python_tricks

July 13, 2024

1 Advanced Python Concepts

In this tutorial, a few advanced concepts are introduced. This includes

- Python sets
- packing and unpacking
- context manager
- decorator and factories
- Exceptions

1.1 Python sets

Python offers a lot of built-in functionality. Next to the most famous containers - tuples, lists, dictionaries - there are also sets (and more).

Sets act like a mathematical set. They are **unordered** and **cannot contain duplicates**.

They can be created with the `set()` function or with the `{}` syntax (but using single entries without colon - otherwise we would get a dictionary).

[1]:
```
set1 = {1, 2, 2, "a"}
list2 = [2, "a", "b", 4, 4]
set2 = set(list2)
print(f"set1: {set1}\nset2: {set2}\nlist2: {list2}")
```

```
set1: {1, 2, 'a'}
set2: {2, 4, 'b', 'a'}
list2: [2, 'a', 'b', 4, 4]
```

We see that only unique entries are stored in the set and that the order of the entries is not preserved.

More elements can be added (*inplace*) to a set with the `add()` method.

[2]:
```
set1.add(3)
print(f"set1: {set1}")
```

```
set1: {1, 2, 3, 'a'}
```

[3]:
```
set1.remove(3)
print(f"set1: {set1}")
```

```
set1: {1, 2, 'a'}
```

[4]:
```
# multiple elements
set1.update([4, 5, 6])
```

[5]:
```
set1
```

[5]:
```
{1, 2, 4, 5, 6, 'a'}
```

1.1.1 Set operations

Sets can be used to perform mathematical set operations. The following operations are supported: - `union`: returns the union of two sets (alternative syntax `A | B`) - `intersection`: returns the intersection of two sets (alternative syntax `A & B`) - `difference`: returns the difference of two sets (alternative syntax `A - B`) - `symmetric_difference`: returns the symmetric difference (xor) of two sets (alternative syntax `A ^ B`)

All of this methods return a new set, so the original sets are not modified. They all have a `_update` version, which modifies the original set inplace. (Exception: `union_update` is simply `update` as seen above).

Furthermore, many tests can be performed on sets, such as: - `element in set`: checks if an element is in the set (for example `"a" in set1`) - `issubset`: checks if a set is a subset of another set (alternative syntax `A <= B`, for *proper* subset `A < B`) - `issuperset`: checks if a set is a superset of another set (alternative syntax `A >= B`, for *proper* superset `A > B`) - `isdisjoint`: checks if two sets have no common elements (alternative syntax `A /= B`) - `isempty`: checks if a set is empty - and more…

[6]:
```
print(set1)
print(set2)
set1.intersection(set2)
```

```
{1, 2, 4, 5, 6, 'a'}
{2, 4, 'b', 'a'}
```

[6]:
```
{2, 4, 'a'}
```

[7]:
```
2 in set1
```

[7]:
```
True
```

1.1.2 Immutable types only

As for dictionary keys, only *immutable* types can be used as keys in a dictionary (i.e. strings, numbers, tuples, …). Otherwise, a `TypeError` is raised.

[8]:
```
set3 = {list2, 5, 3}  # Fails with a `TypeError`
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
```

```
Cell In[8], line 1
----> 1 set3 = {list2, 5, 3}  # Fails with a `TypeError`

TypeError: unhashable type: 'list'
```

[9]:
```
set4 = {set2, 5, 3}  # Fails with a `TypeError`
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[9], line 1
----> 1 set4 = {set2, 5, 3}  # Fails with a `TypeError`

TypeError: unhashable type: 'set'
```

[10]:
```
set5 = {tuple(list2), (4, 3), 3}
print(f"set5: {set5}")
```

```
set5: {3, (2, 'a', 'b', 4, 4), (4, 3)}
```

[11]:
```
(4, 3) in set5
```

[11]:
```
True
```

1.2 Packing and unpacking of values

[12]:
```
# packing
tuple_ = 2,3,5,7
tuple_
```

[12]:
```
(2, 3, 5, 7)
```

[13]:
```
# unpacking
a, b, c, d = tuple_
print(c)
```

```
5
```

Using * we can group/ungroup list-like objects. They act as a "removal" of the parenthesis when situated on the right and as an "adder" of parenthesis when situated on the left of the assigmenemt operator (=).

Let's play around...

[14]:
```
a, c, *b = [3, 4, 4.5, 5, 6]
```

[15]:
```
b
```

[15]:
```
[4.5, 5, 6]
```

As can be seen, b catches now all the remaining elements in a list. Interesting to see is also the special case if no element is left.

[16]:
```
d1, d2, *d3, d4 = [1, 2, 3]  # nothing left for d3
```

[17]:
```
d3
```

[17]:
```
[]
```

This is simply an empty list. However, this has the advantage that we know that it is *always* a list.

Multiple unpackings can be added together (however, the other way around does not work: multiple *packings* are not possible as it is ill-defined which variable would get how many elements).

[18]:
```
a = [3, 4, 5]
```

[19]:
```
d, e, f, g, h, i = *a, *b
```

Now we should be able to understand the `*args` and `**kwargs` for functions. `*args` works like the examples above. `**kwargs` works very similar, but exists only in parameter-lists of functions. There it packs/unpacks keyword-arguments into/from a dictionary. Let's look at it:

[20]:
```
def func(*args, **kwargs):
    print(f'args are {args}')
    print(f"kwargs are {kwargs}")
```

[21]:
```
mykwargs = {'a': 5, 'b': 3}
myargs = [1, 3, 4]
func(*myargs, **mykwargs)
```

```
args are (1, 3, 4)
kwargs are {'a': 5, 'b': 3}
```

[22]:
```
func(5, a=4)
```

```
args are (5,)
kwargs are {'a': 4}
```

[23]:
```
def func(a, x, **kwargs):
    print(x, a, kwargs)

kwargs = {"x": 2}
kwargs['y'] = 3
func(1,**kwargs)
func(x=3, a=5)
```

```
2 1 {'y': 3}
3 5 {}
```

[24]:
```
# play around with it!
```

1.2.1 Aside: do not confuse the different parameter/argument types

function definition

- required parameters: specified without a default value: `def f(p)`
- optional parameters: have a default value: `def f(p=42)`

calling a function

- positional arguments: are given without a keyword `f(0)`
- key-word arguments are passed using a keyword `f(p=0)`

Both requried and optional parameters can be passed as positional or key-word arguments!

(for the curious: there is a bit more that can be done)

1.3 Context manager

A context manager is an object that responds to a `with` statement. It may returns something. The basic idea is that some action is performed when entering a context and again when exiting it.

```
with context as var:
    # do something
```

translates to

```
# execute context entering code
var = result_from_context_entering_code
# do something
# execute context leaving code
```

The great advantage here is that the "leaving code" is automatically executed whenever we step out of the context!

This proved to be incredibly useful when operations have cleanup code that we need to execute yet that is tedious to write manually and can be forgotten.

1.3.1 Using a class

We can have control over the enter and exit methods by creating a class and implementing the two methods `__enter__` and `__exit__`

[25]:
```
class MyContext:

    def __init__(self, x):
        self.x = x
        print('initialized')

    def __enter__(self):
        x = self.x
        print('entered')
        return x**2
```

```
    def __exit__(self, type_, value, traceback):  # but let's not go into␣
↪things in detail here
        self.x = None
        print('exited')
```

[26]:
```
context_5 = MyContext(5)
```

```
initialized
```

[27]:
```
with context_5 as value:
    print("inside")
    print(value)

print("Outside")
print(value)
print("context_5.x:", context_5.x)
```

```
entered
inside
25
exited
Outside
25
context_5.x: None
```

[28]:
```
context_5 = MyContext(5)
xsquare = context_5.__enter__()
print("inside", xsquare)
#context_5.__exit__(...)
```

```
initialized
entered
inside 25
```

Where is this useful Basically with stateful objects. This includes anything that can be set and changed (mutable objects).

[29]:
```
with open('python_tricks.ipynb') as notebook:
    print("".join(notebook.readlines()[:5]))
```

```
{
 "cells": [
  {
   "cell_type": "markdown",
   "metadata": {
```

The implementation roughly looks like this:

[30]:
```
class MyOpen:

    def __init__(self, f, mode):
        self._file = f
        self._mode = mode

    def __enter__(self):
        return self._file.open(self._mode)

    def __exit__(self, type_, value, traceback):  # but let's not go into␣
 ↪things in detail here
        self._file.close()
```

Exercise: create a context manager that *temporarily* sets a `'value'` key to 42 of a dict and switches it back to the old value on exit

[31]:
```
testdict = {'value': 11, 'name': 'the answer'}
```

to be invoked like this

```
with manager(testdict) as obj:
    # here the value is 42
# here the value is 11
```

1.4 Generators

Generators are special functions that use `yield`.

What is `yield`?: It's like a return, except that the execution stops at the `yield`, lets other code execute and, at some point, **continues** again where the yield was. Examples are: - iterator: a function that yields elements. Everytime it is called, it is supposed to yield an element and then continue from there - asynchronous programing: it stops and waits until something else is finished - in the context manager, as we will see

[32]:
```
def squares(start=0, num=100):
    i = start
    while i < start+num:
        yield i**2
        i += 1
```

[33]:
```
gen = squares()
print(next(gen))
print(next(gen))
```

```
0
1
```

[34]:
```
for v in squares(5,10):
    print(v)
```

```
25
36
49
64
81
100
121
144
169
196
```

1.5 Function factories and Decorators

Sometimes we can't write a function fully by hand but want to create it programatically. This pattern is called a "factory". To achieve this, instead of having a function that returns an integer (an object), a list (an object), a dict (an object) or an array (an object), we return a function (an object). We see that the concept of Python, "everything is an object", starts being very useful here.

[35]:
```
# goal: create a power function but let the power be specified by the code
# exponent = 2

# def power_2(x):  # not 2, but "exponent"
#     return x ** exponent
```

[36]:
```
def make_power_func(power):
    def func(x):
        print("Executed")
        return x ** power

    return func
```

[37]:
```
pow3 = make_power_func(3)
```

[38]:
```
pow3(2)
```

```
Executed
```

[38]:
```
8
```

1.5.1 Pitfall lexical lookup

The name `power` inside the function `func` is remembered *by the name* (**lexical**) and **NOT BY REFERENCE**. This means that the last object named `power` is the one that is used inside `func`.

[39]:
```
def make_power_func(power):
    def func(x):
        return x ** power
```

```
    power = 42
    return func
power = 1
```

[40]:
```
pow3 = make_power_func(3)
```

[41]:
```
pow3(2)
```

[41]:
```
4398046511104
```

Another example is to create a timing wrapper. **Exercise**: create a timing function that can be used as follows

```
timed_pow3 = fime_func(pow3)
pow3(...)
```

HINT, scetch of solution

```
def time_func(func):
    def new_func(...):
        print('start')
        func(...)
        print('stop')
    return new_func
```

[42]:
```
import time

# SOLUTION
def timed_func(func):
    def wrapped_func(*args, **kwargs):
        start = time.time()
        res = func(*args, **kwargs)
        end = time.time()
        print(f'time needed: {end - start}')
        return res

    return wrapped_func
```

[43]:
```
def add_notime(x, y):
    return x + y
```

[44]:
```
add_timed = timed_func(add_notime)
```

[45]:
```
add_timed(y=4, x=5)
```

```
time needed: 9.5367431640625e-07
```

[45]:
```
9
```

[46]:
```
add_timed(5, y=4)
```

```
time needed: 9.5367431640625e-07
```

[46]:
```
9
```

1.5.2 Decorator

There is another way, just syntactical sugar, to make this automatic: a decorator. It is invoked as below

[47]:
```
@timed_func
def add(x, y):
    return x + y
```

Again, as for the contextmanager, we can also use a class here to give more flexibility and create a decorator that takes *arguments*.

1.6 ADVANCED ONLY: contect-manager, function decorators and yield all together

One "easy" way to create a context manager is to have a function that has a `yield` and decorate it with `@contextlib.contextmanager`

[48]:
```
# ADVANCED ONLY
import contextlib

@contextlib.contextmanager
def printer(x):
    print(f'we just entered the context manager and will yield {x}')
    yield x
    print(f'Finishing the context manager, exiting')
```

[49]:
```
with printer(5) as number:
    print(f"we're inside, with number={number}")
print("left manager")
```

```
we just entered the context manager and will yield 5
we're inside, with number=5
Finishing the context manager, exiting
left manager
```

1.7 Exceptions

Exceptions are used to stop the execution at a certain point and surface to higher stacks in the code, e.g. to go up in the call stack. A typical use-case is when an error is encountered, such as the wrong type of object is given. Exceptions can also be caught in a `try ... except ...` block in order to handle the exception.

There are a few built-in exceptions, the most common ones are: - `TypeError`: object has the wrong type, e.g. string instead of float - `ValueError`: the value of the object is illegal, e.g. negative but should be positive - `RuntimeError`: if a function is illegally executed or a status is wrong. E.g. if

an object first has to be loaded before it gets parsed. It covers any error that does not fall into an other category. - `KeyError`, `IndexError`: if a key or index is not available, e.g. in a `dict` or `list`

An Exception can manually be raised by

[50]:
```
raise TypeError("Has to be int, not str")
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[50], line 1
----> 1 raise TypeError("Has to be int, not str")

TypeError: Has to be int, not str
```

Note that it is often convenient to create an instance such as in the example above where the first argument is the message (as we see in the raised Exception above), but we can also raise an exception by only using the class itself

[51]:
```
raise TypeError
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[51], line 1
----> 1 raise TypeError

TypeError:
```

1.7.1 Custom Exception

In Python, exceptions are simply a class. And as such, we can inherit from it and create our own exception.

Attention: inherit from `Exception` or subclasses of it such as `TypeError`, `ValueError`, but NEVER from `BaseException`.

[52]:
```
class MyError(Exception):
    pass
```

[53]:
```
raise MyError("Hello world")
```

```
---------------------------------------------------------------------------
MyError                                   Traceback (most recent call last)
Cell In[53], line 1
----> 1 raise MyError("Hello world")

MyError: Hello world
```

An exception can also be created by inheriting from an already existing exception if it is more specific and provides hints on the nature of the exception.

[54]:
```
class NegativeValueError(ValueError):
    pass
```

1.7.2 Catching exceptions

An exception can be caught in a `try..except` block. This works as follows: - if an exception is raised in the `try` block, the next `except` is invoked - it is tested whether the raised exception is of type subclass of the exception type specified to be caught. For example, `except TypeError` checks if the raised error is of type `TypeError` or a subclass of it. - if that is not the case, it goes to the next `except` statement (yes, there can be multiple) - … more below

[55]:
```
try:
    raise NegativeValueError("Negative value encountered")
except ValueError as error:
    print(f"Caught {error}")
```

```
Caught Negative value encountered
```

By using the `as` keyword, the error that is raised is assigned to a variable. We can inspect the error now if we want or, as above, just print it.

If no error is specified, *any* error is caught (this should NOT be used, except for special cases)

[56]:
```
try:
    raise TypeError
# Anti-pattern, do NOT use in general!
except:  # any exception if not specified
    pass
```

[57]:
```
try:
    raise TypeError("Type was wrong, unfortunately")




except TypeError as error:  # any exception
    print(f'caught TypeError: {error}')
except ValueError as error:
    print(f'caugth ValueError: {error}')
```

```
caught TypeError: Type was wrong, unfortunately
```

To continue from above: after the last `except`, an `else` statement is looked for. The `else` is executed if *no* exception was raised.

[58]:
```
# comment and uncomment
try:
```

```
#     print('no error raised')
    raise TypeError("Type was wrong, unfortunately")
#     raise ValueError("Value was wrong, ")
except TypeError as error:  # any exception
    print(f'caught Type {error}')
except ValueError as error:
    print(f'caugth Value: {error}')
else:
    print("No error was caught")

print("Executed after block")
```

```
caught Type Type was wrong, unfortunately
Executed after block
```

...and finally, after the else, a `finally` block is looked for. This is *guaranteed* to be executed! Whether an exception is raised, whether it is caught or not, whether there is an `else` or not, the `finally` is *always* executed.

Therefore it is suitable for any cleanup code such as closing files, removing temporary files and more.

[59]:
```
try:
    #     pass
    #     raise TypeError("Type was wrong, unfortunately")
    raise RuntimeError("Type was wrong, unfortunately")
except TypeError as error:  # any exception
    print(f'caught Type {error}')
except ValueError as error:
    print(f'caugth Value: {error}')
else:
    print("No error was caught")
finally:  # POWERFUL! Guarantied to be executed
    print('Finally run')
print("Executed when passed")
```

```
Finally run
```

```
---------------------------------------------------------------------------
RuntimeError                              Traceback (most recent call last)
Cell In[59], line 4
      1 try:
      2     #     pass
      3     #     raise TypeError("Type was wrong, unfortunately")
----> 4     raise RuntimeError("Type was wrong, unfortunately")
      5 except TypeError as error:  # any exception
      6     print(f'caught Type {error}')
```

```
RuntimeError: Type was wrong, unfortunately
```

Note that in the above example, the error was *not* caught! All the other statements could also be omitted and only a `try...finally` block can be created.

Typical usecase: cleanup (temporary file removal)

[60]:
```
try:
    raise ValueError
finally:
    print('raised')
```

```
raised
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[60], line 2
      1 try:
----> 2     raise ValueError
      3 finally:
      4     print('raised')

ValueError:
```

1.7.3 pitfall "guaranteed execution"

As the `finally` is guaranteed to be executed, this can have an odd effect: possible return statements can be ignored *before the finally* **IF** the `finally` also has a return statement. The logic says here that the `finally` return *must* be executed, as it is guaranteed to be executed.

[61]:
```
def func(x):
    try:
        if x == 5:
            raise RuntimeError('called inside func')
    except RuntimeError as error:
        return error
    else:
        print('else before 42')
        return 42
        print('after else 42')
    finally:
        print("cleaned up")
        return 11
```

[62]:
```
result = func(6)
print(f"Result: {result}")
```

```
else before 42
cleaned up
Result: 11
```

[63]:
```python
try:
    raise ValueError
except ValueError as error:
    print("raising")
    raise RuntimeError from None
finally:
    raise TypeError
    print("finally")
print("Unreachable")
```

```
raising
```

```
---------------------------------------------------------------------------
RuntimeError                              Traceback (most recent call last)
Cell In[63], line 5
      4     print("raising")
----> 5     raise RuntimeError from None
      6 finally:

RuntimeError:

During handling of the above exception, another exception occurred:

TypeError                                 Traceback (most recent call last)
Cell In[63], line 7
      5     raise RuntimeError from None
      6 finally:
----> 7     raise TypeError
      8     print("finally")
      9 print("Unreachable")

TypeError:
```