OS-Level Python 3 Scripting
Useful Modules, Tips and Tricks

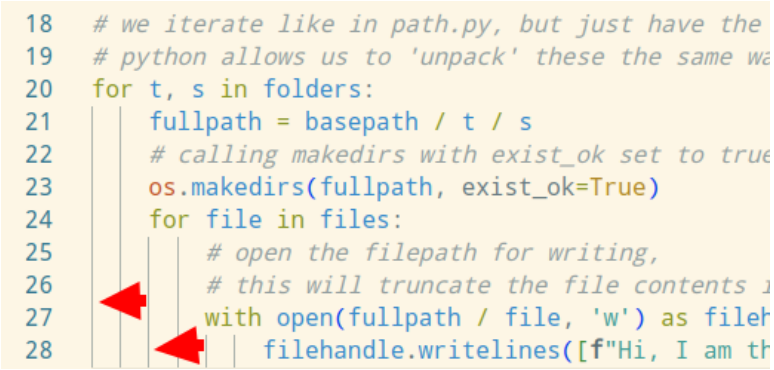
Markus Krause

Python 3 - basics of basics

- Python uses indentation for functional blocks, there is no semicolons needed

- gold standard is 4 spaces - DO NOT USE TABS - DO NOT MIX WHITESPACE TYPES (!!!)

- editor with ‘falling’ indentation lines helps



- official documentation: https://docs.python.org/3/

- the ‘common oldest denominator’ if you wish to support almost all systems is Python 3.6, versions below that were early when Python 2 still dominated and were never widely adopted

“Shebang” shell line - how to call python correctly

- several options, none is truly 100% portable

- python3 should be requested directly, as some distros still have python 2.7 as default, so just using python is a slippery slope

- most likely the best variants:

```
#!/usr/bin/env python3
```

```
#!/bin/env python3
```

- also often seen:

```
#!/bin/python3
```

```
#!/usr/bin/python3
```

- do not code new stuff in python 2, it’s dead, deprecated and unsupported (and less nice)

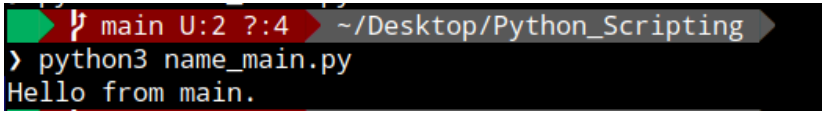
- alternative: omit the shebang completely and call script via ‘python yourscript.py’ - not always the best option but guarantees you use the interpreter of your choice

a word on if __name__ == “__main__”

- especially in python that mimics a script to be executed directly, you will find the infamous `if __name__ == “__main__”:` construct
- this is to mark the execution entry point if the script is called as the top level by the interpreter
- purpose is to avoid unintended code execution when importing from your python file
- usually not a concern for single-file helper scripts
- if you are doing something more complex that consists of multiple files and imports: use it!

```python
#!/usr/bin/env python3

def main():
    print("Hello from main.")

# only execute our code if we are the script that has been called as the main program
# this mainly makes sense if you re-use the code of some functions or even main
# in another python module, otherwise on import the code immidiately executes, which usually
# is not what you want
# for single-file scripts, it is often recommended, but not strictly necessary
if __name__ == "__main__":
    main()
```

name_main.py



basic arithmetics and types

- integers are auto-sized in Python by the runtime, do not expect automatic overflows, they are signed
- + (add), - (sub), * (mult), % (modulo), ** (power of) function as expected
- Division is special:
  - / is floating point division and will return a float even if the result fits in an integer
  - / also is ‘overloaded’ in some contexts for other functions (see pathlib sections)
  - // is flooring division, it always rounds DOWN, resulting type is int, or float min. one operand was float
- + (add) has also many other uses when not used on integers, e.g.:
  - you can concatenate lists
  - you can concatenate strings
- for very powerful matrix, vector, analytical math and data science, there is numpy, scipy and pandas, but these are normally not used in OS-level scripting and come with huge dependencies

basic arithmetics and types

```
1   #!/usr/bin/env python3
2
3   # results are integers if all operands are integers
4   print("1+2 = ", 1+2)
5   print("1-2 = ", 1-2)
6   print("3*2 = ", 3*2)
7   print("3%2 = ", 3%2)
8   print("3**2 = ", 3**2)
9   print("3/2 = ", 3/2)  # only exception, this always returns a float
10  print("3//2 = ", 3//2)
11
12  # some overloaded + magic
13  print("Concatenated lists: ", [1,2,3] + [4,5,6])
14  print("Concatenated strings: ", "Hello " + "there.")
```

arith.py and its output

```
~/Desktop/Python_Scripting
❯ python3 arith.py
1+2 =  3
1-2 =  -1
3*2 =  6
3%2 =  1
3**2 =  9
3/2 =  1.5
3//2 =  1
Concatenated lists:  [1, 2, 3, 4, 5, 6]
Concatenated strings:  Hello there.
```

strings

- python strings can be delimited using “ ” or ‘ ‘, which is useful for nesting if you need these within the string

- f” “ marks a format string, where within { } variables an expressions can directly be inserted, this makes for very readable templating

- the older .format or C-style %-formatting syntax is also still available

- consult the official docs for formatting options, there is a lot available (pre-/after comma digits, hex, bin, filling with zeroes and much more) https://docs.python.org/3/library/string.html#format-examples

- there are loads of useful helper functions like split, join, endswith, startswith, find etc.

- if you want to check if a substring is within a given string, you can just use the ‘in’ operator

- for more complex pattern matching there are the fnmatch and regex modules, these go beyond the scope here, check their docs and the excellent howto:

  - https://docs.python.org/3/library/fnmatch.html?highlight=fnmatch#module-fnmatch

  - https://docs.python.org/3/howto/regex.html

strings

```python
#!/usr/bin/env python3

string = "This is a wonderful teststring."

multistring = """
This is a multiline string.

It is very conventient to use.
Use it either for documentation or for e.g. templates
you wish to fill later etc. .
"""

# you can split strings into a list like this,
# split() also takes other strings as seperator, whitespace is default
words = string.split()  # this splits on any whitespace (space, tab etc.)
print("Whitespace split: ", words)
lines = multistring.splitlines()  # this splits on newlines (cross-OS safe)
print("Newline split: ", lines)

# now a few more neat string examples
print("_".join(words))  # take the string we split at whitespace and re-join it, but using _

# custom 'grep' filtering with 3 conditions
print("\nNon-empty lines not ending with a '.' and lines that contain 'later':")
for line in lines:
    if (line and not line.endswith('.')) or 'later' in line:
        print(line)

# some number formatting
i = 1289468
f = -34.8482
print(f"i: {i}\ni(hex, padded 0 to 14 digits): {i:014x}")
print(f"f: {f}\nf(2 comma digits): {f:.2f}")
```

strings.py

strings

Example output of strings.py:

```
 main U:2 ?:4  ~/Desktop/Python_Scripting
❯ python3 strings.py
Whitespace split:  ['This', 'is', 'a', 'wonderful', 'teststring.']
Newline split:  ['', 'This is a multiline string.', '', 'It is very conventient to use.', 'Use it either for documentation or for e.g. templates', 'you wish to fill later etc. .']
This_is_a_wonderful_teststring.

Non-empty lines not ending with a '.' and lines that contain 'later':
Use it either for documentation or for e.g. templates
you wish to fill later etc. .
i: 1289468
i(hex, padded 0 to 14 digits): 0000000013acfc
f: -34.8482
f(2 comma digits): -34.85
```

collections

- pythons collections are one of the most powerful features of the language

- beware: the normal dictionary (aka HashMap in other languages) guarantees retention of insertion order in CPython in 3.6 and it is an actual language feature in 3.7, but if you rely on it, using OrderedDict explicitly might be a good idea

- next to lists, which can grow and shrink, there is also the fixed tuple, which mostly behaves like a list, but cannot be modified after creation (uses () instead of [] around it)

- another useful one is the ‘set’, which is unordered, unchangeable, and does not allow duplicate values

- there are also more fancvy containers like ‘deque’ (double-ended queue)

- dict, list, tuple and set are standard features and available without any import, others can be retrieved using the standard module import ‘collections’

- check the official docs for all the awesome features: https://docs.python.org/3/library/collections.html

- beware variables containing a collection are POINTERS, so an assignment will not copy the underlying collection unless explicitly requested using .copy()

collections - dict and lists

```python
#!/usr/bin/env python3
from pprint import pprint

# empty dictionary (key-value store, some languages call it a map or hash map)
d = {}
# empty list
l1 = []

# dict keys can basically be anything that is hashable, so even e.g. tuples are allowed
# but it cannot be anything dynamic like a list, another dict or so, these only work as values
# insertion is just done by inventing another key in the []-access syntax
d["a"] = 123
d[(1, "c")] = "value"

# dict keys and values can easily be retrieved using these functions
# they are iterables, but can be flattened into a list like so
print("Printing dict d keys and values as lists: ")
vals = list(d.values())
print("Values: ", vals)
keys = list(d.keys())
print("Keys: ", keys)

# you can also get an iterator over the items, which unpacks into k(eys) and v(alues)
print("Printing dict d from a for loop: ")
for k, v in d.items():
    print(k, ": ", v)
# especially for more complex types or objects like dicts,
# pprint (aka pretty print) provides a more human readable output
print("Pretty-Printing dict d: ")
pprint(d)

# list and other collection variables are pointers, so l2 = l1 copies the reference, but
# points to the same list, so if you change l1 or l2, the other one changes too
l1.append(1)
l2 = l1
l2.append(2)
# both l1 and l2 have changed because they are just differently named pointers to the same list
print("\nList1: ", l1, " List2: ", l2)
# if you really wish to copy the list to make it independent later, do it like so:
l2 = l1.copy()
l2.append(3)
print("List1: ", l1, " List2 (copied before append): ", l2)
```

collections_dictlist.py and its output:

```
 main U:2 ?:4  ~/Desktop/Python_Scripting
❯ python3 collections_dictlist.py
Printing dict d keys and values as lists:
Values:  [123, 'value']
Keys:  ['a', (1, 'c')]
Printing dict d from a for loop:
a :  123
(1, 'c') :  value
Pretty-Printing dict d:
{'a': 123, (1, 'c'): 'value'}

List1:  [1, 2]  List2:  [1, 2]
List1:  [1, 2]  List2 (copied before append):  [1, 2, 3]
```

collections - iterators

```python
#!/usr/bin/env python3
from pprint import pprint

# enumerate provides a very conventient way to iterate over a collection
# and get a counter variable for free, the start is 0 by default, here we choose 1
letters = ['a', 'b', 'c', 'd']
for i, letter in enumerate(letters, start=1):
    print(f"{i} {letter}")
# len lets you know the size of a collection like lists
print(f"There are {len(letters)} in the letters list.")

# range provides an iterator to easily generate series of numbers
# we use Pythons list comprehension to quickly generate lists:
# even numbers up to (not including), 10
evens_mod = [i for i in range(10) if i % 2 == 0]
print("Evens up to 10 using mod (%): ", evens_mod)
# also even numbers up to 10, but using the step parameter of range
evens_step = [i for i in range(0,10,2)]
print("Evens up to 10 using range step: ", evens_step)
# iterators can often also be cast directly to lists, same result as above
evens_list_cast = list(range(0,10,2))
print("Evens up to 10 using list casting: ", evens_list_cast)
# squared versions of numbers between 10 (included) and 20 (not included)
squares = [f"{i}**2 = {i**2}" for i in range(10,20)]
print("Squares from 10 up to 20: ")
for s in squares:
    print(s)
```

collections_iterators.py and its output:

```
 main U:2 ?:4  ~/Desktop/Python_Scripting
❯ python3 collections_iterators.py
1 a
2 b
3 c
4 d
There are 4 in the letters list.
Evens up to 10 using mod (%):  [0, 2, 4, 6, 8]
Evens up to 10 using range step:  [0, 2, 4, 6, 8]
Evens up to 10 using list casting:  [0, 2, 4, 6, 8]
Squares from 10 up to 20:
10**2 = 100
11**2 = 121
12**2 = 144
13**2 = 169
14**2 = 196
15**2 = 225
16**2 = 256
17**2 = 289
18**2 = 324
19**2 = 361
```

sys/os-module - exit codes and environment

```python
#!/usr/bin/env python3
import os
import sys

env = os.environ  # retrieve the environment the script runs in

try:
    # the " and ' characters can be mixed like below to not collide with the string
    # the f before the string makes it a 'format-string' where variables can be
    # inserted directly in curly braces, which makes it very readable
    print(f"Hello from Python, your name is: {env['NAME']}")
    sys.exit(0) # exit with 0, all went well
except KeyError:
    # if we end up here, env['NAME'] could not be looked up
    # but instead of crashing we catch the exception and give a nice error
    print(f"Hello from Python, your NAME env variable is not set :( .")
    sys.exit(1) # exit with 1, something went wrong

```

sys_os.py

pathlib-module - os path handling

- pathlib is a cross-platform way of handling paths, and preferred in most cases over the old os.path

- automatically handles path delimiters the right way (“/” or “\”)

- by now in Python 3, Path classes are accepted basically by almost all system functions that also take simple strings as path inputs

- provides an overloaded / operator to easily attach further directories in the form of strings

- provides loads of useful check-functions like exists(), is_absolute()

- with iterdir() it is easy to iterate over all files in a directory

- with chmod() you can set permissions on a path

- loads more functionality, see the spec: https://docs.python.org/3/library/pathlib.html#pathlib.Path

pathlib-module - os path handling

```python
#!/usr/bin/env python3

from pathlib import Path
import os

# compared to the sys_os.py script, the get method never 'crashes',
# but returns None if TARGET_DIR would be undefined
home = os.environ.get("TARGET_DIR")

# if the TARGET_DIR env variable is set we use it, otherwise we use /tmp as default
basepath = Path("/tmp" if not home else home)

topfolders = ["Pictures", "Downloads", "Documents"]
subfolders = [".thumbnails", ".cache", ".tmp"]

# a bit senseless in this example, but good for demonstration
# we 'zip' both lists together and with each iteration we get
# the pairing of the topfolder and the subfolder we want to create
for t, s in zip(topfolders, subfolders):
    # the path class 'overloads the division symbol so you can attach further
    # path components by simply 'dividing' the path by a string
    fullpath = basepath / t / s
    # calling makedirs with exist_ok set to true behaves like the well know 'mkdir -p'
    os.makedirs(fullpath, exist_ok=True)
```

path.py script and its output:

```
~/Desktop/Python_Scripting
❯ export TARGET_DIR="/tmp/demo/"
~/Desktop/Python_Scripting
❯ python3 path.py
~/Desktop/Python_Scripting
❯ tree -a /tmp/demo/
/tmp/demo/
├── Documents
│   └── .tmp
├── Downloads
│   └── .cache
└── Pictures
    └── .thumbnails

6 directories, 0 files
```

file handling

- file handling in python comes with many very useful convenience functions

- file handles are usually aquired using the ‘with’ context manager and the ‘open’ function

- most common open modes are (for more see spec): ‘w’ for write, ‘r’ for read, ‘a’ for append

- the context manager ensures after the block is done, the file handle is properly closed and if stuff goes wrong, you get a meaningful stack trace from the program

- file handling goes well together with the previously mentioned pathlib module

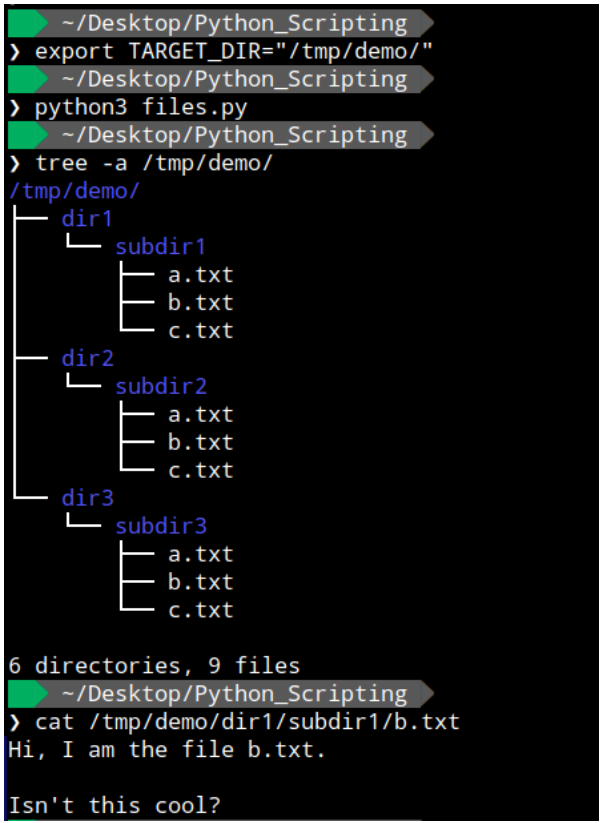
file handling

```
 8  home = os.environ.get("TARGET_DIR")
 9
10  basepath = Path("/tmp" if not home else home)
11
12  folders = [("dir1", "subdir1"),
13             ("dir2", "subdir2"),
14             ("dir3", "subdir3")]
15
16  files = ["a.txt", "b.txt", "c.txt"]
17
18  # we iterate like in path.py, but just have the list folders, which is a list of tuples now
19  # python allows us to 'unpack' these the same way when we used zip
20  for t, s in folders:
21      fullpath = basepath / t / s
22      # calling makedirs with exist_ok set to true behaves like the well know 'mkdir -p'
23      os.makedirs(fullpath, exist_ok=True)
24      for file in files:
25          # open the filepath for writing,
26          # this will truncate the file contents if it existed before
27          with open(fullpath / file, 'w') as filehandle:
28              filehandle.writelines([f"Hi, I am the file {file}.\n", "\n", "Isn't this cool?\n"])
```

files.py

file handling

Example result of the files.py script:



user input

- except using env variables, there is several options to input things into the program either using e.g. command line arguments, config files or interactive input

- interactive textual input on the CLI is aquired using the ‘input’ function

- we can also get fancier and build a command-line UI using e.g. pydialog (not part of the standard library, must be installed via os-package or pip), docs are here: https://pythondialog.sourceforge.io/doc/widgets.html

- python offers easy input for all kinds of standard config and data formats including but not limited to: JSON, INI, TOML, YAML, CSV, XLS(X) (some of them require extra packages which must be installed to work)

- to build a proper command line tool with flags and positional arguments or subcommands, using the module argparse is strongly recommended

user input - input function

```python
#!/usr/bin/env python3

name = ""

# an empty string evaluates to false in python, so we keep asking until we get something
while not name:
    name = input("What is your name? Tell me: ")

# the .format method is another way of filling variables into string and was the main
# way before f""-strings were introduced, it can still be useful when you want to fill
# longer expressions in or want them in multiple places, as it supports naming the arguments

# without naming, just by position
print("Hello {}!".format(name))
# with naming in 2 positions, converting the name to uppercase first
print("How are you {n}? Can you hear me {n}?".format(n=name.upper()))
```

```
~/Desktop/Python_Scripting
❯ python3 input.py
What is your name? Tell me:
What is your name? Tell me: Markus
Hello Markus!
How are you MARKUS? Can you hear me MARKUS?
```

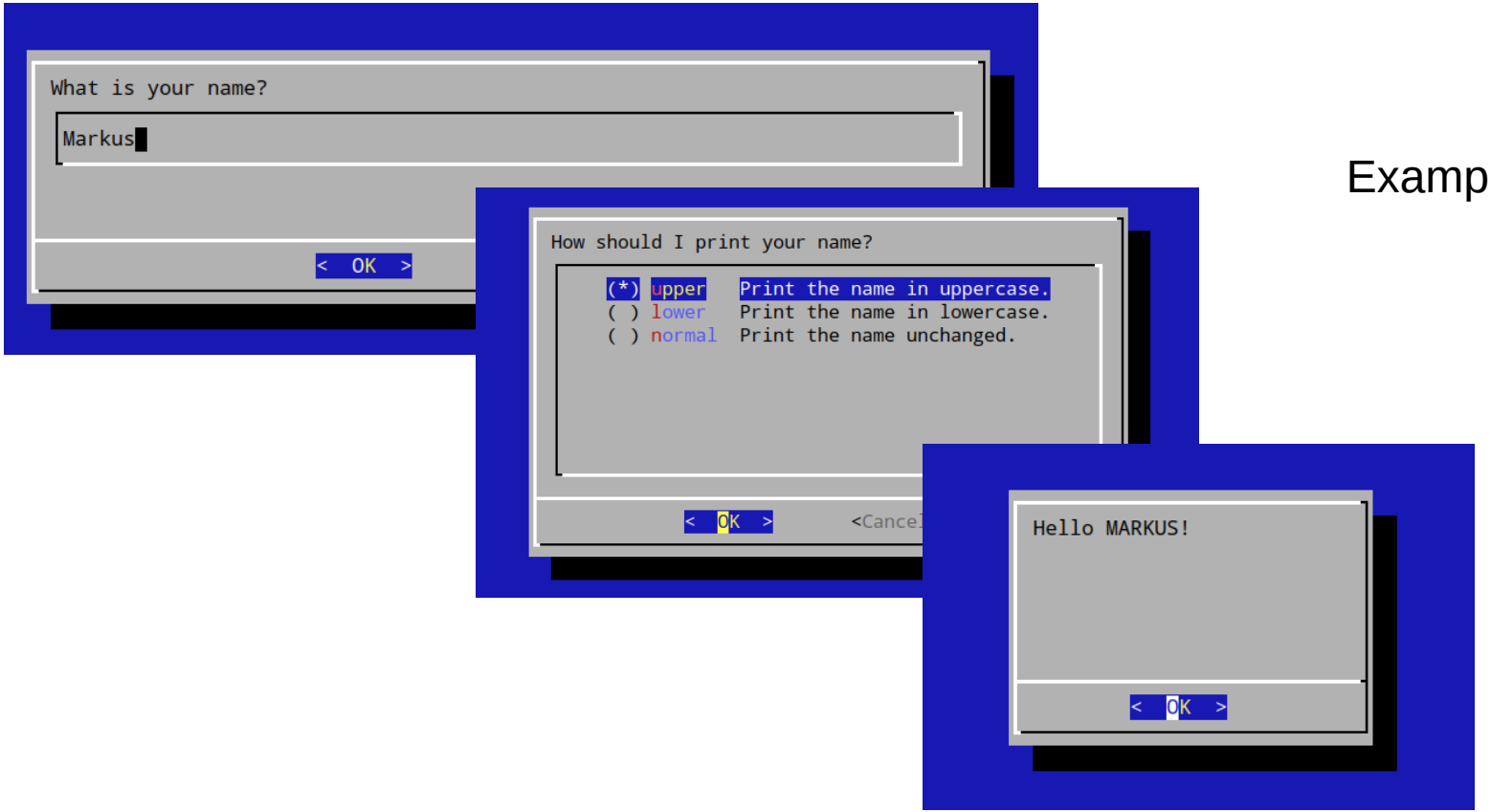
input.py

user input - pydialog

```python
#!/usr/bin/env python3
from dialog import Dialog
import sys

d = Dialog()
name = ""

# now we use a nice TUI-input-box for asking
# check the Dialog documentation, there is several cool widgets like checklists, yes/no dialogs,
# comboboxes etc. to really easily build e.g. a simple configuration TUI
while not name:
    code, name = d.inputbox(text="What is your name?", width=80)
    if code == d.CANCEL:
        # cancel has been pressed, exit
        sys.exit(0)

# we now ask how we should print the name
code, choice = d.radiolist(text="How should I print your name?", choices=[
    ("upper", "Print the name in uppercase.", True),
    ("lower", "Print the name in lowercase.", False),
    ("normal", "Print the name unchanged.", False),
])

if code == d.CANCEL:
    sys.exit(0)
if choice == "upper":
    name = name.upper()
elif choice == "lower":
    name = name.lower()

d.msgbox(text=f"Hello {name}!", width=30)
```

dialog_input.py

user input - pydialog



Example output for dialog_inp

user input - configparser ini file

```python
#!/usr/bin/env python3
from configparser import ConfigParser
import sys

# in this example we use configparser, which supports an INI-like structure
# there is also support for default and fallback values etc.
# see the full spec here: https://docs.python.org/3/library/configparser.html
cfg = ConfigParser()
cfg.read("config.ini")

if not "credentials" in cfg:  # some error handling is always good practice
    print("Error: No credentials section in config.ini .")
    sys.exit(1)
else:
    cred = cfg["credentials"]

name = cred.get("name")
age = cred.get("age")
city = cred.get("city")
if not name or not age or not city:
    print("Error: Malformed config, name, city or age missing.")
    sys.exit(1)

print(f"{name} lives in {city} and is {age} years old.")
```

config_input.py

```
[credentials]
name = Markus
age = 33
city = Dresden
```

config.ini

Example output:

```
~/Desktop/Python_Scripting
❯ python3 config_input.py
Markus lives in Dresden and is 33 years old.
```

user input - argparse CLI interface

```python
#!/usr/bin/env python3
import argparse

parser = argparse.ArgumentParser(
                    prog = 'argparse_input.py',
                    description = 'Shows you what nice things argparse can do!',
                )
# positional, required arguments
parser.add_argument('name', type=str, help="Name of the caller.")
parser.add_argument('age', type=int, help="Age of the caller.")
# optional flag argument that does not take a value (true when given)
parser.add_argument('-u', '--uppercase', action="store_true")

args = parser.parse_args()

# uppercase the name if the flag has been given
if args.uppercase:
    name = args.name.upper()
else:
    name = args.name
# age is an integer, but python knows by itself to call the string representation of it
print(f"{name} is {args.age} years old.")
```

argparse_input.py

user input - argparse CLI interface

```
~/Desktop/Python_Scripting
 1 ❯ python3 argparse_input.py -u Markus 33
MARKUS is 33 years old.
~/Desktop/Python_Scripting
❯ python3 argparse_input.py Markus 33
Markus is 33 years old.
~/Desktop/Python_Scripting
❯ python3 argparse_input.py Markus
usage: argparse_input.py [-h] [-u] name age
argparse_input.py: error: the following arguments are required: age
~/Desktop/Python_Scripting
 2 ❯ python3 argparse_input.py Markus Bla
usage: argparse_input.py [-h] [-u] name age
argparse_input.py: error: argument age: invalid int value: 'Bla'
~/Desktop/Python_Scripting
 2 ❯ python3 argparse_input.py -h
usage: argparse_input.py [-h] [-u] name age

Shows you what nice things argparse can do!

positional arguments:
  name             Name of the caller.
  age              Age of the caller.

options:
  -h, --help       show this help message and exit
  -u, --uppercase
```

Example output for argparse_input.py

- checks type and presence of args
- auto-provides help

subprocess-module - running programs

- subprocess is part of the python standard lib and the main way to run programs from your script
- provides a clean interface to get STDOUT, STDERR, exit codes, create pipes
- large amount of parameters to control execution, e.g.:
  - cwd to set the current working directory the process runs in
  - env to pass a possibly manipulated environment
  - stdout=PIPE, stderr=PIPE or capture_output=True (for both) to capture program outputs
  - shell to request execution inside a shell context (avoid if not explicitly needed, because security)
  - input parameter to construct pipes by sending in the output of a previous command
- documentation at: https://docs.python.org/3/library/subprocess.html

subprocess-module - running programs

```python
#!/usr/bin/env python3
from pathlib import Path
from subprocess import run
import sys

# here we use the Python magic 'dunder' (double underscore) __file__, which is the
# location of the current python file
# using pathlib, we can easily get the directory where we are located
mydir = Path(__file__).parent

# arguments must be passed as a list with spaces omitted between them
# this is to ensure proper argument parsing and avoid any injection attacks
# capture_output does not just print the ls on the console, but collects everything
# for later use
# cwd sets the current working directory where the command should run
result = run(["ls", "-lah"], capture_output=True, cwd=mydir)
# try this line to see stderr
# result = run(["ls", "-lah", "doesnotexist"], capture_output=True, cwd=mydir)

# print also can take an arbitrary number of arguments, handy for something like this:
print("STDOUT: ", result.stdout.decode())

# stdout and stderr come as raw byte strings, so if you expect text you have to decode
# them, the default is usually fine (UTF-8)
print("STDERR: ", result.stderr.decode())

# we can collect exitcodes and pass it on upwards
print("Exit Code: ", result.returncode)
sys.exit(result.returncode)
```

subprocesses.py

subprocess-module - running programs

Example result of the subprocesses.py script:

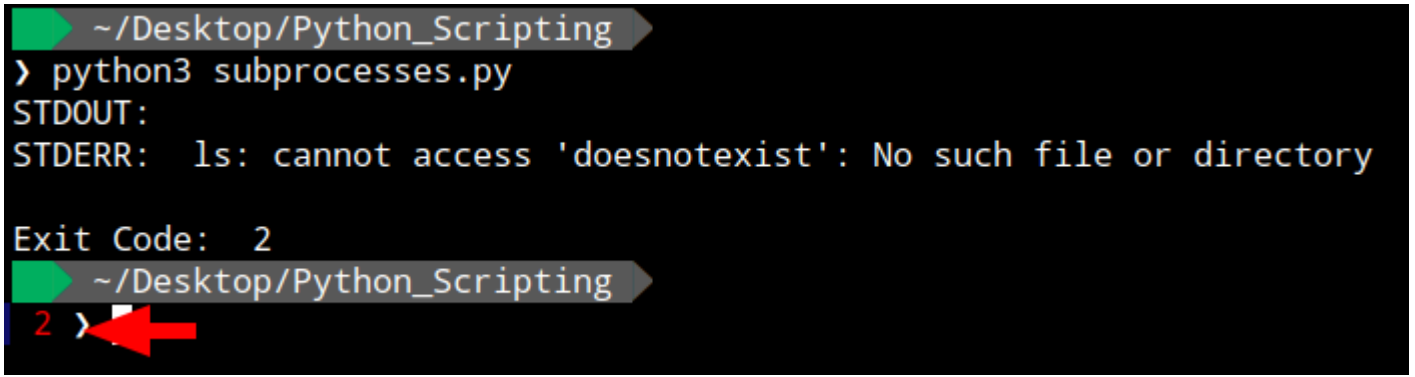
```
~/Desktop/Python_Scripting
 1 ❯ python3 subprocesses.py
STDOUT:  total 48K
drwxr-xr-x.  3 mkrause mkrause  4.0K Jan 16 09:22 .
drwxr-xr-x. 18 mkrause gerusers 4.0K Jan 14 03:08 ..
-rw-r--r--.  1 mkrause mkrause   777 Jan 13 23:33 argparse_input.py
-rw-r--r--.  1 mkrause mkrause    52 Jan 13 21:59 config.ini
-rw-r--r--.  1 mkrause mkrause   766 Jan 13 22:52 config_input.py
-rw-r--r--.  1 mkrause mkrause   921 Jan 13 22:52 dialog_input.py
-rw-r--r--.  1 mkrause mkrause  1021 Jan 13 22:52 files.py
-rw-r--r--.  1 mkrause mkrause   664 Jan 13 20:59 input.py
-rw-r--r--.  1 mkrause mkrause   982 Jan 13 18:31 path.py
drwxr-xr-x.  2 mkrause mkrause  4.0K Jan 16 09:21 __pycache__
-rw-r--r--.  1 mkrause mkrause  1020 Jan 16 10:10 subprocesses.py
-rw-r--r--.  1 mkrause mkrause   731 Jan 13 22:52 sys_os.py

STDERR:
Exit Code:  0
```

subprocess-module - running programs

When using the other call that results in an error, we can see the STDERR and Exit code

```
17  # try this line to see stderr
18  result = run(["ls", "-lah", "doesnotexist"], capture_output=True, cwd=mydir)
```

Example result of the subprocesses.py script with STDERR:



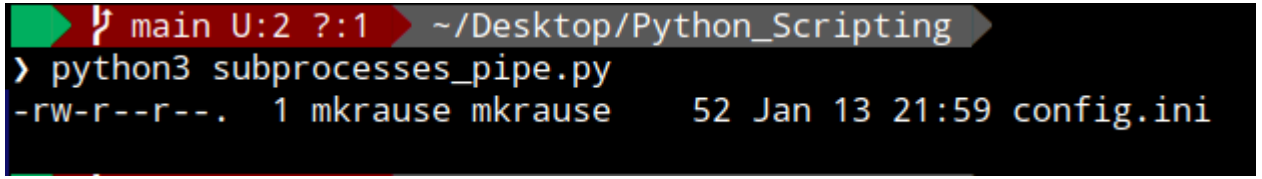
subprocess-module - constructing pipes

Use the input parameter to construct a pipe into another program call

```python
#!/usr/bin/env python3
from pathlib import Path
from subprocess import run
import sys

mydir = Path(__file__).parent
result = run(["ls", "-lah"], capture_output=True, cwd=mydir)

# when ls was successful, we run grep on the output by sending it into the stdin of grep
# this is basically the equivalent of a unix shell pipe, you could also send in stderr,
# which would then equate to a pipe in a unix shell with redirection
if result.returncode == 0:
    grep_result = run(["grep", "config.ini"], capture_output=True, input=result.stdout)
    print(grep_result.stdout.decode())
```

subprocesses_pipe.py

Example output:



closing remarks

- use the official python docs and an editor with proper formatting and completion, we only scratched the surface here, all the collections, modules and types have loads more nice functionality that makes the python standardlib one of the best out there

- if you use leading edge stuff like the new switch-case statement in 3.10, make sure the systems you are targeting can support that (as said, in IT anything conforming to 3.6 or 3.8 for newer ones is a good call right now)

- educate yourself on Python environment tools like conda, pipenv, poetry etc., these can give you reproducible environments where you can test your stuff against a fixed version and set of dependencies

- don’t be ashamed to google the ‘pythonic’ way. usually for certain things there is a recommended and sleek way to do it, which the community happily communicates.

- especially for scripting and IT purposes, stick to standard CPython and keep your dependencies in check, this will keep your stuff portable and easily deployable

THANK YOU

You can find all the files of the tutorial and this presentation on GitHub:

https://github.com/markusdd/python_scripting_tutorial