

# 06 Quantities

## 0.1 Quantities

Sets are like lists, but with the difference that they can contain each element only once. On the other hand (and not only on the other hand, but also technically) they are like dictionaries. They can only contain elements that are immutable, and we can very quickly determine whether a set contains a certain element or not, in the same way as with dictionaries, we can quickly find out whether they contain a certain key or not.

Sets are written with curly brackets, just as we are used to in mathematics.

```
danasnji_klic1 = {"Ana", "Cilka", "Eva"}
```

This can only produce a non-empty set. If we just write a parenthesis and a bracket, {}, we get a dictionary. (Why did they decide to make it a dictionary, not a set? The dictionary came first, Python got sets later. Hence. Besides, we really need dictionaries a lot and sets much less often.)

If we want to make an empty set, we say:

```
prazna = set()
```

The "function" set is a bit like the "function" int: you can give it different arguments and it will turn them into a set. You can give it, say, a list, and you'll get a set with all the elements that appear in it.

```
set([1, 2, 3])
```

```
{1, 2, 3}
```

```
set(range(5))
```

```
{0, 1, 2, 3, 4}
```

```
set([6, 42, 1, 3, 1, 1, 6])
```

```
{1, 3, 6, 42}
```

In addition to lists, we can pass anything to the sets that could be passed through a for loop, say a string or a dictionary.

```
set("Benjamin")
```

```
{'B', 'a', 'e', 'i', 'j', 'm', 'n'}
```

```
set(stevilke)
```

```
-----
NameError                                Traceback (most recent call last)
Cell In[7], line 1
----> 1 set(stevilke)

NameError: name 'stevilke' is not defined
```

The variable stems (still) contain a dictionary whose keys are the names of Benjamin's female fans.

Since the loop "returns" the keys via the dictionary, the set constructed from the dictionary will also contain the keys.

We can add elements to the set, and we can ask whether the set contains a particular element.

```
s = set("Benjamin")
```

```
"e" in s
```

```
"u" in s
```

```
s.add("u")
```

```
"u" in s
```

```
s.add("a")
```

```
s.add("a")
```

```
s.add("a")
```

```
s
```

Finally, we tried to add an element to the set that already contains it. This is not possible, of course, because the set contains each element only once.

If we have two sets, we can calculate their union, intersection, difference (elements that appear in the first set but not in the second set) and symmetric difference (elements that appear in one set and not in the other) ...

```
u = {1, 2, 3}
```

```
v = {3, 4, 5}
```

```
u | v
```

```
u & v
```

```
u - v
```

```
u ^ v
```

Just as we can use += to add numbers (and write  $x += a$  instead of  $x = x + a$ ), and, similarly, subtract, multiply and divide with -=, \*= and /=, we can add, subtract or divide a set with &=, |= and -=. So, for example,  $u \&= v$  is the same as  $u = u \& v$ .

We can also check whether a set is a subset (or superset) of another set. The simplest way to do this is to use the comparison operators.

```
u = {1, 2, 3}
```

```
{1, 2} <= u
```

```
{1, 2, 3, 4} <= u
```

```
{1, 2, 3} <= u
```

```
{1, 2, 3} < u
```

{1, 2, 3} is a subset of u, but it is not a proper subset, since it contains the whole of u.

There are many other interesting things you can do with sets - but enough is enough. We'll look at more examples as we go along.

# 06 Modules

## 1 Modules

We've said it before: Python has thousands and thousands of functions, so they need to be organised, placed where we can find them.

One of the principles we've already learned: methods. Methods are attached to things (more learned and correct: objects) and do things that are typical of those things. Arrays have methods like `lower` and `split`, dictionaries have `get` and `setdefault`.

In addition to methods, Python has functions that do not belong to any specific type. Examples are `input` and `print`. But these are few; only a few functions have the privilege of lying idle.

Most functions are boxed. In Python, boxes are called modules. You'll probably also hear the terms `package` and even `library`. Don't worry about them: a package is a hierarchical collection of modules, and a library is some rounded collection of different ... modules,

packages ... whatever. For us here, only modules are important. We'll find out how to use them and see some useful things in some useful modules.

### 1.1 Importing modules

To use the functions in a module, we first need to import the module.

So far, for example, we have been using functions from the `math` module and importing them with `from math import`

\*. When I teach programming, I avoid having to type in some phrases that you don't understand. `from math import *` was one of the few, and it's time to find out what it's all about. It's also time to get rid of the phrase and learn to write it properly. The `math` functions module, `math`, is properly imported like this:

```
import math
```

```
math
```

```
<module 'math' from '/Users/janez/opt/miniconda3/envs/prog/lib/python3.11/lib-dynload/math.cpython-311-darwin.so'>
```

This is the strangest type of variable ever. It is not simply a number (`int`, `float`) or a string (`str`) or a list or a tuple (`list`, `tuple`), but a variable of type `"module"`.

Functions belong to a module, just like methods belong to an object. If we have a string name, we can access its upper method with a dot, `name.upper`. Here it is similar: if we have a module `math`, we will access its function `sqrt` with `math.sqrt`, and its variable `pi` with `math.pi`.

```

math.pi
3.141592653589793

math.sin(math.pi / 4)
0.7071067811865475

Uvozimo še en modul, os.
import os

```

Let's import another module, os.

```
import os
```

There are functions in the axis that can be used to create a directory or delete a file. In addition, it contains something interesting: a module. The path module is a module within the axis module. It contains, for example, the splitext function, which returns a base and an extension for a given filename.

```

osnova, koncnica = os.path.splitext("nek_film.avi")

osnova
'nek_film'

koncnica
'.avi'

```

This is how modules are imported.

### 1.1.1 Importing individual functions

```
x = 2 * math.sin(math.radians(phi) + math.pi/2) - 2 * math.cos(math.radians(alpha))
```

It would be better to have these functions at hand without repeating math. Let's import them with

```

from math import sin, cos, radians, pi

cos(pi / 4)
0.7071067811865476

```

from math import sin, cos, radians, pi imports the functions listed and makes them available to the program under those names. We will therefore have the names sin, cos, radians and pi, but not tan and log.

But especially not math. That is:

- if we import import math, we have math and, say, math.cos, but not cos. We have imported math, not cos.

- If we import from math import cos, we have cos and not math or math.cos. We have imported cos and not math.

### 1.1.2 Importing everything

There is an even "simpler" way of importing.

```
from math import *
```

This one is similar to the previous one, except that we don't list the functions. \* represents everything in the module.

The first way, importing the module, has the advantage that in each function call it is clear where the function comes from. The disadvantage is that the expressions are much longer.

The second way, importing individual functions, shortens the expressions. We can still see where a function comes from, but we have to look at the beginning of the program, in the import.

The third way is generally bad. The expressions are shorter, but for single functions we don't know where they came from. This way we import at most the math module and nothing else. Especially in larger projects.

Amendment: I wrote the above paragraph ten years ago. In fact, I haven't imported a module in this way for a long, long time, except in the first weeks of Programming 1. It's not done.

## 1.2 Where to import

Always at the beginning of the program. Not just in between.

Exception: at the beginning of a function. This is done, for example, when it is not clear whether a module can be imported, or when importing is slow, and it is only needed in a function. Another situation where this would come in handy is when two modules are imported mutually, one after the other. This is resolved by importing a module only when it is needed.

This is not a law, just an agreement.

Importing modules only happens once in reality. If we write `import math` five times, the module will only actually be imported the first time.

## 1.3 Maths module

We will look at some useful modules. Python has about 200 of them; you can find thousands and thousands more on the Internet.

The first is math, but we won't say much about it. It has all the functions that any calculator has ... and a bit more. :) For example, greatest common divisor, faculty and binomial coefficients

...

Let's just take this opportunity to mention two features related to numbers. The float data type has two special features in addition to the normal numbers: infinity (and minus infinity) and not a number.

```
math.inf
```

```
inf
```

```
-math.inf
```

```
-inf
```

```
math.nan
```

```
nan
```

The first two can be useful for finding a minimum or maximum.

```

def min(s):
    m = math.inf
    for x in s:
        if x < m:
            m = x
    return m

```

`math.inf` is greater than all numbers (except itself) and therefore a useful starting value for finding the minimum.

`nan` is more interesting. In some languages it is obtained as a result of wrong arguments to mathematical functions, say when trying to calculate the root or logarithm of a negative number. Python functions return an error in this case, but it is returned by some functions in libraries (modules) that we will install additionally, so it is good to know about it.

`nan`, not a number, is a dead end number. Whatever we do with it - adding, subtracting, multiplying, even multiplying by zero - it always remains `nan`. Worse: `nan` is neither bigger nor smaller than any number. It is not smaller than infinity (and of course not greater, but not equal either).

```
math.nan < math.inf
```

False

```
math.nan > math.inf
```

False

```
math.nan == math.inf
```

False

And, worst of all, he is not even like himself.

```
math.nan == math.nan
```

False

How do we then check if a variable has the value `nan` - if we can't even compare it with `nan`?

```

x = math.nan

if x == math.nan:
    print("Ojoj!")
else:
    print("Vse je OK.")

```

Vse je OK.

Rather, let us use his stupidity against himself.

```

x = math.nan

if x != x:
    print("Svet se podira!")

```

Svet se podira!

A nicer, more correct way to check whether a variable has the value `nan` is the function `math.isnan`.

```
if math.isnan(x):  
    print("Ojoj!")
```

Ojoj!

It will be useful to remember all this when we work with, for example, some statistical libraries that will return nan when something cannot be calculated.

By the way, this strange behaviour, where nan is not even equal to itself, is not some Python fad, but part of the IEEE 754 standard that defines floating point notation and its behaviour. Every language that has been trained behaves the same way.

## 1.4 The random module

The random module contains functions that do (pseudo-)random things. (That is: they look random, but they are not, because everything that is computed by ordinary computers is computed, not drawn.) We will mention just a few. `random.random()` returns a random number between 0 and 1.

```
import random  
  
random.random()
```

0.6494175036385138

```
random.random()
```

0.7117311287907189

`random.uniform(a, b)` returns a random number between a and b. The function is called uniform because it is a uniform distribution - all numbers are equally likely.

```
random.uniform(10, 20)
```

19.118192660781837

```
random.uniform(10, 20)
```

17.985517474304324

If we need a random integer, we call `randint`.

```
random.randint(10, 20)
```

15

If we are interested in some other distribution than uniform: the random module knows many: beta, gamma, exponential ... and of course Gaussian.

```
: iq = random.gauss(100, 10)
iq
: 98.30080171519482
```

Let's make a list of names.

```
imena = ["Ana", "Berta", "Cilka", "Dani", "Ema", "Fanči"]
```

random.choice returns a random element of the given list.

```
random.choice(imena)
```

```
'Cilka'
```

random.sample selects a random sample of the given size.

```
random.sample(imena, 3)
```

```
['Fanči', 'Ema', 'Ana']
```

random.choices is similar, but the selected cases can also be repeated. The sample size must be given as an argument named k.

```
random.choices(imena, k=5)
```

```
['Berta', 'Berta', 'Ana', 'Dani', 'Cilka']
```

random.shuffle shuffles the given list.

```
imena
```

```
['Ana', 'Berta', 'Cilka', 'Dani', 'Ema', 'Fanči']
```

```
random.shuffle(imena)
```

```
imena
```

```
['Ana', 'Dani', 'Cilka', 'Ema', 'Fanči', 'Berta']
```

## 1.5 module os

The os module contains a bunch of stuff related to the operating system. Because we don't know enough about operating systems, we wouldn't understand most of the features.

We will use them regularly anyway.

- getcwd() returns the current directory. That is, the directory in which the open function would look for the file if given only the filename.

- chdir(path) changes the current directory. The path can be absolute (with / at the beginning) or relative.

- mkdir(path) makes a new directory.

- remove(filename) deletes the file with the given name. No mercy. No "do you really want to delete".



- `rename(name, newname)` renames the file.
- `listdir(path)` returns a list of all filenames in the given directory.

Most often you will need the latter. For example, I got weather station data in the form of thousands of files, which I then read and wrote the (filtered, of course) data to a new file.

This is the current contents of the current directory (where these records are).

```
import os
```

```
os.listdir()
```

```
['progkog 10-24.ipynb',  
 'Untitled1.ipynb',  
 'kolesa.txt',  
 '.DS_Store',  
 'kolesa2.txt',  
 '03c Slovarji.ipynb',  
 'temperature.txt',  
 '04 seznam1.ipynb',  
 'vreme',  
  
 '03a Kako racunalnik shrani besedilo.ipynb',  
 '05 Moduli.ipynb',  
 '02b logični izrazi.ipynb',  
 '.ipynb_checkpoints',  
 '02 datoteke, zanke, pogoji.ipynb',  
 'december.txt',  
 '03b Metode nizov.ipynb']
```

## 1.6 The `os.path` module

`path` is a module within the `os` module. It also has many features, some of which are suitable for us.

- `os.path.exists(name)` returns `True` if a file or directory with the given name exists (in the current directory).
- `os.path.isdir(name)` returns `True` if the given directory name exists (within the current directory).
- `os.path.isfile(name)` returns `True` if the given name is a file (within the current directory).
- `os.path.splitext(name)` returns the base and extension of the given filename.

The others are obvious, but the last one is particularly interesting for us.

Let's list the names of all files with extension `.txt` that are located in the current directory.

```
for ime in os.listdir():
    osnova, koncnica = os.path.splitext(ime)
    if os.path.isfile(ime) and koncnica == ".txt":
        print(ime)
```

```
kolesa.txt
kolesa2.txt
temperature.txt
december.txt
```

## 1.7 Collections module

Among all the interesting things in collections, let's mention just two, at least for now.

### 1.7.1 defaultdict

defaultdict is a dictionary that adds non-existent keys on the fly, and we need to give it a function to invent their values. Only functions that do not accept arguments are suitable, or more precisely, functions that can be called (even) without arguments. Most often, this will be int. If called without arguments, it returns 0.

```
int()
```

```
0
```

```
from collections import defaultdict
```

```
d = defaultdict(int)
```

```
d["Ana"] = 5
d["Berta"] = 3
```

```
d
```

```
defaultdict(int, {'Ana': 5, 'Berta': 3})
```

```
d["Cilka"]
```

```
0
```

```
d
```

```
defaultdict(int, {'Ana': 5, 'Berta': 3, 'Cilka': 0})
```

We can even, say, increase the value of a non-existent key.

```
: d["Dani"] += 1
```

```
d
```

```
: defaultdict(int, {'Ana': 5, 'Berta': 3, 'Cilka': 0, 'Dani': 1})
```

Let's count how many times the author of the bicycle.txt file has ridden which bicycle and how far he has travelled on it. Let's remember: the lines of the file represent individual journeys and contain the name of the bike, the distance in kilometres and something else that we are not interested in here (the height).

```

uporaba = defaultdict(int)
pot = defaultdict(int)

for vrstica in open("kolesa.txt"):
    kolo, razdalja, _ = vrstica.split(",")
    uporaba[kolo] += 1
    pot[kolo] += int(razdalja)

print(uporaba)
print(pot)

```

```

defaultdict(<class 'int'>, {'Nakamura': 22, 'Cube': 43, 'Canyon': 26, 'Stevens': 9})
defaultdict(<class 'int'>, {'Nakamura': 439, 'Cube': 3174, 'Canyon': 2766, 'Stevens': 607})

```

Uporaba = use

pot = way

razdalja = distance

You could do the same with normal dictionaries, but it would require some if statements or methods like setdefault or get.

Now let's turn to the auction: we want to build a dictionary whose keys will be objects, the values of which will be the bid lists for that object. Again we will use defaultdict, but the values will not be ints, but lists. The list function, if called without arguments, conveniently returns an empty list.

```

d = defaultdict(list)

d["foo"]

```

```
[]
```

We can even add to such a list with append!

```

d["bax"].append(12)
d["bax"].append(5)

```

```
d
```

```
defaultdict(list, {'foo': [], 'bax': [12, 5]})
```

Now for real.

```

ponudbe = defaultdict(list)

for vrstica in open("../domace-naloge/03-drazba-brez-anonimnosti/zapisnik.txt"):
    predmet, oseba, cena = vrstica.split(",")
    ponudbe[predmet].append(int(cena))

for predmet, cene in ponudbe.items():
    print(predmet, cene)

```

slika [31, 33, 35, 37, 40, 45]

pozlačen dežnik [29]

Meldrumove vaze [44, 46, 48, 53, 57, 60, 61, 63, 67, 71, 76, 78]

skodelice [50, 55, 60, 61, 62, 65, 68, 70, 74, 76, 80, 83]

kip [30, 32, 37, 39, 43, 44, 45, 50, 53, 55, 58, 61, 63, 68, 72, 76, 77, 81, 85, 86, 90, 92, 94, 97, 98, 99, 100, 103, 107]

čajnik [15]

srebrn jedilni servis [27, 30, 35, 39, 40, 45, 47, 49, 53, 55, 58, 59, 62, 63]

perzijska preproga [16, 21]

Ponudbe = offers  
predmet = subject

### 1.7.2 Counter

For some reason we write Counter with a capital. (The reason is not a very good one. For the same reason, we could also capitalize defaultdict. Counter is related to defaultdict and sometimes replaces it. We don't currently have a good example of its use at hand, or, more accurately, to use it effectively we would need to know something we don't know yet. However, we can show what it does. Let's say we have a list of names of people who have been telephoned.

```
klici = ["Ana", "Ana", "Berta", "Cilka", "Ana", "Cilka", "Dani", "Ema"]
```

Of course, we want to count how many times he called who. We could do this trivially with defaultdict, but with Counter it's even more trivial:

```
from collections import Counter

stevci = Counter(klici)

stevci
```

```
Counter({'Ana': 3, 'Cilka': 2, 'Berta': 1, 'Dani': 1, 'Ema': 1})
```

Stevci = counters

Often we will want to know who he called most often - or which three - and there is a method for this.

```
stevci.most_common(3)

[('Ana', 3), ('Cilka', 2), ('Berta', 1)]
```

This is the list, ordered by frequency.

## 1.8 The csv module

You'll like this one. Files containing comma-separated data, like our minutes.txt, are quite common. This form of notation is called comma separated values or, in abbreviation,

Excel can also save in this format - with the caveat that it will lose all formatting and whatnot. (I once promised that we would learn how to read Excel files. That's not it yet. We will also read .xlsx. But not yet.) Our auction minutes and all the other files were in that format. Python therefore has a csv module that can read such things.

### 1.8.1 reader

Are we interested in the names of all the participants in the auction?

```
import csv

udelezenci = set()
for predmet, oseba, cena in csv.reader(open("../domace-naloge/04-analiza-drazbe/
-zapisnik.txt")):
```

```
udelezenci.add(oseba)

udelezenci
```

```
{'Ana', 'Berta', 'Cilka', 'Dani', 'Ema', 'Fanči', 'Greta', 'Helga'}
```

Give the `csv.reader` function the file (not just the name, you also need to call `open`). It returns something that can be passed over with a `for` loop, and we get the data from the lines. No `split`.

`csv.reader` defaults to comma separated data. If they are by something else, we tell it that with an additional `delimiter` argument. The `delimiter` in the "wheels.txt" file from the fifth homework assignment was `-`.

```
ime_dat = "../domace-naloge/05-druzabno-omrezje-drazbe/kolesa.txt"

for v in csv.reader(open(ime_dat), delimiter="-"):
    print(v)
```

```
['Cube', '5031', '159', 'Janez', '2017']
['Stevens', '3819', '1284', 'Ana', '2012']
['Focus', '3823', '1921', 'Benjamin', '2019']
```

In addition to punctuation, a file can have a bunch of other properties. Suppose a set of Louis XIV spoon, knife and fork is being sold at auction. In the file, you would get the line `Louis XIV spoon, knife and fork,Ana,12945` and `s.split(",")` would return four things instead of three, just because of the comma between the spoon and the knife. Excel, in this case, would write something like `"Louis XIV's spoon, knife and fork",Ana,12945`

By enclosing the first field in quotation marks, it would say that there is a single thing and that the comma within it should be ignored. Different programs and systems have different rules; this is called a *dialect* in the language of the `csv.reader` function. The default dialect is `"excel"` and this will usually work for Excel files. For others you can use `Sniffer`.

```
sniffer = csv.Sniffer()
```

The `.read()` method reads the entire contents of the file.

```
open(ime_dat).read()

'Cube-5031-159-Janez-2017\nStevens-3819-1284-Ana-2012\nFocus-3823-1921-Benjamin-2019\n'
```

`sniffer` has a `sniff` method, which takes as argument the contents of a file (or at least a chunk big enough to show how the file is formatted). The `sniffer` will guess what style the file is written in.

```
dialect = sniffer.sniff(open(ime_dat).read())
```

Then call `csv.reader` and give it the `dialect` along with the file. To make it easier to see, let's do it all again.

```
import csv

ime_dat = "../domace-naloge/05-druzabno-omrezje-drazbe/kolesa.txt"
sniffer = csv.Sniffer() # vrne novega "snifferja"
dialect = sniffer.sniff(open(ime_dat).read()) # preberemo vsebino datoteke in
# jo damo posniffati :)
for v in csv.reader(open(ime_dat), dialect):
    print(v)
```

```
['Cube', '5031', '159', 'Janez', '2017']
['Stevens', '3819', '1284', 'Ana', '2012']
['Focus', '3823', '1921', 'Benjamin', '2019']
```

## 1.8.2 DictReader

Suppose you had a file like this with wheels:

bike,distance,height,owner,year of purchase

Cube,5031,159,Janez,2017

Stevens,3819,1284,Ana,2012

Focus,3823,1921,Benjamin,2019

Unlike the previous files (and all the ones we have seen so far), this file has column names in the first line. Therefore, instead of using a reader, it can be read with DictReader, which returns not a list but a dictionary for each line, the keys of which are the column names. That's great.

```
import csv

for vrstica in csv.DictReader(open("kolesa-z-glavo.txt")):
    print(vrstica["kolo"], ":", vrstica["leto nakupa"])
```

```
Cube : 2017
Stevens : 2012
Focus : 2019
```

As your files will often have a header, you will mainly use DictReader. This is practical, as dictionaries are easier to work with than lists. Column names such as bike, owner and year of purchase are easier to read than indexes 0, 3 and 4, and you won't make any mistakes when counting.

## 1.9 Statistics module

We will not discuss the statistics module too much. Personally, I never use it, because all this and much more can be found in the numpy library modules, which also make it much easier to do much more - but only if you know how. Python only has this module because numpy is a huge extra library that you don't get with Python. (On the other hand, any serious Python user will install numpy.)

In short: statistics contains mean, median, mode, stdev and a bunch of other functions that can calculate mean, median, mode, standard deviation and a bunch of other things for a given list.

```
: import statistics

visine = [185, 192, 160, 173, 180]

: statistics.mean(visine)

: 178

: statistics.stdev(visine)

: 12.227019260637483
```

He also knows about correlation and linear regression, but that's where it stops. Anyone who wants to know more should consult the statistics module documentation.

## 1.10 Time, datetime, calendar modules

In your work you will probably often come across dates, times and so on. The functions related to these are spread over three modules. It's all a mess. It is not so much Python's fault as, above all, the fact that it refers to various standardised functions of different systems.

Some things work differently on Windows than they do on macOS or Linux, and then we get what we get.

### 1.10.1 time

The time module (documentation) deals with the current time and other times and the conversion between them.

The main function is time, which returns the number of seconds elapsed since 1 January 1970 according to Greenwich.

```
: import time
:
: time.time()
:
: 1699905222.8740501
```

This is useful if you want to measure the (real) time elapsed between two events in the programme.

But ... well, not really. There are better functions for that.

For our purposes, gmtime and localtime are perhaps more useful.

```
print(time.gmtime())
print(time.localtime())
```

```
time.struct_time(tm_year=2023, tm_mon=11, tm_mday=13, tm_hour=19, tm_min=53,
tm_sec=43, tm_wday=0, tm_yday=317, tm_isdst=0)
time.struct_time(tm_year=2023, tm_mon=11, tm_mday=13, tm_hour=20, tm_min=53,
tm_sec=43, tm_wday=0, tm_yday=317, tm_isdst=0)
```

You both return a miracle of the same type. It is called struct\_time. Both the name and the format come from the C language.

The name is not important, what matters are the names of the fields containing the current year, month and day, hour, minutes and seconds, and the day of the week and year, with a field to tell whether it is the time of year (1) or not (0).

```
zdej = time.localtime()
print("Danes je ", zdej.tm_mday, ". ", zdej.tm_mon, ". ", zdej.tm_year, ". ",
      sep="")
```

Danes je 13. 11. 2023.

The module also contains a date formatting function, which can be used as follows:

```
time.strftime("Danes je %d. %m. %Y, ura pa je %H.%M", zdej)
'Danes je 13. 11. 2023, ura pa je 20.59'
```

It is given a string containing characters such as %d, %m, %Y and so on, and a time. The function will replace these characters with the corresponding values.

The full list of possible characters can be found at <https://docs.python.org/3/library/time.html#time.strftime>.

```
time.strftime("Danes je %A, %d. %B %Y", zdaj)

'Danes je Monday, 13. November 2023'
```

This one didn't turn out so well. Apparently he didn't choose Slovenian. He could have been prepared for this.

```
import locale

locale.setlocale(locale.LC_ALL, "sl_SI")

'sl_SI'
```

Now it will be.

```
time.strftime("Danes je %A, %d. %B %Y", zdaj)

'Danes je ponedeljek, 13. november 2023'
```

We've also used the locale module here, which contains everything related to the different usages of different languages - from the names of days and months, to whether they use a decimal point or a period, to how currencies are written.

```
locale.currency(42.19)

'42,19 SIT'
```

### 1.10.2 datetime

The core of the datetime (documentation) module is date manipulation. It allows you to subtract two times and find out how many years, months, days, hours, minutes, seconds there are between them. Or you can create a timedelta object, which will be, say, two months and three days, and add that to a date. More importantly: the module can convert times from strings.

```
from datetime import datetime

datetime.strptime("13. november 2023, 21:14", "%d. %B %Y, %H:%M")

datetime.datetime(2023, 11, 13, 21, 14)
```

First, a strange import: we import a "function" datetime from the datetime module. (It's not really a function, it's a type, but there's no time for that right now. :)

The strptime function requires a string containing the date, and a string telling the format of the string - again in letters, as we saw in strftime. Parts of the result are again accessed by field names.



```
cas = datetime.strptime("13. november 2023, 21:14", "%d. %B %Y, %H:%M")
```

```
cas.day
```

```
13
```

```
cas.hour
```

```
21
```

This will be able to break up the date for you in any format you like. Here's how we do it with the Americans

```
datetime.strptime("11/13/2023", "%m/%d/%Y")
```

```
datetime.datetime(2023, 11, 13, 0, 0)
```

And so it is with the Americans, who are writing a year without centuries.

```
datetime.strptime("11/13/23", "%m/%d/%y")
```

```
datetime.datetime(2023, 11, 13, 0, 0)
```

### 1.10.3 calendar

The last of the three time-related modules is the simple calendar (documentation). This contains things like names of days, months and so on.

## 1.11 The urllib module

(In progress. To be added. Sooner or later. Maybe. Probably.)