

Errors

Error's so napake v programu, ki nam ponavadi zaustavijo izvajanje programa.

Klasificiramo jih v:

- Snytaks errors
- Runtime errors
- Logical errors

Syntax errors

Syntax errors so napake pri uporabi Python jezika.

Python bo našel te napake med parsanjem našega programa. Če najde takšno napako bo exit-u brez, da bi pognal ta del kode.

Najbolj pogoste Syntax napake so:

- izpuštitvev keyword
- uporaba keyword na napačnem mestu
- izpuštitvev simbolov, kot je :
- napačno črkovanje
- napačen indentation

```
In [1]: # Primer: manjka keyword def
myfunction(x, y):
    return x + y

File "<ipython-input-1-8b32d31d1203>", line 2
    myfunction(x, y):
          ^
SyntaxError: invalid syntax
```

```
In [2]: else:
    print("Hello!")

File "<ipython-input-2-429811f9164b>", line 1
    else:
        ^
SyntaxError: invalid syntax
```

```
In [3]: # Primer: manjka :
if mark >= 50
    print("You passed!")

File "<ipython-input-3-2bfd10af2cba>", line 2
    if mark >= 50
        ^
SyntaxError: invalid syntax
```

```
In [4]: # Primer: napačno črkovanje "else"
if arriving:
    print("Hi!")
esle:
    print("Bye!")

File "<ipython-input-4-1cca186d8b5e>", line 4
    esle:
        ^
SyntaxError: invalid syntax
```

```
In [5]: # Primer: napačen indentation
if flag:
    print("Flag is set!")

File "<ipython-input-5-2009e1311970>", line 3
    print("Flag is set!")
    ^
IndentationError: expected an indented block
```

Runtime errors

Primer runtime errors:

- Deljenje z 0
 - Dostopanje do elementov, ki ne obstajajo
 - Dostopanje do datotek, ki ne obstajajo
- division by zero
- performing an operation on incompatible types
 - using an identifier which has not been defined
 - accessing a list element, dictionary value or object attribute which doesn't exist
 - trying to access a file which doesn't exist

```
In [6]: # Primer: deljenje z 0
1 / 0

-----
ZeroDivisionError                                 Traceback (most recent call last)
<ipython-input-6-eca1cc1fcdbe> in <module>
      1 # Primer: deljenje z 0
----> 2 1 / 0

ZeroDivisionError: division by zero
```

Logical errors

Logične napake nam povzročijo napačne rezultate. Program je lahko sintaksično pravilno zapisan ampak nam ne bo vrnil iskanega rezultata.

Primeri

- Uporabna napačne spremenljivke
- napačna indentacija
- uporaba celoštevilskega deljenja in ne navadnega deljenja

Logical errors are the most difficult to fix. They occur when the program runs without crashing, but produces an incorrect result. The error is caused by a mistake in the program's logic. You won't get an error message, because no syntax or runtime error has occurred. You will have to find the problem on your own by reviewing all the relevant parts of your code – although some tools can flag suspicious code which looks like it could cause unexpected behaviour.

```
In [ ]:
```

The try and except statements

Da obvladujemo morebitne napake uporabljamo try-except:

```
In [7]: for _ in range(3):
    x = int(input("Vnesi prvo številko: "))
    y = int(input("Vnesi drugo številko: "))
    rezultat = x / y
    print(f"{x}/{y} = {rezultat}")
    print()

Vnesi prvo številko: 1
Vnesi drugo številko: 2
1/2 = 0.5

Vnesi prvo številko: a

-----
ValueError                                 Traceback (most recent call last)
<ipython-input-7-b27c485d0cd1> in <module>
      1 for _ in range(3):
      2     x = int(input("Vnesi prvo številko: "))
      3     y = int(input("Vnesi drugo številko: "))
      4     rezultat = x / y
      5     print(f"{x}/{y} = {rezultat}")

ValueError: invalid literal for int() with base 10: 'a'
```

```
In [1]: for _ in range(3):
    x = int(input("Vnesi prvo številko: "))
    y = int(input("Vnesi drugo številko: "))
    rezultat = x / y
    print(f"{x}/{y} = {rezultat}")
    print()

Vnesi prvo številko: 1
Vnesi drugo številko: 2
1/2 = 0.5

Vnesi prvo številko: 1
Vnesi drugo številko: 0

-----
ZeroDivisionError                           Traceback (most recent call last)
Input In [1], in <cell line: 1>()
      2 x = int(input("Vnesi prvo številko: "))
      3 y = int(input("Vnesi drugo številko: "))
----> 4 rezultat = x / y
      5 print(f"{x}/{y} = {rezultat}")
      6 print()

ZeroDivisionError: division by zero
```

Ko se zgodi napaka, Python preveri ali se naša koda nahaja znotraj **try** bloka. Če se ne nahaja, potem bomo dobili error in izvajanje programa se bo ustavilo.

Če se nahaja znotraj try-except blocka, se bo izvedla koda znotraj **except** bloka in program bo nadaljeval z izvajanjem.

```
In [8]: for _ in range(3):
    try:
        x = int(input("Vnesi prvo številko: "))
        y = int(input("Vnesi drugo številko: "))
        rezultat = x / y
        print(f"{x}/{y} = {rezultat}")
    except:
        print("Prislo je do napake!")
    print()

Vnesi prvo številko: 1
Vnesi drugo številko: 0
Prislo je do napake!

Vnesi prvo številko: 1
Vnesi drugo številko: a
Prislo je do napake!

Vnesi prvo številko: 1
Vnesi drugo številko: 2
1/2 = 0.5
```

In []:

Če se je napaka zgodila znotraj funkcije in znotraj funkcije ni bila ujeta (ni bila znotraj try-except bloka), potem gre Python preverjati ali se klic te funkcije nahaja znotraj try-except bloka.

```
In [9]: def delilnik():
    try:
        x = int(input("Vnesi prvo številko: "))
        y = int(input("Vnesi drugo številko: "))
        rezultat = x / y
        print(f"{x}/{y} = {rezultat}")
    except:
        print("Prislo je do napake!")

for _ in range(3):
    delilnik()
    print()
```

Vnesi prvo številko: a
Prislo je do napake!

Vnesi prvo številko: a
Prislo je do napake!

Vnesi prvo številko: 1
Vnesi drugo številko: 2
1/2 = 0.5

```
In [10]: def delilnik():
    x = int(input("Vnesi prvo številko: "))
    y = int(input("Vnesi drugo številko: "))
    rezultat = x / y
    print(f"{x}/{y} = {rezultat}")

for _ in range(3):
    try:
        delilnik()
    except:
        print("Prislo je do napake!")
    print()
```

Vnesi prvo številko: 1
Vnesi drugo številko:
Prislo je do napake!

Vnesi prvo številko: s
Prislo je do napake!

Vnesi prvo številko: 1
Vnesi drugo številko: 0
Prislo je do napake!

In []:

Naloga:

Napišite funkcijo **fakulteta**, ki uporabnika vpraša naj vnese cifro in izračuna fakulteto te cifre. Fakulteta se izračuna: $3! = 3 \cdot 2 \cdot 1 = 6$

Funkcija naj vrne rezultat. Oziroma, če uporabik ni vnesel številke naj funkcija ponovno zahteva od uporabnika vnos cifre.

INPUT:
`print(fakulteta())`

OUTPUT:
Vnesi cifro: a
To ni bila številka.
Vnesi cifro: b
To ni bila številka.
Vnesi cifro: 3
6

```
In [5]: def fakulteta():
    while True:
        try:
            num = int(input("Vnesi cifro: "))
            rezultat = 1
            for i in range(1, num+1):
                #print(i)
                rezultat *= i
            return rezultat
        except:
            print("To ni bila številka.")

print(fakulteta())
```

Vnesi cifro: 5
120

In []:

Handling an error as an object

```
In [12]: def delilnik():
    try:
        x = int(input("Vnesi prvo številko: "))
        y = int(input("Vnesi drugo številko: "))
        rezultat = x / y
        print(f"{x}/{y} = {rezultat}")
    except:
        print("Prislo je do napake!")

for _ in range(3):
    delilnik()
    print()
```

Vnesi prvo številko: a
Prislo je do napake!

Vnesi prvo številko: 1
Vnesi drugo številko: 0
Prislo je do napake!

Vnesi prvo številko: 1
Vnesi drugo številko: 2
1/2 = 0.5

Tako kot sedaj hendlamo error ne dobimo nobenega podatka o errorju nazaj. Ne vemo zakaj je prišlo do napake in do kakšne napake je prišlo.

```
In [13]: def delilnik():
    try:
        x = int(input("Vnesi prvo številko: "))
        y = int(input("Vnesi drugo številko: "))
        rezultat = x / y
        print(f"{x}/{y} = {rezultat}")
    except Exception as e:
        print("Prislo je do napake!")
        print(type(e))
        print(e)

for _ in range(3):
    delilnik()
    print()
```

Vnesi prvo številko: 1
Vnesi drugo številko: a
Prislo je do napake!
<class 'ValueError'>
invalid literal for int() with base 10: 'a'

Vnesi prvo številko: 1
Vnesi drugo številko: 0
Prislo je do napake!
<class 'ZeroDivisionError'>
division by zero

Vnesi prvo številko: 2
Vnesi drugo številko: 1
2/1 = 2.0

Handling different errors differently

```
In [14]: def delilnik():
    try:
        x = int(input("Vnesi prvo številko: "))
        y = int(input("Vnesi drugo številko: "))
        rezultat = x / y
        print(f"{x}/{y} = {rezultat}")
    except Exception as e:
        print("Prislo je do napake!")
        print(type(e))
        print(e)

    for _ in range(3):
        delilnik()
        print()
```

```
Vnesi prvo številko: 1
Vnesi drugo številko: a
Prislo je do napake!
<class 'ValueError'>
invalid literal for int() with base 10: 'a'

Vnesi prvo številko: 1
Vnesi drugo številko: 0
Prislo je do napake!
<class 'ZeroDivisionError'>
division by zero

Vnesi prvo številko: 2
Vnesi drugo številko: 3
2/3 = 0.6666666666666666
```

V našem primeru sedaj hendlamo katerikoli **Exception** na enak način.

Lahko pa različne errorje hendlamo na različni način.

Preprosto dodamo še en except stavek.

```
In [15]: def delilnik():
    try:
        x = int(input("Vnesi prvo številko: "))
        y = int(input("Vnesi drugo številko: "))
        rezultat = x / y
        print(f"{x}/{y} = {rezultat}")
    except ValueError as e:
        print("Obe spremenljivki morata biti številki!")
        print(type(e))
        print(e)
    except ZeroDivisionError as e:
        print("Druga številka ne sme biti 0!")
        print(type(e))
        print(e)

    for _ in range(3):
        delilnik()
        print()
```

```
Vnesi prvo številko: 1
Vnesi drugo številko: a
Obe spremenljivki morata biti številki!
<class 'ValueError'>
invalid literal for int() with base 10: 'a'

Vnesi prvo številko: 1
Vnesi drugo številko: 0
Druga številka ne sme biti 0!
<class 'ZeroDivisionError'>
division by zero

Vnesi prvo številko: 2
Vnesi drugo številko: 1
2/1 = 2.0
```

V primeru napake bo Python preveril vsak `except` od vrha navzdol, če se tipa napaki ujemata. Če se napaka ne ujema z nobenim `except` potem bo program crashnu.

Če se ujemata bo pa `except` pohendlu error. `Except` pohendla errorje tega razreda in vse, ki dedujejo iz tega razreda.

except clause that mentions a particular class, that clause also handles any exception classes derived from that class (but not exception classes from which it is derived).

(<https://docs.python.org/3/library/exceptions.html> (<https://docs.python.org/3/library/exceptions.html>))

(Se pokaže kako ma Python zgrajeno hierarhijo dedovanja Errorjev). Se pravi, če damo kot prvi except except Exception bomo z njim prestregli vse, ker vsi dedujejo iz tega classa.

BaseException

- SystemExit
- KeyboardInterrupt
- GeneratorExit
- Exception
 - StopIteration
 - StopAsyncIteration
 - ArithmeticError
 - FloatingPointError
 - OverflowError
 - ZeroDivisionError
 - AssertionException
 - AttributeError
 - BufferError
 - EOFError
 - ImportError
 - ModuleNotFoundError
 - LookupError
 - IndexError
 - KeyError
 - MemoryError
 - NameError
 - UnboundLocalError
 - OSError
 - BlockingIOError
 - ChildProcessError
 - ConnectionError
 - BrokenPipeError
 - ConnectionAbortedError
 - ConnectionRefusedError
 - ConnectionResetError
 - FileExistsError
 - FileNotFoundError
 - InterruptedError
 - IsADirectoryError
 - NotADirectoryError
 - PermissionError
 - ProcessLookupError
 - TimeoutError
 - ReferenceError
 - RuntimeError
 - NotImplemented
 - RecursionError
 - SyntaxError
 - IndentationError
 - TabError
 - SystemError
 - TypeError
 - ValueError
 - UnicodeError
 - UnicodeDecodeError
 - UnicodeEncodeError
 - UnicodeTranslateError
 - Warning
 - DeprecationWarning
 - PendingDeprecationWarning
 - RuntimeWarning
 - SyntaxWarning
 - UserWarning
 - FutureWarning
 - ImportWarning
 - UnicodeWarning
 - BytesWarning

-
-
- ResourceWarning

In [16]:

```
import inspect

def delilnik():
    try:
        x = int(input("Vnesi prvo številko: "))
        y = int(input("Vnesi drugo številko: "))
        rezultat = x / y
        print(f"{x}/{y} = {rezultat}")
    except Exception:
        print("Zmeraj ta prestreže.")
    except ValueError:
        print("Obe spremeljivki morata biti številki.")
    except ZeroDivisionError:
        print("Deljitelj ne sme biti 0.")

for _ in range(3):
    delilnik()
    print()

print(inspect.getmro(Exception))
print(inspect.getmro(ValueError))
print(inspect.getmro(ZeroDivisionError))
```

Vnesi prvo številko: 1
 Vnesi drugo številko: 0
 Zmeraj ta prestreže.

Vnesi prvo številko: 1
 Vnesi drugo številko: a
 Zmeraj ta prestreže.

Vnesi prvo številko: a
 Zmeraj ta prestreže.

(<class 'Exception'>, <class 'BaseException'>, <class 'object'>)
 (<class 'ValueError'>, <class 'Exception'>, <class 'BaseException'>, <class 'object'>)
 (<class 'ZeroDivisionError'>, <class 'ArithmeticError'>, <class 'Exception'>, <class 'BaseException'>, <class 'object'>)

In []:

Raising exceptions

Napake lahko raišamo tudi sami.

In [18]:

```
def delilnik_pozitivnih_st():
    try:
        x = int(input("Vnesi prvo številko: "))
        if x < 0:
            raise ValueError("Vnešena mora biti pozitivna številka")

        y = int(input("Vnesi drugo številko: "))
        if y < 0:
            raise ValueError("vnešena mora biti pozitivna številka")

        rezultat = x / y
        print(f"{x}/{y} = {rezultat}")
    except ValueError as e:
        print(e)
    except ZeroDivisionError:
        print("Deljitelj ne sme biti 0.")

for _ in range(3):
    delilnik_pozitivnih_st()
    print()
```

Vnesi prvo številko: -1
 Vnešena mora biti pozitivna številka

Vnesi prvo številko: 1
 Vnesi drugo številko: 2
 $1/2 = 0.5$

Vnesi prvo številko: 1
 Vnesi drugo številko: 2
 $1/2 = 0.5$

V tem primeru lahko uporabnik vnese negativno številko in ne bomo dobili errora pri pretvorbi:

```
int(input("Vnesi število: "))
```

Zato smo sami dodali preverjanje ali je številka pozitivna ali ne. V primeru, ko številka ni pozitivna smo sami vzdignili **ValueError** z našim specifičnim sporočilom.

In []:

Naloga:

Napišite funkcijo **is_palindrom**, ki od uporabnika zahteva naj vnese besedo. Funkcija naj vrne True, če je beseda palindrom, v nasprotnem primeru False. Palindrom je beseda, ki se prebere isto od leve proti desni in od desne proti levi.

Če uporabnik vnese samo številke naj funkcija rabi ValueError.

Program naj 3x zažene funkcijo. V kolikor pride do ValueError naj se izpiše sporočilo in izvajanje programa nadaljuje.

OUTPUT:

```
Vnesi besedo: Ananas
The word is NOT palindrom.
```

```
Vnesi besedo: 1234
Vnešene so bile samo številke.
```

```
Vnesi besedo: racecar
The word is PALINDROM
```

```
In [ ]: def is_palindrom():
    beseda = input("Vnesi besedo: ")
    if beseda.isnumeric():
        raise ValueError("Vnešene so bile samo številke.")

    st_crk = len(beseda)
    for i in range(st_crk):
        #print("Checking", beseda[i], "and", beseda[-1*i-1])
        if not(beseda[i] == beseda[-1*i -1]):
            return False
    return True

for _ in range(3):
    try:
        if is_palindrom():
            print("The word is PALINDROM")
        else:
            print("The word is NOT palindrom.")
    except ValueError as e:
        print(e)
    print()
```

In []:

The else and finally statements

Skupaj z try-except lahko uporabimo tudi else in finally .

else se bo izvršil, če try ne vrže napake.

```
In [19]: try:
    x = int(input("Vnesi številko: "))
except ValueError:
    print("To ni številka.")
else:
    print("Else statement.")

print("End")
```

```
Vnesi številko: 1
Else statement.
End
```

`finally` se izvede po koncu `try-except` ne glede ali se je napaka ni zgodila, ali se je napaka zgodila in je bila pohendljana, ali se je napaka zgodila in ni bila pohendljana.

Ponavadi se uporabi za čiščenje kode.

```
In [20]: try:
    x = int(input("Vnesi številko: "))
    print(5/x) # da simuliramo deljenje z 0, ki bo naš nepohendlan error
except ValueError:
    print("To ni številka.")
finally:
    print("Finally statement.")

print("End")
```

Vnesi številko:
To ni številka.
Finally statement.
End

Writting our own Exceptions

Napišemo lahko tudi naše Exceptions.

Svoje exceptione lahko ustvarimo tako, da ustvarimo nov razred, ki deduje iz nekega Exception razreda. Ponavadi je to kar direktno iz osnovnega `Exception` razreda.

```
In [21]: class MojError(Exception):
    pass

try:
    raise MojError("We raised MojError")
except MojError as e:
    print(e)
```

We raised MojError

Ko pišemo bolj obsežen python program, je dobra praksa, da vse naše errorje zapišemo v posebno datoteko. Ponavadi je datoteka poimenovana `errors.py` ali `exceptions.py`.

In []:

Če si pogledamo na bolj konkretnem primeru:

Ustvarili bomo program, kjer uporabnik ugiba neko določeno celo številko. Ustvarili bomo dva naša error classa. Enega v primeru, če je ugibana številka prevelika, drugega v primeru, da je ugibana številka premajhna.

```
In [23]: class VrednostPremajhna(Exception):
    pass

class VrednostPrevisoka(Exception):
    pass

number = 10 # številka katero ugibamo

while True:
    try:
        i_num = int(input("Enter a number: "))
        if i_num < number:
            raise VrednostPremajhna
        elif i_num > number:
            raise VrednostPrevisoka
        break
    except VrednostPremajhna:
        print("Ugibana vrednost je premajhna!")
        print()
    except VrednostPrevisoka:
        print("Ugibana vrednost je previsoka!")
        print()

print("PRAVILNO.")
```

```
Enter a number: 20
Ugibana vrednost je previsoka!

Enter a number: 1
Ugibana vrednost je premajhna!

Enter a number: 5
Ugibana vrednost je premajhna!

Enter a number: 11
Ugibana vrednost je previsoka!

Enter a number: 9
Ugibana vrednost je premajhna!

Enter a number: 10
PRAVILNO.
```

In []:

Naloga:

Napišite funkcijo, ki kot parameter x premje neko celo število. Funkcija naj izpiše zadnjih x vrstic v datoteki *naloga2.txt*.

INPUT:
funkcija(3)

OUTPUT:
line 7
line 8
line 9

```
def funkcija(n): with open("naloga2.txt", "r") as f: data = f.readlines() for line in data[-n:]: print(line, end="")
```

funkcija(3)

Naloga:

Napišite funkcijo, ki v datoteko *naloga5.txt* zapiše vse datume, ki so **petek 13.** v letih od 2020 do 2030.

Da najdete datume si lahko pomagate s knjižnjico datetime.

```
INPUT:
print(fakulteta())
```

OUTPUT:
Vnesi cifro: a
To ni bila številka.
Vnesi cifro: b
To ni bila številka.
Vnesi cifro: 3
6

```
In [ ]: try:
    x = int(input("Vnesi številko: "))
    rezultat = 10 / x
except ValueError:
    print("To ni številka")
else:
    print("Else statement")
finally:
    print("Finally statement")
print("Nadaljevanje programa")
```

```
In [ ]:
```

Importing

Importing je način, kako lahko kodo iz ene datoteke/modula/package uporabimo v drugi datoteki/modulu.

- **module** je datoteka, ki ima končnico .py
- **package** je direktorij, ki vsebuje vsaj en modul

Da importiramo modul uporabimo besedo `import`.

```
import moj_modul
```

Python sedaj prvo preveri ali se *moj_modul* nahaja v **sys.modules** – to je dictionary, ki hrani imena vseh importiranih modulov.

Če ne najde imena, bo nadaljeval iskanje v built-in modulih. To so moduli, ki pridejo skupaj z inštalacijo Pythona. Najdemo jih lahko v Python Standardni Knjižnjici – <https://docs.python.org/3/library/> (<https://docs.python.org/3/library/>) .

Če ponovno ne najde našega modula, Python nadaljuje iskanje v **sys.path** – to je list direktorijev med katerimi je tudi naša mapa.

Če Python ne najde imena vrže **ModuleNotFoundError**. V primeru, da najde ime, lahko modul sedaj uporabljam v naši datoteki.

Za začetek bomo importiral **math** built-in modul, ki nam omogoča naprednejše matematične operacije, kot je uporaba korenjenja.

`math` documentation – <https://docs.python.org/3/library/math.html> (<https://docs.python.org/3/library/math.html>).

Da pogledamo katere spremenljivke / funkcije / objekti / itd. so dostopni v naši kodi lahko uporabimo **dir()** funkcijo.

`dir` documentation – <https://docs.python.org/3/library/functions.html#dir> (<https://docs.python.org/3/library/functions.html#dir>)

```
In [ ]: import math

moja_spremenljivka = 5
print(dir())

print(moja_spremenljivka)
print(math)
```

S pomočjo **dir(...)** lahko tudi preverimo katere spremenljivke, funkcije, itd. se nahajajo v importiranih modulih.

```
In [ ]: import math
moja_spremenljivka = 5
print(dir(math))
```

Funkcijo, spremenljivko, atribut v math modulu uporabimo na sledeč način:

```
In [ ]: import math
print(math.sqrt(36))
```

```
In [ ]:
```

Naloga:

S pomočjo `math` modula izračunajte logaritem 144 z osnovo 12.

<https://docs.python.org/3/library/math.html> (<https://docs.python.org/3/library/math.html>)

```
In [6]: # Rešitev
import math
math.log(144, 12)
```

```
Out[6]: 2.0
```

Importing our own module

Ustvarimo novo datoteko `moj_modul.py` zraven naše datoteke s kodo.

```
└── _python_tecaj/
    ├── moj_modul.py
    └── skripta.py
```

moj_modul.py

```
In [ ]: class Pes():
    def __init__(self, ime):
        self.ime = ime

    def sestevalnik(a, b):
        return a+b

moja_spremenljivka = 100
```

skripta.py

```
In [ ]: import moj_modul

print(dir())
print(dir(moj_modul))

fido = moj_modul.Pes("fido")
print(fido.ime)

print(moj_modul.sestevalnik(5, 6))

print(moj_modul.moja_spremenljivka)
```

Načini importiranja

Importiramo lahko celotno kodo ali pa samo specifične funkcije, spremenljivke, objekte, itd.

Celotno kodo importiramo na sledeči način:

```
import moj_modul
```

```
In [ ]: import moj_modul
print(dir())
fido = moj_modul.Pes("fido")
print(fido.ime)
print(moj_modul.sestevalnik(5, 6))
print(moj_modul.moja_spremenljivka)
```

Specifične zadeve importiramo na sledeč način:

```
from moj_modul import moja_spremenljivka
```

```
In [ ]: from moj_modul import moja_spremenljivka
print(dir())
print(moja_spremenljivka)
```

```
In [ ]: from moj_modul import sestevalnik
print(dir())
print(sestevalnik(5,6))
```

```
In [ ]: from moj_modul import Pes
print(dir())
fido = Pes("fido")
print(fido.ime)
```

Importirane zadeve se lahko shrani tudi pod drugim imenom

```
import moj_modul as mm
```

```
In [ ]: import moj_modul as mm
print(dir())
fido = mm.Pes("fido")
print(fido.ime)
print(mm.sestevalnik(5, 6))
print(mm.moja_spremenljivka)
```

```
In [ ]: from moj_modul import sestevalnik as sum_
print(dir())
print(sum_(5,6))
```

Za premikanje med direktoriji med importiranjem se uporablja " ..".

```
from package1.module1 import function1
```

```
├── _python_tecaj/
│   ├── moj_modul.py
│   └── skripta.py
└── _moj_package/
    └── modul2.py
```

modul2.py

```
In [ ]: def potenciranje(x, y):
    return x**y

spremenljivka2 = 200
```

skripty.py

```
In [ ]: from moj_package import modul2

print(dir())
print(modul2.potenciranje(2,3))
```

```
In [ ]: from moj_package.modul2 import potenciranje

print(dir())
print(potenciranje(2,3))
```

Naloga:

Ustvarite nov modul imenovan **naloga1.py**. Znotraj modula napišite funkcijo **pretvornik(x, mode)**, ki spreminja radiane v stopinje in obratno.

Funkcija naj sprejme 2 argumenta. Prvi argument je vrednost, katero želimo pretvoriti. Drugi argument, imenovan **mode** pa nam pove v katero enoto spreminjamamo.

`mode = "deg2rad"` pomeni, da spreminjamamo iz stopinj v radiane
`mode = "rad2deg"` pomeni, da spreminjamamo iz radianov v stopinje

Za pomoč pri pretvarjanju uporabite **math** modul.

Zravn modula prilepite podano skripto **test.py** in to skripto zaženite.

```
In [ ]: # test.py
import naloga1

r1 = naloga1.pretvornik(180, mode="deg2rad")
if float(str(r1)[:4]) == 3.14:
    print("Rešitev pravilna.")
else:
    print("Nekaj je narobe.")

r2 = naloga1.pretvornik(360, mode="deg2rad")
if float(str(r2)[:4]) == 6.28:
    print("Rešitev pravilna.")
else:
    print("Nekaj je narobe.")

r3 = naloga1.pretvornik(1.5707963267948966, mode="rad2deg")
if r3 == 90:
    print("Rešitev pravilna.")
else:
    print("Nekaj je narobe.")

r3 = naloga1.pretvornik(4.71238898038469, mode="rad2deg")
if r3 == 270:
    print("Rešitev pravilna.")
else:
    print("Nekaj je narobe.")
```

```
In [ ]: # Rešitev
import math

def pretvornik(x ,mode="deg2rad"):
    if mode == "deg2rad":
        return math.radians(x)
    elif mode == "rad2deg":
        return math.degrees(x)
```

Importiramo lahko tudi vse naenkrat z uporabo "*" vendar se to odsvetuje, saj nevem kaj vse smo importirali in lahko na tak način ponesreči kaj spremenimo.

```
In [ ]: from math import *

print(dir())
print(pi)

pi = 3
print(pi)
```

```
In [ ]:
```

__name__ variable

Python ima posebno spremenljivko `__name__`. Spremenljivka dobi vrednost, glede na to kako smo zagnali naš modul.

Če zaženemo naš modul direktno, bo spremenljivka enaka `__main__`.

m1.py

```
In [ ]: def my_name():
    print(__name__)

my_name()
```

To bi delovalo v primeru, ko smo ustvarili svoj modul in vanj sproti zapisali kakšen preprost test naše kode.

Problem se pojavi, ko `moj_modul` importiramo, sam se ob importiranju celotna koda v modulu izvede.

```
In [ ]: import m1

print(__name__)
print(m1.__name__)
```

Da preprečimo nepotrebeno izvajanje funkcij lahko uporabimo `__name__` spremenljivko.

Naš modul bi sedaj izgledal sledeče:

m1.py

```
In [ ]: def my_name():
    print(__name__)

if __name__ == "__main__":
    my_name()
```

skripta.py

```
In [ ]: import m1

print(__name__)
print(m1.__name__)
```

```
In [ ]:
```

Delo z datotekami

What is a file?

Datoteke uporabljamo, da v njih trajno shranimo podatke.

V splošnem delo z datotekami poteka na sledeč način:

- Odpremo datoteko
- Izvedemo operacijo (pisanje podatkov v datoteko, branje podatkov, itd..)
- Zapremo datoteko (ter tako sprostimo vire, ki so vezani na upravljanje z datoteko -> spomin, procesorska moč, itd..)

Odpiranje datotek

Python ima že vgrajeno funkcijo `open()` za odpiranje datotek.

Funkcija nam vrne `file object`, imenovan tudi `handle`, s katerim lahko izvajamo operacije nad datoteko.

```
In [ ]: f = open("test.txt")      # open file in current directory
#f = open("C:/Python33/README.txt") # specifying full path
```

Dodatno lahko specificiramo v kakšnem načinu želimo odpreti datoteko.

Lahko jo odpremo v **text mode**. Ko beremo podatke v tem načinu, dobivamo *strings*. To je *default mode*. Lahko pa datoteko odpremo v **binary mode**, kjer podatke beremo kot *bytes*. Takšen način se uporablja pri branju non-text datotek, kot so slike, itd..

Datoteke lahko odpremo v načinu:

- **r** - Podatke lahko samo beremo. (default način)
- **w** - Podatke lahko pišemo v datoteko. Če datoteka ne obstaja jo ustvarimo. Če datoteka obstaja jo prepišemo (če so bli noter podatki jih izgubimo)
- **x** - Ustvarimo datoteko. Če datoteka že obstaja operacija fail-a
- **a** - Odpremo datoteko z namenom dodajanja novih podatkov. Če datoteka ne obstaja jo ustvarimo.
- **t** - odpremo v "text mode" (dafult mode)
- **b** - odpremo v "binary mode"

```
In [ ]: f = open("test.txt")      # equivalent to 'r' or 'rt'
```

```
In [ ]: f = open("test.txt", 'r')  # write in text mode
print(type(f))
print(f)
```

```
In [ ]: f = open("test.txt", 'w')  # write in text mode
```

Unlike other languages, the character 'a' does not imply the number 97 until it is encoded using ASCII (or other equivalent encodings).

Moreover, the default encoding is platform dependent. In windows, it is 'cp1252' but 'utf-8' in Linux.

So, we must not also rely on the default encoding or else our code will behave differently in different platforms.

Hence, when working with files in text mode, it is highly recommended to specify the encoding type.

```
In [ ]: f = open("test.txt",mode = 'r',encoding = 'utf-8')
```

Zapiranje datotek

Ko končamo z našo operacijo moramo datoteko zapreti, ker tako sprostimo vire, ki so vezani na uporabo datoteke (spomin, procesorska moč, itd..).

```
In [ ]: f = open("test.txt", "a")
# perform file operations

f.close()
```

na Linuxu ne dela ta `f.close()`. Še kr lah spreminjaam

Tak način upravljanja z datotekami ni najbolj varen. Če smo odprli datoteko in potem med izvajanjem operacije nad datoteko pride do napake, datoteke ne bomo zaprli.

Varnejši način bi bil z uporabo **try-finally**.

```
In [ ]: try:
    f = open("test.txt", "a")
    # perform file operations
    raise ValueError
finally:
    f.close()

# če ta koda deluje, potem bi morali biti zmožni spremenjati datoteko tudi potem, ko dobimo error.
```

[Python with Context Managers \(<https://jeffknupp.com/blog/2016/03/07/python-with-context-managers/>\)](https://jeffknupp.com/blog/2016/03/07/python-with-context-managers/)

Isto stvar dosežemo z uporabo `with` statement .

```
In [25]: with open("test.txt", "a") as f:
    pass
    # perform file operations
```

```
In [ ]:
```

dodat zadevo glede fopen flag da vidjo kdaj je zaprt

```
In [ ]:
```

Branje datotek

Za branje, datoteko odpremo v `read (r)` načinu.

(Imamo datoteko katere vsebina je: `Hello World!\nThis is my file.`)

```
In [28]: with open("test.txt", 'r') as f:
    file_data = f.read()      # read all data
    print(file_data)

#print(file_data)
#file_data
```

Hello world
This is my file

```
In [29]: with open("test.txt", "r") as f:
    file_data = f.read(2) # read the first 2 data
    print(file_data)
    file_data = f.read(6) # read the next 6 data
    print(file_data)
    file_data = f.read() # reads till the end of the file
    print(file_data)
    file_data = f.read() # further reading returns empty string
    print(file_data)
```

He
llo wo
rld
This is my file

We can see that, the `read()` method returns newline as "`\n`". Once the end of file is reached, we get empty string on further reading.

Po datoteki se lahko tudi premikamo z uporabo `seek()` in `tell()` metode.

```
In [30]: with open("test.txt", "r") as f:
    print(f.tell()) # get current position of file cursor in bytes
    f.read(4) # read 4 bytes
    print(f.tell())
```

0
4

```
In [31]: with open("test.txt", "r") as f:
    print(f.tell()) # get position in bytes

    reading = f.read(6) # read 6 bytes
    print(reading)
    print(f.tell()) # get new position in bytes

    f.seek(0) # move cursor to position 0
    print(f.tell())

    reading = f.read(6)
    print(reading)
```

```
0
Hello
6
0
Hello
```

Datoteko lahko hitro in učinkovito preberemo vrstico po vrstico, z uporabo `for` loop.

```
In [32]: with open("test.txt", "r") as f:
    for line in f:
        print(line) # The Lines in file itself has a newline character '\n'.
```

```
Hello world

This is my file
```

Alternativno lahko uporabljamo `readline()` metodo za branje individualnih vrstic.

Metoda prebere podatke iz datoteke do newline (`\n`) .

```
In [33]: with open("test.txt", "r") as f:
    print(f.readline())
    print(f.readline())
    print(f.readline())
```

```
Hello world

This is my file
```

`readlines()` nam vrne listo preostalih linij v datoteki.

(če providm `readlines()` prebere vrstice in postavi cursor na konc)

```
In [34]: with open("test.txt", "r") as f:
    list_of_lines = f.readlines()
    print(list_of_lines)
    print(list_of_lines[1])
```

```
['Hello world\n', 'This is my file\n']
This is my file
```

Naloga:

Napišite funkcijo, ki kot parameter `x` premje neko celo število. Funkcija naj izpiše zadnjih `x` vrstic v datoteki `naloga2.txt`.

INPUT:
`funkcija(3)`

OUTPUT:
`line 7
line 8
line 9`

```
In [37]: def funkcija(n):
    with open("nalog2.txt", "r") as f:
        data = f.readlines()
        for line in data[-n:]:
            print(line, end="")

funkcija(3)
```

line 7
line 8
line 9

In []:

Naloga:

Napišite funkcijo `dictionary`, ki vpraša uporabnika naj vnese določen string in nato vrne vse besede, ki vsebujejo podani string.

Vse možne besede najdete v datoteki `words_alpha.txt`

INPUT:
`dictionary()`

OUTPUT:
Vnesi besedo: meow
homeown
homeowner
homeowners
meow
meowed
meowing
meows

```
In [9]: def dictionary():
    beseda = input("Vnesi besedo: ")

    with open("words_alpha.txt", "r") as f:
        for line in f.readlines():
            if beseda in line:
                print(line, end="")

dictionary()
```

Vnesi besedo: meow
homeown
homeowner
homeowners
meow
meowed
meowing
meows

In []:

Pisanje datotek

Za pisanje v datoteko jo odpremo v načinu za pisanje:

- **w** (ta način bo prepisal vse podatke že shranjene v datoteki)
- **a** (s tem načinom bomo dodajali podatke na konec datoteke)
- **x** (s tem ustvarimo datoteko in lahko začnemo v njo pisati)

Writing a string or sequence of bytes (for binary files) is done using `write()` method. This method returns the number of characters written to the file.

```
In [ ]: with open("test.txt", 'w') as f:
    f.write("my first file\n")
    f.write("This file\n\n")
    f.write("contains three lines\n")

# This program will create a new file named 'test.txt' if it does not exist. If it does exist, it is overwritten.
# We must include the newline characters ourselves to distinguish different lines.
```

```
In [ ]: with open("test.txt", 'a') as f:
    f.write("We are adding another line.")
    x = f.write("And another one")
    #print(x) write() returns number of bytes we wrote

# this program will open a file and append what we want to the end
```

```
In [ ]: with open("test2.txt", "x") as f:
    f.write("New .txt")

# this program will create a new file 'test2.txt' if it doesn't exist and write into it.
# if the file exists it will throw an error
```

```
In [ ]:
```

Naloga:

Napišite funkcijo, ki v datoteko *naloga5.txt* zapiše vse datume, ki so **petek 13.** v letih od 2020 do 2030.

Da najdete datume si lahko pomagate s knjižnjico **datetime**.

INPUT:
funkcija()

OUTPUT:
13. Mar 2020
13. Nov 2020
13. Aug 2021
13. May 2022
13. Jan 2023
13. Oct 2023
13. Sep 2024
13. Dec 2024
13. Jun 2025
13. Feb 2026
13. Mar 2026
13. Nov 2026
13. Aug 2027
13. Oct 2028
13. Apr 2029
13. Jul 2029

```
In [38]: from datetime import date

def funkcija():
    with open("naloga5.txt", "w") as f:
        for year in range(2020, 2031):
            for month in range(1, 13):
                datum = date(year, month, 13)
                #print(datum)
                if datum.weekday() == 4: # 0=Mon, 1=Tue, ..., 4=Fri
                    f.write(f'{datum.strftime("%d. %b %Y")}\n')

funkcija()
```

```
In [ ]:
```

JSON

<https://realpython.com/python-json/> (<https://realpython.com/python-json/>)

JSON - JavaScript Object Notation, je način zapisa informacij v organizirano in preprosto strukturo, ki je lahko berljiva tako za ljudi kot tudi za računalnike.

```
{
    "firstName": "Jane",
    "lastName": "Doe",
    "hobbies": ["running", "sky diving", "singing"],
    "age": 35,
    "children": [
        {
            "firstName": "Alice",
            "age": 6
        },
        {
            "firstName": "Bob",
            "age": 8
        }
    ]
}
```

Za manipuliranje z JSON podatki v Pythonu uporabljamo `import json` modul.

Python object translated into JSON objects

Python	JSON
dict	object
list, tuple	array
str	string
int, long, float	number
True	true
False	false
None	null

Primer shranjevanja JSON podatkov.

In [99]: `import json`

In [100]: `data = {`

```

        "firstName": "Jane",
        "lastName": "Doe",
        "hobbies": ["running", "sky diving", "singing"],
        "age": 35,
        "children": [
            {
                "firstName": "Alice",
                "age": 6
            },
            {
                "firstName": "Bob",
                "age": 8
            }
        ]
    }
```

In [112]: `with open("data_file.json", "w") as write_file:`
 `json.dump(data, write_file)`
`# Note that dump() takes two positional arguments:`
`#{(1) the data object to be serialized,`
`#and (2) the file-like object to which the bytes will be written.`

Branje JSON podatkov

JSON	Python
object	dict
array	list

JSON	Python
string	str
number (int)	int
number (real)	float
true	True
false	False
null	None

Technically, this conversion isn't a perfect inverse to the serialization table. That basically means that if you encode an object now and then decode it again later, you may not get exactly the same object back.

In reality, the simplest example would be encoding a tuple and getting back a list after decoding.

```
In [114]: with open("data_file.json", "r") as read_file:
    data = json.load(read_file) # use Loads() if the JSON data is in "python string" type
    print(data)
    print(type(data))

{'firstName': 'Jane', 'lastName': 'Doe', 'hobbies': ['running', 'sky diving', 'singing'], 'age': 35, 'children': [{'firstName': 'Alice', 'age': 6}, {'firstName': 'Bob', 'age': 8}]}
<class 'dict'>
```

In []:

In []:

Naloga:

Napišite program, ki prebere **podatki.json**. Program naj primerja zaslužke vseh oseb med seboj (salary + bonus) in nato izpiše ime in celotni zaslužek te osebe.

OUTPUT:

The top earner is martha, making 10300€

```
In [5]: import json

with open("podatki.json") as f:
    data = json.load(f) # use Loads() if the JSON data is in "python string" type
    #print(data)

max_pay = 0
name = ""
for employee in data["company"]["employees"]:
    print(employee)
    if employee["payble"]["salary"] + employee["payble"]["bonus"] > max_pay:
        name = employee["name"]
        max_pay = employee["payble"]["salary"] + employee["payble"]["bonus"]

print(f"The top earner is {name}, making {max_pay}€")

{'name': 'emma', 'payble': {'salary': 7000, 'bonus': 800}}
{'name': 'derek', 'payble': {'salary': 4000, 'bonus': 1000}}
{'name': 'alex', 'payble': {'salary': 7500, 'bonus': 500}}
{'name': 'susan', 'payble': {'salary': 6300, 'bonus': 350}}
{'name': 'martha', 'payble': {'salary': 9100, 'bonus': 1200}}
{'name': 'clark', 'payble': {'salary': 7700, 'bonus': 270}}
{'name': 'luise', 'payble': {'salary': 8200, 'bonus': 900}}
The top earner is martha, making 10300€
```

In []:

In []: