02. Built-in Types

October 13, 2025

0.1 1. Everything in Python is an Object

One of the most important features of the Python language is that *everything* is an **object**. This concept is pivotal to understanding how Python works and how to write effective Python code. Let's delve into what this means and its implications.

Objects and Classes In Python, every piece of data is represented as an object, which is an instance of a class. A class serves as a blueprint for creating objects, defining the properties (attributes) and behaviors (methods) that the objects created from the class will have. Even primitive data types like integers and strings are objects derived from their respective classes.

[1]:
```
x = 42            # x is an object of the `int` class
y = "Hello"       # y is an object of the `str` class
```

In order to get the class of an object we will use the `type` built-in function:

[2]:
```
type(x)
```

[2]:
```
int
```

[3]:
```
type(y)
```

[3]:
```
str
```

Object Attributes and Methods Since everything is an object, even simple data types come with associated attributes and methods. For instance, strings have methods for manipulating text, and lists have methods for managing collections of items.

[4]:
```
x.numerator  # accessing `numerator` attribute of an integer object
```

[4]:
```
42
```

[5]:
```
y.upper()  # calling `upper` method of a string object
```

[5]:
```
'HELLO'
```

In order to see all methods and attributes of an object (or a type), you can use `dir()` built-in function:

[6]:
```
dir(y)  # all attributes and methods of a string object
```

```
[6]: ['__add__',
      '__class__',
      '__contains__',
      '__delattr__',
      '__dir__',
      '__doc__',
      '__eq__',
      '__format__',
      '__ge__',
      '__getattribute__',
      '__getitem__',
      '__getnewargs__',
      '__getstate__',
      '__gt__',
      '__hash__',
      '__init__',
      '__init_subclass__',
      '__iter__',
      '__le__',
      '__len__',
      '__lt__',
      '__mod__',
      '__mul__',
      '__ne__',
      '__new__',
      '__reduce__',
      '__reduce_ex__',
      '__repr__',
      '__rmod__',
      '__rmul__',
      '__setattr__',
      '__sizeof__',
      '__str__',
      '__subclasshook__',
      'capitalize',
      'casefold',
      'center',
      'count',
      'encode',
      'endswith',
      'expandtabs',
      'find',
      'format',
      'format_map',
      'index',
      'isalnum',
      'isalpha',
```

```
 'isascii',
 'isdecimal',
 'isdigit',
 'isidentifier',
 'islower',
 'isnumeric',
 'isprintable',
 'isspace',
 'istitle',
 'isupper',
 'join',
 'ljust',
 'lower',
 'lstrip',
 'maketrans',
 'partition',
 'removeprefix',
 'removesuffix',
 'replace',
 'rfind',
 'rindex',
 'rjust',
 'rpartition',
 'rsplit',
 'rstrip',
 'split',
 'splitlines',
 'startswith',
 'strip',
 'swapcase',
 'title',
 'translate',
 'upper',
 'zfill']
```

To get the inline documentation of an object (be it module, class, method, function), call `help()` built-in function on it:

[7]:
```
help(range)  # calling `help` on a built-in class
```

```
Help on class range in module builtins:

class range(object)
 |  range(stop) -> range object
 |  range(start, stop[, step]) -> range object
 |
 |  Return an object that produces a sequence of integers from start (inclusive)
 |  to stop (exclusive) by step.  range(i, j) produces i, i+1, i+2, …, j-1.
```

```
|  start defaults to 0, and stop is omitted!  range(4) produces 0, 1, 2, 3.
|  These are exactly the valid indices for a list of 4 elements.
|  When step is given, it specifies the increment (or decrement).
|
|  Methods defined here:
|
|  __bool__(self, /)
|      True if self else False
|
|  __contains__(self, key, /)
|      Return bool(key in self).
|
|  __eq__(self, value, /)
|      Return self==value.
|
|  __ge__(self, value, /)
|      Return self>=value.
|
|  __getattribute__(self, name, /)
|      Return getattr(self, name).
|
|  __getitem__(self, key, /)
|      Return self[key].
|
|  __gt__(self, value, /)
|      Return self>value.
|
|  __hash__(self, /)
|      Return hash(self).
|
|  __iter__(self, /)
|      Implement iter(self).
|
|  __le__(self, value, /)
|      Return self<=value.
|
|  __len__(self, /)
|      Return len(self).
|
|  __lt__(self, value, /)
|      Return self<value.
|
|  __ne__(self, value, /)
|      Return self!=value.
|
|  __reduce__(…)
|      Helper for pickle.
|
```

```
 |  __repr__(self, /)
 |      Return repr(self).
 |
 |  __reversed__(…)
 |      Return a reverse iterator.
 |
 |  count(…)
 |      rangeobject.count(value) -> integer -- return number of occurrences of
value
 |
 |  index(…)
 |      rangeobject.index(value) -> integer -- return index of value.
 |      Raise ValueError if the value is not present.
 |
 |  ----------------------------------------------------------------------
 |  Static methods defined here:
 |
 |  __new__(*args, **kwargs) from builtins.type
 |      Create and return a new object.  See help(type) for accurate signature.
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors defined here:
 |
 |  start
 |
 |  step
 |
 |  stop
```

[8]:
```
help(str.strip)  # calling help on a string method
```

```
Help on method_descriptor:

strip(self, chars=None, /)
    Return a copy of the string with leading and trailing whitespace removed.

    If chars is given and not None, remove characters in chars instead.
```

0.1.1 Exercises 1

1. Choose a class - be it a built-in type like `int`, `str`, `dict`, or from a Standard Library module like `datetime.date` or even from a 3rd party library, like `requests.Request`. Instantiate the class. You now have two names: the class and the object. Call `print`, `str`, `repr`, `type` and `dir` on both these names.
2. Call `help` on the following:
   - a module

- a function
- a class
- a method of a class
- a method of an object

You can use the class and object in exercise 1, or new ones.

0.2 2. Overview of Python's Built-in Types

Python offers a rich set of built-in types that allow you to handle different kinds of data efficiently. Here's an overview of the most commonly used built-in types:

Numeric Types An overview of Python's numeric types can be found here.

1. **int**: Represents integers. Supports unlimited precision.

   ```
   a = 42
   b = -7
   ```

2. **float**: Represents floating-point numbers. Conforms to the IEEE 754 double-precision standard.

   ```
   c = 3.14
   d = -2.718
   ```

3. **complex**: Represents complex numbers with real and imaginary parts.

   ```
   e = 2 + 3j
   f = -1 - 4j
   ```

Boolean Type Read more about `bool` here.

4. **bool**: Represents Boolean values, `True` or `False`.

   ```
   u = True
   v = False
   ```

0.2.1 Exercises 2.1

1. Write a function `is_prime` that checks whether a given number is prime.

Sequence Types An overview of Python's sequence types can be found here.

5. **list**: Mutable sequence of objects, can hold mixed types.

   ```
   i = [1, 2, 3]
   j = ["apple", "banana", "cherry"]
   ```

6. **tuple**: Immutable sequence of objects, can hold mixed types.

   ```
   k = (1, 2, 3)
   l = ("a", "b", "c")
   ```

7. **range**: Immutable sequence of numbers, commonly used for looping a specific number of times.

```
m = range(10)  # 0 to 9
n = range(1, 10, 2)  # 1, 3, 5, 7, 9
```

0.2.2 Exercises 2.2

1. Write a program to create a list of tuples (each containing the number, the square of the number and the cube of the number) from a range object.

Text Sequence Type Read more about `str` here.

8. **str**: Immutable sequence of characters.

```
g = "Hello, World!"
h = 'Python'
```

0.2.3 Exercises 2.3

1. Write a function `normalize` that receives a string as a parameter and returns the string modified as follows:
   - any leading/trailing whitespace is removed
   - any leading/trailing punctuation is removed (you can use `string.punctuation`)
   - the string is transformed to all lowercase

Binary Sequence Types An overview of binary sequence types can be found here.

9. **bytes**: Immutable sequence of bytes.

```
w = b"hello"
x = bytes([104, 101, 108, 108, 111])
```

10. **bytearray**: Mutable sequence of bytes.

```
y = bytearray(b"hello")
z = bytearray([104, 101, 108, 108, 111])
```

0.2.4 Exercises 2.4

1. Given the following string: `python  greeting = "  - !"`
   - print its length
   - encode it using `utf-8` encoding and print the `bytes` object's length
   - encode it using `cp866` encoding and print the `bytes` object's length
   - decode the `bytes` objects to string and check if the result is equal to the initial string

Set Types An overview of set types can be found here.

11. **set**: Mutable collection of unique elements.

```
q = {1, 2, 3}
r = set(["apple", "banana", "cherry"])
```

12. **frozenset**: Immutable collection of unique elements.

```
s = frozenset([1, 2, 3])
t = frozenset(["apple", "banana", "cherry"])
```

0.2.5 Exercises 2.5

1. Write a function that receives two texts and returns a set of common words: `python get_commmon_words("good morning everyone", "everyone was good today")  # should return {"good", "everyone"}`

Mapping Types Read more about `dict` here.

13. **dict**: Mutable collection of key-value pairs.

```
o = {"name": "Alice", "age": 25}
p = {1: "one", 2: "two"}
```

0.2.6 Exercises 2.6

1. Write a function `invert_dict` that inverts a dictionary, i.e., the keys become values and the values become keys. Handle cases where values are not unique by storing the keys in a list.

0.3 3. Mutable vs Immutable Types

This chapter will be a recap of the built-in Python data types, with a focus on their mutability / immutability.

In programming, an object is considered *immutable* if its state cannot be altered after it has been created. Conversely, a *mutable* object allows modifications to its internal state post-creation. Essentially, the ability to change an object's state or its contained data determines whether the object is mutable or immutable.

Immutable objects are prevalent in functional programming, whereas mutable objects are extensively used in object-oriented programming. As a multiparadigm programming language, Python offers both mutable and immutable objects, giving you the flexibility to choose the most suitable type for your problem-solving needs.

0.3.1 3.1. Variables and objects

A variable in Python is a symbolic name that references an object. Unlike in many other programming languages, variables in Python do not have fixed types; instead, they can refer to objects of any type and can be reassigned to objects of different types.

Python objects are concrete pieces of information that live in specific memory positions on your computer. In Python, everything is an object, including primitive data types like integers and strings, modules and functions. An object has three main characteristics:

- **Identity**: A unique identifier for the object, which can be obtained using the `id()` function.
- **Type**: The type of the object, which can be obtained using the `type()` function.
- **Value**: The data contained in the object.

[9]:
```
name = "Jane"  # variable `name` references an object of type `str`
```

[10]:
```
print(name)  # the value of the object
```

```
Jane
```

[11]:
```
id(name)  # the identity of the object
```

[11]:
```
4428286448
```

[12]:
```
type(name)  # the type of the object
```

[12]:
```
str
```

The key point to understand is that variables and objects are distinct entities in Python:

- **Variables** hold references to objects.
- **Objects** reside in specific memory locations.

Though these concepts are independent, they are closely intertwined. Once a variable is created with an assignment statement, the referenced object can be accessed using the variable name throughout the code. If the object is mutable, its state can be modified via the variable. Mutability or immutability is a characteristic of objects, not variables.

If the referenced object is immutable, its internal state or data cannot be changed. You can, however, reassign the variable to reference a different object, which may or may not be of the same type as the original object.

Without a reference (variable) to an object, it cannot be accessed in the code. If all references to an object are lost or removed, Python's garbage collector will eventually free the memory occupied by that object for future use.

[13]:
```
name = "John"  # reference to the object above ("Jane") is lost
```

[14]:
```
id(name)  # variable points to a new object
```

[14]:
```
4428292256
```

0.3.2 3.2. Immutable built-in data types

Numeric types like `int`, `float`, and `complex` hold single-item objects that cannot be altered during code execution, making them immutable. In Python, the `bool` class, which represents Boolean values, is a subclass of `int` and is also immutable.

Among collection types, `str`, `bytes`, and `tuple` are immutable, despite being able to store multiple-item values.

There is no way in which you can change the intrinsic value of a immutable object. Some operations will apparently change the value of a variable, but its identity will also change in this process.

Augmented operations on immutable objects Let's take a look at augmented operators, like `+=`, `*=`, etc. When Python evaluates the expression `x += y`, it first searches for `__iadd__` magic method on `x` (`x.__iadd__(y)`), short for *in place addition*, which would change the value of `x` in place, meaning the object would mutate. For immutable types, though, this method is not

implemented, and the augmented operation falls back on `x = x + y` (`x = x.__add__(y)`), which does the reassignment of the `x` variable to a new value.

[15]:
```
"__iadd__" in dir(int)
```

[15]:
```
False
```

[16]:
```
"__iadd__" in dir(str)
```

[16]:
```
False
```

[17]:
```
"__iadd__" in dir(tuple)
```

[17]:
```
False
```

Let's take an example:

[18]:
```
number = 5
```

[19]:
```
id(number)
```

[19]:
```
4390244000
```

[20]:
```
number += 1
```

[21]:
```
id(number)
```

[21]:
```
4390244032
```

You can replicate this example using any other immutable type and any other augmented operation, you'll notice the same behaviour.

Supported operations for immmutable types Except for the augmented operations, which only *seem* to change the object, immutable types don't support other mutating operations. For example, trying to change an item of a string, bytes object or tuple will result in the same error:

[22]:
```
my_tuple = (1, 2, 3)
```

[23]:
```
my_tuple[0] = 10
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[23], line 1
----> 1 my_tuple[0] = 10

TypeError: 'tuple' object does not support item assignment
```

Methods on immutable types In terms of methods supported, some immutable types only support *read* operations, like `tuple`:

[24]:
```
for attr in dir(tuple):
    if attr.startswith("__"):
        continue
    print(attr)
```

```
count
index
```

While others, like `str`, define methods that *sound like* they do mutations on the object on which they are called, but actually return modified copies of the original object, leaving it unchanged:

[25]:
```
my_string = "hello world"
```

[26]:
```
my_string.replace("l", "*")
```

[26]:
```
'he**o wor*d'
```

[27]:
```
my_string
```

[27]:
```
'hello world'
```

Another nice effect of this behaviour is the fact that `str` methods can be chained:

[28]:
```
my_string.replace("l", "*").upper().center(20)
```

[28]:
```
'    HE**O WOR*D     '
```

0.3.3 Exercises 3.3

1. Can you think of any other strategy to mutate immutable objects? Try it out and see if you can change the intrinsic value of an immutable object.

0.3.4 3.3. Mutable built-in data types

Mutable data types are another face of the built-in types in Python. The language provides a few useful mutable collection types that you can use in many situations. These types allow you to change the value of specific items without affecting the identity of the container object. Most notable examples are: `list`, `dict` and `set`.

Augmented operations on mutable types As we said earlier, using an augmented operator will search for the corresponding dunder method; for mutable types, these methods are implemented:

[29]:
```
"__iadd__" in dir(list)
```

[29]:
```
True
```

[30]:
```
"__iand__" in dir(set)
```

[30]: True

[31]:
```
"__ior__" in dir(dict)
```

[31]: True

So, using augmented operators will change mutable objects in place:

[32]:
```
numbers = [2, 6, 7, 1]
id(numbers)
```

[32]: 4434198656

[33]:
```
numbers += [6, 4, 1]
```

[34]:
```
numbers
```

[34]: [2, 6, 7, 1, 6, 4, 1]

[35]:
```
id(numbers)
```

[35]: 4434198656

Set/delete item/slice for lists and dictionaries An operation supported only by lists/dictionaries is adding/updating/deleting an element (or a slice, for lists) by its index/key:

[36]:
```
numbers[4] = 100
```

[37]:
```
numbers
```

[37]: [2, 6, 7, 1, 100, 4, 1]

[38]:
```
id(numbers)
```

[38]: 4434198656

Mutating methods on mutable types Another feature of mutable objects is that mutating methods will change the object on which they are called **in place**, and return `None`.

[39]:
```
sort_result = numbers.sort()
```

[40]:
```
print(sort_result)
```

None

[41]:
```
numbers
```

[41]:
```
[1, 1, 2, 4, 6, 7, 100]
```

[42]:
```
id(numbers)
```

[42]:
```
4434198656
```

0.3.5 Exercises 3.3

1. Can you think of another type of operation available on mutable objects that changes the value of the object?

0.3.6 3.4. Common Pitfalls Related to Mutability

3.4.1. Aliasing Variables Assigning an existing variable to a new one creates an alias, meaning both variables reference the same memory address and object. Python variables store references to memory addresses, not the data itself. Thus, an alias shares the same reference as the original variable.

[43]:
```
a_number = 745
```

[44]:
```
another_number = a_number
```

[45]:
```
id(a_number), id(another_number)
```

[45]:
```
(4433950096, 4433950096)
```

For immutable types, like numbers, strings and tuples, aliases work like copies of the original object, because operations like augmented assignment will actually create new objects, so you don't have to worry about mutations. With mutable types, mutation on a given alias affects the rest of the aliases:

[46]:
```
numbers = [1, 2, 3]
other_numbers = numbers
numbers += [4, 5]
```

[47]:
```
numbers
```

[47]:
```
[1, 2, 3, 4, 5]
```

[48]:
```
other_numbers
```

[48]:
```
[1, 2, 3, 4, 5]
```

3.4.2. Creating Copies of Mutable Objects As we've seen, creating an alias is not a good strategy for copying a mutable object. Copies of mutable objects can be created in multiple ways, but there are two types of copies which can be created: - a **shallow copy**, which you can create: - by calling the `.copy()` method (`list`, `set` and `dict` implement this method) - by using slicing for lists - by using `copy.copy()` function - a **deep copy**, which you can create using `copy.deepcopy()` function.

When you make a shallow copy of an existing object, you create a new object with a different identity. However, the internal components or data items in the new object are just references to the members in the original object. On the other hand, if you make a deep copy, then you create a completely new copy of the original object.

Let's see an example using `set.copy()` method:

[49]:
```
original_set = {1, 3, 4, 7}
```

[50]:
```
new_set = original_set.copy()
```

[51]:
```
new_set
```

[51]:
```
{1, 3, 4, 7}
```

[52]:
```
new_set is original_set
```

[52]:
```
False
```

[53]:
```
new_set.add(8)
```

[54]:
```
new_set  # new_set was changed
```

[54]:
```
{1, 3, 4, 7, 8}
```

[55]:
```
original_set  # original_set has the original value
```

[55]:
```
{1, 3, 4, 7}
```

And an example using list slicing (`list.copy()` would have the same effect):

[56]:
```
original_list = [1, 2, 3, [4, 5]]
```

[57]:
```
new_list = original_list[:]
```

[58]:
```
new_list is original_list  # the new list has a different identity
```

[58]:
```
False
```

[59]:
```
new_list[0] is original_list[0]  # members of the two lists are identical
```

[59]:
```
True
```

[60]:
```
new_list[0] = 100  # changing new_list
```

[61]:
```
new_list  # new_list is changed
```

[61]:
```
[100, 2, 3, [4, 5]]
```

[62]:
```
original_list  # original_list is not affected
```

[62]:
```
[1, 2, 3, [4, 5]]
```

[63]:
```
new_list[3].append(100)  # changing a list inside new_list
```

[64]:
```
new_list  # new_list is changed
```

[64]:
```
[100, 2, 3, [4, 5, 100]]
```

[65]:
```
original_list  # original_list is also affected
```

[65]:
```
[1, 2, 3, [4, 5, 100]]
```

Now, let's create a deep copy for a dictionary and see the differences:

[66]:
```
person = {"name": "Jane", "hobbies": ["painting", "swimming"]}
```

[67]:
```
import copy
shallow_person = copy.copy(person)
deep_person = copy.deepcopy(person)
```

[68]:
```
shallow_person
```

[68]:
```
{'name': 'Jane', 'hobbies': ['painting', 'swimming']}
```

[69]:
```
deep_person
```

[69]:
```
{'name': 'Jane', 'hobbies': ['painting', 'swimming']}
```

[70]:
```
shallow_person["hobbies"].append("hiking")
deep_person["hobbies"].append("playing guitar")
```

[71]:
```
person, shallow_person, deep_person
```

[71]:
```
({'name': 'Jane', 'hobbies': ['painting', 'swimming', 'hiking']},
 {'name': 'Jane', 'hobbies': ['painting', 'swimming', 'hiking']},
 {'name': 'Jane', 'hobbies': ['painting', 'swimming', 'playing guitar']})
```

In this example, `person` and its two copies are completely independent objects. They have different identities. The values stored in `shallow_person` share the same identity as those in `person`. The behavior of shallow copies can save a lot of memory when your objects are quite large. You won't need to use the same amount of memory to store copies of all the data, you create new objects only for the top-objects.

In the meantime, the values in `deep_person` have different identities from those in the original object, `person`. The inner lists are now completely different objects. They don't have the same identity. Because of this behavior, deep copies may duplicate the memory usage because they need to store an entire copy of the data.

0.3.7 Exercises 3.4.2

1. Create a list of lists `matrix`, where each inner list contains numbers. Create three other variables based on `matrix`:
   - an alias
   - a shallow copy
   - a deep copy
2. Try out different mutations (methods like `append`, `remove`, `sort`, item/slice assignment/deletion, etc) on `matrix` and see which of the copies change.
3. Try out different mutations (methods like `append`, `remove`, `sort`, item/slice assignment/deletion, etc) on a inner list inside `matrix` and see which of the copies change.

3.4.3. Mutating arguments in functions When creating a function in Python, you may need it to accept arguments as input for its computations. If you pass a global variable as an argument, the function's parameter becomes an alias for that variable, sharing the same memory address.

When the argument is a mutable object, any modifications within the function affect the original object, potentially causing subtle bugs. Therefore, be cautious when coding functions that receive mutable objects, ensuring you consider whether the caller expects the arguments to be modified. Consider the following function:

[72]:
```
def capitalize_all(names):
    for i in range(len(names)):
        names[i] = names[i].capitalize()
    return names
```

[73]:
```
first_names = ["jane", "michael", "allison"]
```

[74]:
```
capitalize_all(first_names)
```

[74]:
```
['Jane', 'Michael', 'Allison']
```

[75]:
```
first_names
```

[75]:
```
['Jane', 'Michael', 'Allison']
```

Our function takes a list (a mutable object) as argument and, because we mutated the argument, these mutations affect the input data. Now `first_names` contains modified values rather than the original data. This may be the wrong final result because you're losing your original data. To prevent this kind of issue, you must avoid mutating arguments in your functions:

[76]:
```
def capitalize_all(names):
    result = []
    for name in names:
        result.append(name.capitalize())
    return result
```

[77]:
```
first_names = ["jane", "michael", "allison"]
```

[78]:
```
capitalize_all(first_names)
```

[78]:
```
['Jane', 'Michael', 'Allison']
```

[79]:
```
first_names
```

[79]:
```
['jane', 'michael', 'allison']
```

0.3.8 Exercises 3.4.3

1. Considering the following function:

```
def merge(first, second):
    first += second
    return first
```

call it with parameters of the following types: `int`, `str`, `tuple`, `list`. Use a global variable as the first parameter and a value created on the spot as the second parameter. Print the result and the global variable. Is the global variable changed?

3.4.4. Using Mutable Default Values Using mutable default arguments in functions can lead to unexpected behavior because the default argument is evaluated only once when the function is defined, not each time the function is called. This is because functions are first-class objects, and the function object gets created on its definition, default argument values being *member data* for the function. This means that if the default argument is a mutable object like a list or dictionary, modifications to this object persist across function calls.

[80]:
```
def append_to_list(value, lst=[]):
    lst.append(value)
    return lst
```

[81]:
```
append_to_list(1)
```

[81]:
```
[1]
```

[82]:
```
append_to_list(2)
```

[82]:
```
[1, 2]
```

In this example, the list `lst` retains its values across multiple function calls, resulting in a cumulative effect. To avoid this, use `None` as the default value and create a new list inside the function if needed:

[83]:
```
def append_to_list(value, lst=None):
    if lst is None:
        lst = []
    lst.append(value)
    return lst
```

[84]:
```
append_to_list(1)
```

[84]:
```
[1]
```

[85]:
```
append_to_list(2)
```

[85]:
```
[2]
```

Here, each call to `append_to_list` starts with a fresh list, avoiding the issue with mutable default arguments.

Mutable default arguments can be useful in certain situations, for example for local caches:

[86]:
```
def factorial(nr, memo={}):
    if nr in memo:
        print(f"Getting value from cache for nr={nr}")
        return memo[nr]
    else:
        print(f"Computing value for nr={nr}")
        result = nr * factorial(nr-1) if nr else 1
        memo[nr] = result
        return result
```

In this function, `memo` will be used to save results for different parameters.

[87]:
```
factorial(5)
```

```
Computing value for nr=5
Computing value for nr=4
Computing value for nr=3
Computing value for nr=2
Computing value for nr=1
Computing value for nr=0
```

[87]:
```
120
```

[88]:
```
factorial(3)
```

```
Getting value from cache for nr=3
```

[88]:
```
6
```

[89]:
```
factorial(7)
```

```
Computing value for nr=7
Computing value for nr=6
Getting value from cache for nr=5
```

[89]:
```
5040
```

0.3.9 Exercises 3.4.4

1. Considering the following function: `python def http_get(url, headers={}): if "/en/" in url: headers["Accept-Language"] = "en" elif "/de/" in url: headers["Accept-Language"] = "de" print(f"Making HTTP request to {url} with headers {headers}")`

   call it multiple times:

   - first, send both an URL and custom headers
   - then, send an URL containing `/de/` (like `https://www.dw.com/de/themen/s-9077`) and no headers
   - then, send an URL containing a reference to another language (like `https://www.dw.com/fr/`) and no headers

   What do you notice? How can you fix this behavior?