, 9}`.
3. Write a nested dictionary comprehension to create a dictionary where the keys are words and the values are dictionaries of occurrences (keys -> characters, values -> number of occurrences in word), starting from a sentence. E.g. `python     text = "hello everyone" # should produce     occurrences_dict = {         'hello': {'h': 1, 'e': 1, 'l': 2, 'o': 1},         'everyone': {'e': 3, 'v': 1, 'r': 1, 'y': 1, 'o': 1, 'n': 1}     }`
4. Write a nested list comprehension to transpose a 3x3 matrix (switch its rows and columns).

0.6 6. Iterators

In Python, an **iterable** is anything that you can iterate over. **Iterators** are lazy single-use iterables:

- they are "lazy", because they have the ability to only compute items as you loop over them
- they are "single-use", because once you've consumed an item from a iterator, you cannot go back to it; after looping over the iterator, it is exhausted

You can get an iterator from any iterable, by using the `iter()` function:

[47]:
```
iter([1, 2])
```

[47]:
```
<list_iterator at 0x107aba1d0>
```

You can get the next item in an iterator by using the `next()` function:

[48]:
```
my_iterator = iter('hi')
next(my_iterator)
```

[48]:
```
'h'
```

[49]:
```
next(my_iterator)
```

[49]: 'i'

StopIteration is raised when there are no more items in the iterator:

[50]:
```
next(my_iterator)
```

```
---------------------------------------------------------------------------
StopIteration                             Traceback (most recent call last)
Cell In[50], line 1
----> 1 next(my_iterator)

StopIteration:
```

All iterators are also iterables, meaning you can get an iterator from an iterator (it'll give you itself back):

[51]:
```
my_iterator = iter([1, 2])
other_iterator = iter(my_iterator)
my_iterator == other_iterator
```

[51]: True

You can iterate on iterators:

[52]:
```
for item in my_iterator:
    print(item)
```

```
1
2
```

Iterators are stateful, meaning once you've consumed an item from an iterator, it's gone. After you've looped over an iterator once, it'll be empty if you try to loop over it again:

[53]:
```
items_left = []
for item in my_iterator:
    items_left.append(item)
print(items_left)
```

```
[]
```

0.7 7. Generators

Python generators are a simple way of creating iterators. Simply speaking, a generator is a function that returns an object (iterator) which we can iterate over (one value at a time). Read more on differences between iterables, iterators and generators here.

0.7.1 7.1. Generator functions

Generator functions use `yield` keyword instead of `return`. The difference is that while a `return` statement terminates a function entirely, `yield` statement pauses the function saving all its states and later continues from there on successive calls.

[54]:
```
def generator_func():
    yield 1
    yield 2
gen_obj = generator_func()
gen_obj
```

[54]:
```
<generator object generator_func at 0x108070f60>
```

The resulting generator object is an iterator. We can get items from it using `next()`:

[55]:
```
while True:
    try:
        print(next(gen_obj))
    except StopIteration:
        print('Generator exhausted.')
        break
```

```
1
2
Generator exhausted.
```

Normally, generator functions are implemented with a loop having a suitable terminating condition.

[34]:
```
def squares(start, stop):
    for i in range(start, stop):
        yield i ** 2
```

[35]:
```
for num in squares(100, 105):
    print(num)
```

```
10000
10201
10404
10609
10816
```

0.7.2 7.2. Generator expressions

Simple generators can be easily created on the fly using generator expressions. Generator expressions look very similar to list comprehension, but they use parantheses instead of square brackets.

They have lazy execution (producing items only when asked for). For this reason, a generator expression is much more memory efficient than an equivalent list comprehension.

[58]:
```
squares_generator = (x ** 2 for x in range(100, 105))
print(squares_generator)
for num in squares_generator:
    print(num)
```

```
<generator object <genexpr> at 0x108110860>
10000
10201
10404
10609
10816
```

0.7.3 Exercises 7

1. Create a generator function that receives a parameter `max_nr` and yields a random number between `1` and `max_nr`, indefinitely. From outside, iterate it in a loop that stops after 10 cycles.
2. Write a generator function that yields unique elements from an iterable received as parameter.
3. Write a generator function that takes a path and a file extension as positional-only arguments (both as strings), and a boolean `recursive` as keyword-only argument. The function will yield all files with the `extension` extension in `path`; if `recursive` is true, if will also search inside subdirectories of `path`. Use `os.walk` or `glob.iglob`.

0.8 8. Anonymous functions (`lambda`)

Python lambdas are short, anonymous functions, subject to a more restrictive but more concise syntax than regular Python functions. They are throw away functions, one purpose only, used mainly as parameters to functions that expect callables.

Syntax: `lambda arguments: expression` A lambda function can have any number of arguments but can have only one expression. It cannot contain any statements and it returns a function object.

[59]:
```
lambda x: x + 2
```

[59]:
```
<function __main__.<lambda>(x)>
```

[60]:
```
(lambda x: x ** 2)(15)
```

[60]:
```
225
```

[61]:
```
(lambda x, y: x.index(y))("say something", "some")
```

[61]:
```
4
```

0.9 9. Built-in functions (`filter`, `map`, `enumerate`, `sorted`, `zip`)

These are some of the most important built-in functions that receive iterables as parameters and produce iterables.

`filter(function, iterable)` Construct an iterator from those elements of iterable for which function returns true.

[62]:
```python
for x in filter(len, [(), [], (0, 1), '', 'hello']):
    print(x)
```

```
(0, 1)
hello
```

[63]:
```python
for nr in filter(lambda x: x % 2 == 0, [2, 1, 6, 8, 3, 5]):
    print(nr)
```

```
2
6
8
```

`map(function, iterable)` Return an iterator that applies function to every item of iterable, yielding the results.

[64]:
```python
for x in map(str.capitalize, ['paris', 'london', 'milan']):
    print(x)
```

```
Paris
London
Milan
```

[65]:
```python
for nr in map(lambda x: x ** 3, range(5)):
    print(nr)
```

```
0
1
8
27
64
```

`enumerate(iterable, start=0)` Returns an enumerate object which is an iterator that yields tuples containing a count (from start which defaults to 0) and the values obtained from iterating over iterable.

[66]:
```python
for index, char in enumerate("hello"):
    print(index, char)
```

```
0 h
1 e
2 l
3 l
4 o
```

`sorted(iterable, key=None, reverse=False)` Return a new sorted list from the items in iterable.

Has two optional arguments which must be specified as keyword arguments:

- `key` specifies a function of one argument that is used to extract a comparison key from each element in iterable (for example, `key=str.lower`). The default value is None (compare the elements directly).
- `reverse` is a boolean value. If set to True, then the list elements are sorted as if each comparison were reversed.

[67]:
```
sorted(('hi', 'hello', 'bye'), key=len, reverse=True)
```

[67]:
```
['hello', 'bye', 'hi']
```

`zip(*iterables)` Make an iterator that aggregates elements from each of the iterables.

Returns an iterator of tuples, where the i-th tuple contains the i-th element from each of the argument sequences or iterables. The iterator stops when the shortest input iterable is exhausted.

[2]:
```
for name, age, char in zip(['Anne', 'Luke', 'Jane'], range(20, 40, 7),␣
↪iter("hello")):
    print(name, age, char)
```

```
Anne 20 h
Luke 27 e
Jane 34 l
```

0.9.1 Exercises 9

1. Write a function `filter_short_words(word_list, n)` that returns the words in `word_list` shorter than `n`. Use `filter` built-in function and a lambda function.
2. Write a function that takes a list of tuples, where each tuple contains two integers, and returns an iterable containing the product of the two integers in each tuple. Use the `map` function and a lambda function to implement this.
3. Write a function that receives any number of strings and returns the list of unique strings ordered by number of appearances (most frequent → least frequent). Use `sorted` built-in function.

   E.g. `f('hello', 'there', 'hello', 'hi', 'hi', 'hello') -> ['hello', 'hi', 'there']`
4. Write your own implementation for map function (or any other function mentioned above).

0.10 10. `itertools` module

`itertools` is a standard library module that provides a collection of fast, memory-efficient tools for performing iterative tasks. These tools are designed to work with iterators and can be used to construct and manipulate iterators for various combinatorial and algebraic operations. The `itertools` module offers several functions that make it easy to work with infinite and finite iterators, and it helps to create and manipulate complex iterators with less code.

0.10.1 Infinite Iterators

1. **`itertools.count(start=0, step=1)`**

- Generates an infinite sequence of numbers, starting from `start` and incremented by `step`.

[69]:
```
import itertools

for num in itertools.count(10, 2):
    if num > 20:
        break
    print(num, end=" ")
```

```
10 12 14 16 18 20
```

2. **`itertools.cycle(iterable)`**
   - Cycles through the elements of an iterable indefinitely.

[70]:
```
count = 0
for item in itertools.cycle('ABCD'):
    if count > 10:
        break
    print(item, end=" ")
    count += 1
```

```
A B C D A B C D A B C
```

3. **`itertools.repeat(object, times=None)`**
   - Repeats an object a specified number of times (or indefinitely if `times` is `None`).

[71]:
```
for item in itertools.repeat("hello", 3):
    print(item)
```

```
hello
hello
hello
```

0.10.2 Combinatoric Iterators

1. **`itertools.product(*iterables, repeat=1)`**
   - Computes the Cartesian product of input iterables.

[72]:
```
for item in itertools.product("AB", "12"):
    print(item)
```

```
('A', '1')
('A', '2')
('B', '1')
('B', '2')
```

2. **`itertools.permutations(iterable, r=None)`**
   - Generates all possible permutations of the elements in the iterable.

[73]:
```python
for item in itertools.permutations("ABC", 2):
    print(item)
```

```
('A', 'B')
('A', 'C')
('B', 'A')
('B', 'C')
('C', 'A')
('C', 'B')
```

3. **`itertools.combinations(iterable, r)`**
   - Generates all possible combinations of `r` elements from the iterable.

[74]:
```python
for item in itertools.combinations("ABC", 2):
    print(item)
```

```
('A', 'B')
('A', 'C')
('B', 'C')
```

0.10.3 Filtering Iterators

1. **`itertools.compress(data, selectors)`**
   - Filters elements from `data` using `selectors` (another iterable of boolean values).

[75]:
```python
data = "ABCDEF"
selectors = [1, 0, 1, 0, 1, 0]
result = itertools.compress(data, selectors)
print(list(result))
```

```
['A', 'C', 'E']
```

2. **`itertools.dropwhile(predicate, iterable)`**
   - Drops elements from the iterable as long as the predicate is true, then returns the rest.

[76]:
```python
result = itertools.dropwhile(lambda x: x < 5, [1, 4, 6, 4, 1])
list(result)
```

[76]:
```
[6, 4, 1]
```

3. **`itertools.takewhile(predicate, iterable)`**
   - Returns elements from the iterable as long as the predicate is true.

[77]:
```python
result = itertools.takewhile(lambda x: x < 5, [1, 4, 6, 4, 1])
list(result)
```

[77]:
```
[1, 4]
```

0.10.4 Combining Iterators

1. **`itertools.chain(*iterables)`**

- Combines multiple iterables into a single sequence.

[78]:
```python
result = itertools.chain("ABC", range(3), [7, 8, 9])
list(result)
```

[78]:
```
['A', 'B', 'C', 0, 1, 2, 7, 8, 9]
```

2. **`itertools.chain.from_iterable(iterable)`**
   - Combines iterables from a single iterable of iterables.

[79]:
```python
result = itertools.chain.from_iterable(["ABC", range(3), [7, 8, 9]])
list(result)
```

[79]:
```
['A', 'B', 'C', 0, 1, 2, 7, 8, 9]
```

3. **`itertools.islice(iterable, start, stop, step)`**
   - Returns selected elements from the iterable, similar to slicing.

[80]:
```python
result = itertools.islice("ABCDEFG", 2, 5)
```

[81]:
```python
list(result)
```

[81]:
```
['C', 'D', 'E']
```

0.10.5 Grouping Iterators

1. **`itertools.groupby(iterable, key=None)`**
   - Groups adjacent elements of the iterable for which `key(item)` returns the same value. Generally, the iterable needs to already be sorted on the same key function.

[82]:
```python
words = ['generates', 'a', 'break', 'or', 'new', 'group', 'every', 'time',␣
 ↪'the', 'value', 'of', 'the', 'key', 'function', 'changes']
words.sort(key=len)
words
```

[82]:
```
['a',
 'or',
 'of',
 'new',
 'the',
 'the',
 'key',
 'time',
 'break',
 'group',
 'every',
 'value',
 'changes',
 'function',
```

```
'generates']
```

[83]:
```
for key, group in itertools.groupby(words, key=len):
    print(key, list(group))
```

```
1 ['a']
2 ['or', 'of']
3 ['new', 'the', 'the', 'key']
4 ['time']
5 ['break', 'group', 'every', 'value']
7 ['changes']
8 ['function']
9 ['generates']
```

0.10.6 Utility Functions

1. **`itertools.tee(iterable, n=2)`**
   - Returns `n` independent iterators from a single iterable.

[84]:
```
data = [1, 2, 3, 4]
iter1, iter2 = itertools.tee(data, 2)
print(list(iter1))
print(list(iter2))
```

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

2. **`itertools.zip_longest(*iterables, fillvalue=None)`**
   - Zips iterables together, filling in missing values with `fillvalue`.

[85]:
```
data1 = [1, 2, 3]
data2 = "ab"
result = itertools.zip_longest(data1, data2, fillvalue='-')
print(list(result))
```

```
[(1, 'a'), (2, 'b'), (3, '-')]
```

3. **`itertools.starmap(func, seq)`**
   - Similar to built-in `map`, but works with functions with multiple arguments. `seq` will be an iterable of n-length sequences, where `n` is the number of parameters expected by `func`. Returns an iterator containing: `func(*seq[0]), func(*seq[1]), ...`

[86]:
```
data = [(7, 2), (3, 3), (100, 0), (4, 4)]
result = itertools.starmap(pow, data)
list(result)
```

[86]:
```
[49, 27, 1, 256]
```

0.10.7 Exercises 10

1. Generate all possible combinations of rolling two six-sided dice.

2. Solve the same problem as exercise 9.2 above, but with `starmap`.
3. Try out other functions in `itertools` module.