Scientific Software Development with Python

Python standard library

Simon Pfreundschuh
Department of Space, Earth and Environment

CHALMERS
UNIVERSITY OF TECHNOLOGY

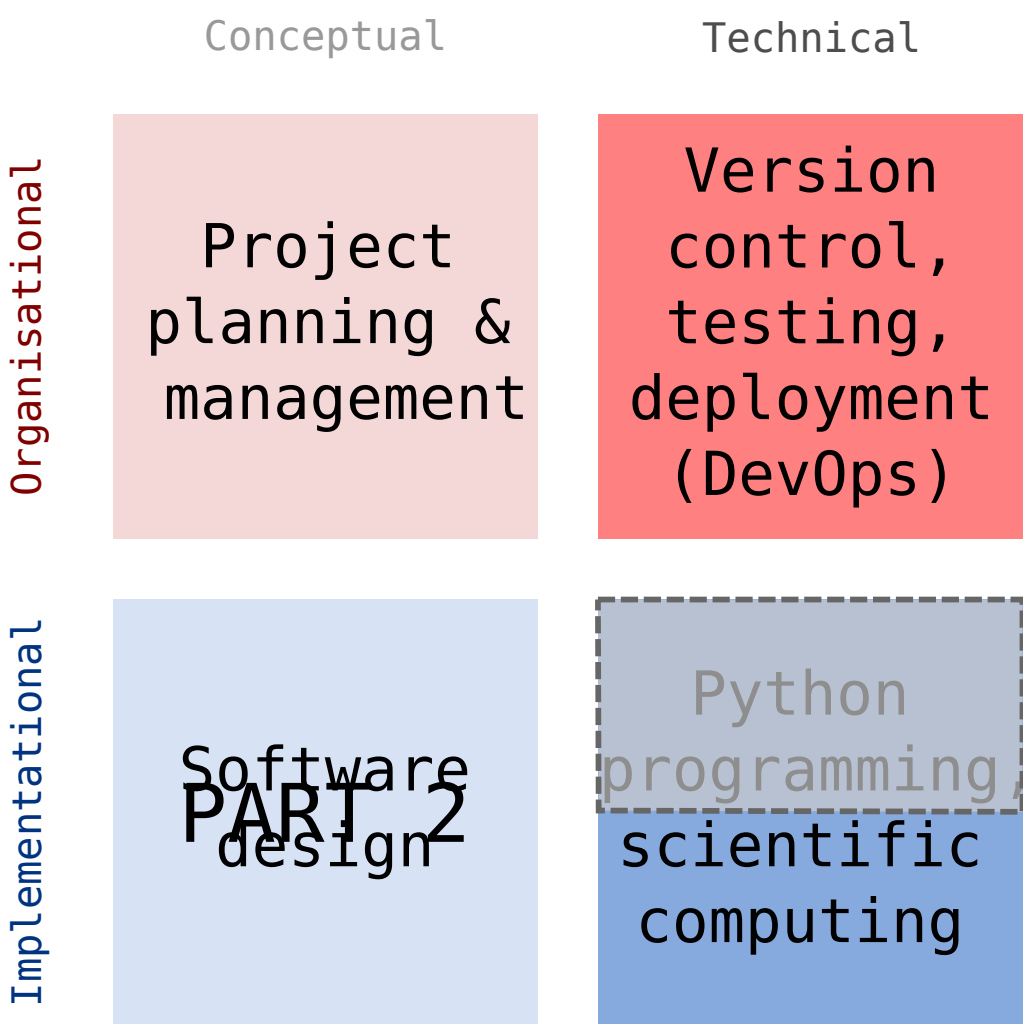
Agenda

1. Overview

2. Data structures

3. A brief tour of the standard library

Overview


Conceptual
Technical
Organisational
Implementational
Project planning & management
Version control, testing, deployment (DevOps)
Software design
PART 2
Python programming
scientific computing

Agenda

1. Overview

2. Data structures

3. A brief tour of the standard library

Classes or data structures?

Classes in object oriented programming

- Define data and associated behavior

What if there is no associated behavior?

- Then defining a class is needlessly verbose.
- Python provides specialized *data structures* to store and retrieve data in different use cases.

Tuple

- A tuple stores a sequence of values of arbitrary types:

```
record = (1, "name", [])
```

- Tuples are *immutable*:
  - An existing `tuple` can't be changed.
  - But it can be used as key in a `dict`
- `tuples` can be unpacked:

```
id, name, properties = record
```

The problem with tuples

- No inherent meaning of different tuple elements:
  - Hard to guess what different elements mean
  - Easy to make an error during unpacking

Solution

- Named tuples:

```
from collections import namedtuple
record_class = namedtuple("Record", ["id" ,"name", "properties"])
record = record_class(1, "name", [])
print(record) # Prints: Record(id=1, name='name', properties=[])
```

Named tuples

```
from collections import namedtuple
record_class = namedtuple("Record", ["id" ,"name", "properties"])
print(record) # Prints: Record(id=1, name='name', properties=[])
```

What's going on here?

- `namedtuple(typename, field_names, ...)` is a factory method that produces a new `class` with the name given by the `typename` argument[1].
- The constructor of the newly created `Record` class expects one value for each of the names in `fieldnames`.
- The newly created class automatically has a useful implementation of the `__repr__` special method.

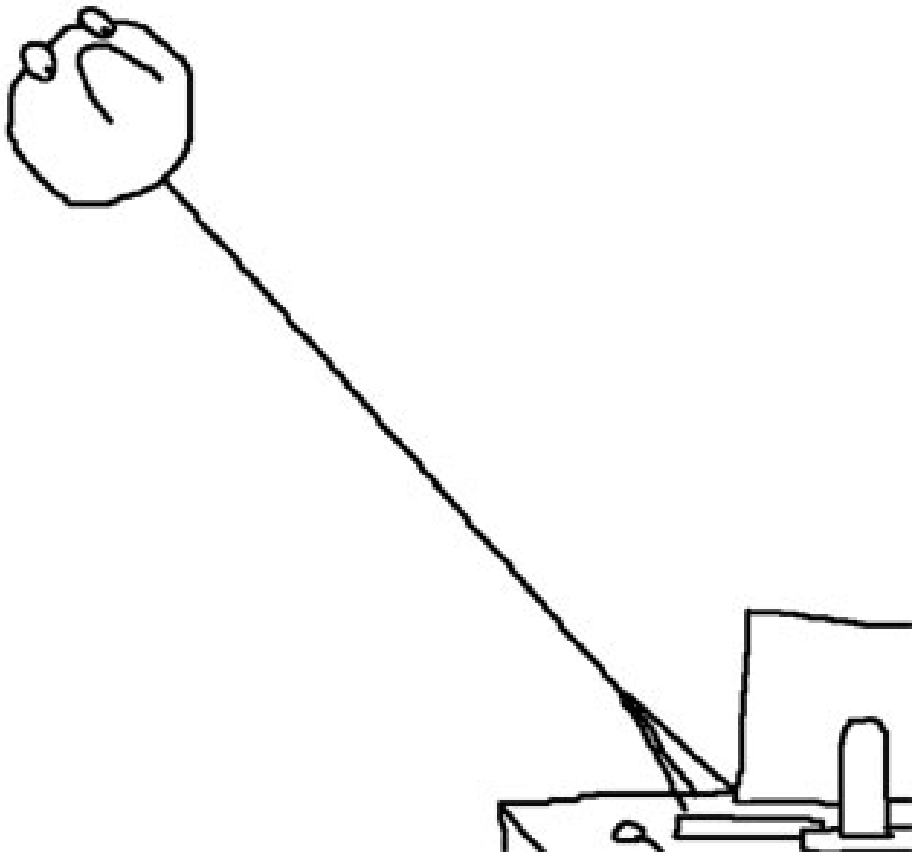
[1] Yes, even classes are first-class objects in Python.

Another solution: Dataclasses

```
from dataclasses import dataclass
@dataclass
class Record:
    id: int
    name: str
    properties: list

record = Record(1, "name", 2)
print(record) # Prints: Record(id=1, name='name', properties=[])
```

Dataclasses



```
from dataclasses import dataclass
@dataclass
class Record:
    id: int
    name: str
    properties: list
```

What's going on here?

- Python 3.5 introduced type annotations[2]:

```
a : int = 1 # This is valid >= Python 3.5 code
```

- The `dataclass` decorator parses the variable annotations and turns them into attributes of the class.

[2] We'll see more of them next lecture.

Default values:

```
from dataclasses import dataclass, field
@dataclass
class Record:
    id: int = 1
    name: str = "name"
    properties: list = field(default_factory=list)
record()
print(record) # Prints: Record(id=1, name='name', properties=[])
```

Exercise 1

- Exercise 1 from exercise sheet
- Time: 5 minutes

The problem with mutable default values

- Default values are created once, when the function definition is parsed.
- **The default values are shared between different invocations of a function**.
- If a mutable default value is changed, these changes affect subsequent calls of the function.

Namedtuples vs. Dataclasses

Namedtuples

- Immutable:
  - Can be used as key in dict.
- Smaller memory footprint than dataclasses

Dataclasses

- More intuitive syntax than named tuples.
- Can add customized class methods and use inheritance

Dataclass with customized behavior

```python
from dataclasses import dataclass
@dataclass
class Record:
    id: int
    name: str
    properties: list = field(default_factory=list)

    def __add__(self, other):
        """ A not very meaningfull addition operator. """
        if isinstance(other, Record):
            Record(self.id, self.name, self.properties + other.properties)
        # Should return NotImplemented if we can't handle type.
        return NotImplemented

record_1 = Record(1, "name", ["proerty 1"])
record_2 = Record(2, "other name", ["proerty 2"])
print(record_1 + record_2)
# Prints: Record(id=1, name='name', properties=['proerty 1', 'proerty 2'])
```

Dictionary

- Container that maps a *key* object to a *value* object.
- Key object must be immutable (*hashable*)
- Highly optimized data structure:
  - Should always be used when certain non-`int` values need to be mapped to arbitrary other values.
  - Used internally by all Python objects that support dynamic attributes and accessible through the `__dict__` special attribute:

```
def a_function():
    pass
a_function.attribute = "some value"
print(a_function.attribute) # Prints: some value
print(a_function.__dict__)  # Prints: {"attribute" : "some_value"}
```

Useful functions

- `get(key, default=None)`: If `key` is present, returns `value` corresponding to `key` otherwise returns `default`
- `setdefault(key, default=None)`: Like `get` but also adds `key` with `default` as value to the `dict` if not already present.
- Iterating over `dict` content: `keys()`, `values()` `items()`

Example

```
scores = {}
current_score = scores.setdefault("player_1", 0)
scores["player_1"] = current_score + 1
print(scores) # Prtins: {'player_1': 1}
```

Other features

- Since Python 3.7: Iterators return elements in order of insertion
  - Use `collections.OrderedDict` in older code if required
- Other specialized dictionary types: `defaultdict` and `Counter` in `collections` module[3].

[3] Check docs for more info:
`https://docs.python.org/3.8/library/collections.html`

Example

- Download text from wikipedia:

```
import urllib.parse
import urllib.request
import json
url = "https://en.wikipedia.org/w/api.php"
values = {"action": "parse",
          "page": "Das Kapital",
          "format": "json",
          "prop": "wikitext"}
data = urllib.parse.urlencode(values)
request = urllib.request.Request(url, data.encode())
with urllib.request.urlopen(request) as response:
  data = json.load(response)
text = data["parse"]["wikitext"]["*"]
```

- Count letters:

```
from collections import Counter
counter = Counter(text)
print(counter.most_common(5))
# Prints: [(' ', 3962), ('e', 2540), ('a', 2102), ('t', 2064), ('i', 2058)]
```

Lists

- Container type designed to hold sequences of objects similar types.[4]

```
numbers = [1, 2, 3, 4]
```

Some useful member functions:

- `append(x)`: Append `x` to list.
- `insert(x, i)`: Insert `x` at index `i`.
- `remove(x)`: Remove first occurrence of `x`
- `index(x)`: Zero-based index of first element equal to `x`
- `sort()`: Sort list

[4] If you find yourself adding values of fundamentally different types to a list, chances are your are using them incorrectly.

Customizing `sort`

```
from dataclasses import dataclass, field
@dataclass
class Record:
    id: int
    name: str
    properties: list = field(default_factory=list)

    def __lt__(self, other):
        """Compares two records using their id attribute."""
        return self.id < other.id

record_1 = Record(1, "name", ["proerty 1"])
record_2 = Record(2, "other name", ["proerty 2"])
print(record_1 < record_2) # Prints: True
```

Customizing `sort`

- `list.sort()` uses the < operator to compare objects
- For user-defined classes, the < is implemented by the `__lt__` special method.

```
record_1 = Record(1, "name", ["proerty 1"])
record_2 = Record(2, "other name", ["proerty 2"])
records = [record_2, record_1]
records.sort()
print(records[0].id) # Prints: 1
```

Set

- Container for unique objects

```
numbers = {1, 1, 2, 2, 3, 3}
print(numbers) # Prints: {1, 2, 3}
```

Useful functions:

- `union()` (or | operator): Union of two sets
- `intersect()` (or & operator): Intersection of two sets
- `difference()` (or - operator): Elements in first but not in second set
- `symmetric_difference()` (or ^ operator): Elements neither in first nor in second set.

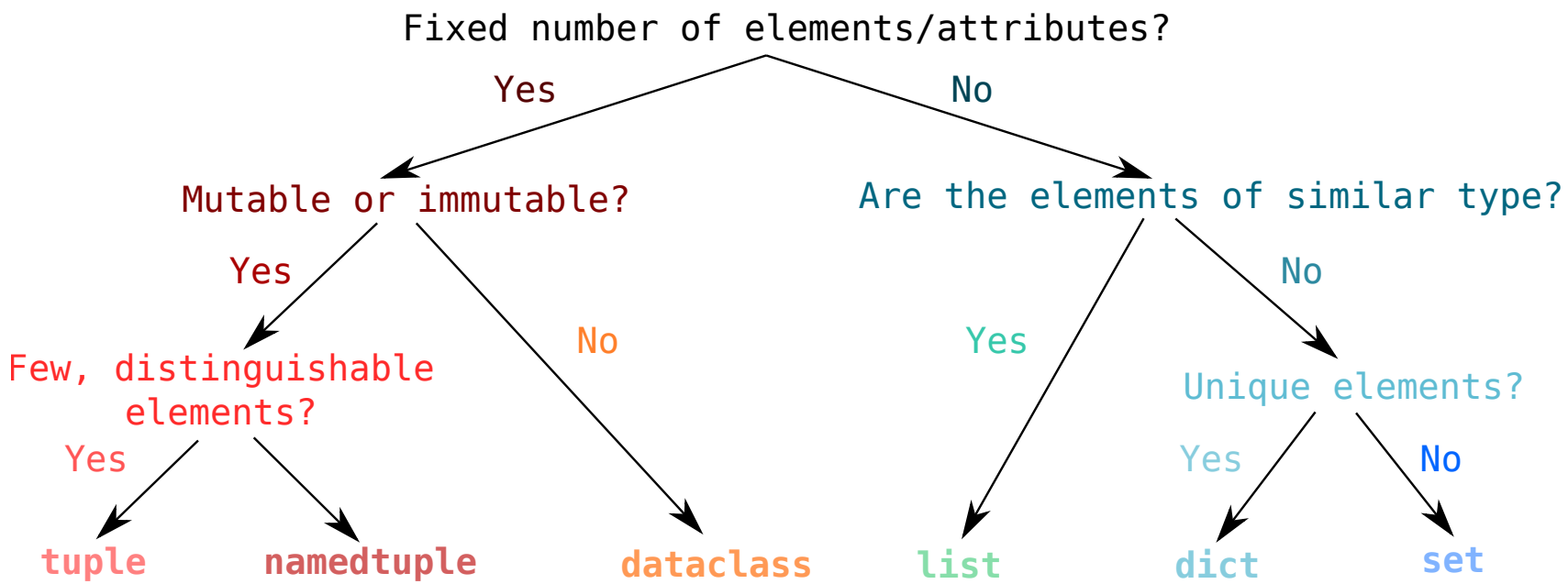
Exercise 2

- Exercise 2 on exercise sheet.
- Time: 10 minutes

Classes vs. data structures

- If data has associated behaviour, make it a class
- Else use a data structure.

Data structure overview



Agenda

1. Overview

2. Data structures

3. A brief tour of the standard library

The Python standard library

- Python comes with an extensive standard library,[5] which is available on any system without the need to install any additional packages.
- Offers solutions for common programming problems.
- Most features are portable between operating systems (linux, windows, mac)

[5] Documented in full detail here: `https://docs.python.org/3/library/`

Built-in functions

- As the name suggests, built-in functions are always available without requiring any additional imports
- For complete list of built-in functions see: `https://docs.python.org/3/library/functions.html`

Some examples:

- `any` and `all`:

```
all([True, False]) # Evaluates to False
any([True, False]) # Evaluates to True
```

Some examples (cont'd):

- `eval`, `exec` and `compile` to interactively execute code:

```
a = eval("1 + 1")
print(a) # Prints: 2
```

DANGER

Don't use this with input you are not controlling. This is how computer systems get hacked.[6]

[6] For details refer to https://nedbatchelder.com/blog/201206/eval_really_is_dangerous.html

Some examples (cont'd):

- `locals`, and `globals` to access the local and module scope as dictionary:

```
globals()["my_variable"] = "my_value"
print(my_variable) # Prints: my_value
```

Some examples (cont'd):

- `hasattr`, `getattr` and `setattr` to manipulate attributes using strings:

```
class A: pass
a = A()
setattr(a, "attribute", 1)
print(a.attribute) # Prints: 1
```

Some examples (cont'd):

- `chr` and `ord` to manipulate sequences of letters[7]:

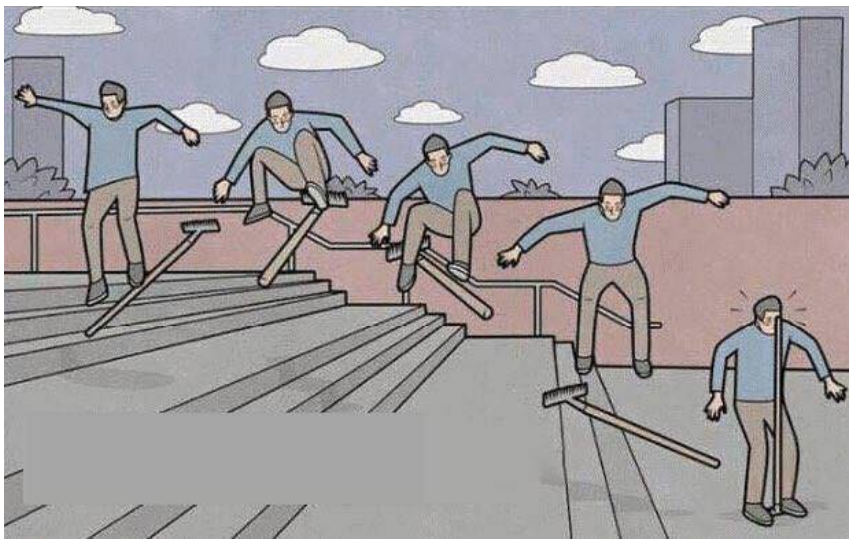
```
letters = [chr(ord("a") + i) for i in range(16)]
print(letters) # Prints: ['a', 'b', ..., 'p']
```

[7] I use this to automatically generate titles for subplots.

Exercise 3

- Exercise 3 on exercise sheet.
- Time: 10 minutes

Manipulating module variables with user input:



Regular expressions

- pattern matching language useful to extract sequences from text
- A regular expression is a string consisting of
  - Regular letters
  - Any of the special characters:

```
. ^ $ * + ? { } [ ] \ | ( )
```

Example: Matching filenames

- Assume you want to identify with the following filenames:

```
file_art_2353.txt
file_ted_12.txt
file_zae_8.txt
file_lpi_9.txt
```

Example: Matching filenames

- Since the first part of the filename is fixed, we can match it using the test as is:

```
file_
```

- This matches:

```
file_art_2353.txt
file_ted_12.txt
file_zae_8.txt
file_lpi_9.txt
```

Example: Matching filenames

- Next, we need to match 3 alphabetic characters
- For, this we need to learn two additional features of regexps:
  - How to match **classes or sets of characters**
  - How to match **repeated characters**

Character classes

- The following special sequences match classes of characters in regular expressions:

`.`	(dot) Matches any character (except newline)
`\d`	Any digit
`\D`	Anything *not matched* by `\d`
`\s`	Any whitespace character
`\S`	Anything *not matched* by `\s`
`\w`	Any alphanumeric (letter or digit) character
`\W`	Anything *not matched* by `\w`

Character classes

- Example:

```
file_\w\w\w
```

- This matches:

```
file_art_2353.txt
file_ted_12.txt
file_zae_8.txt
file_lpi_9.txt
```

- But also:

```
file_123_2353.txt
```

Character sets

- A character set `[...]` is defined using square brackets and may contain:
  - Individual characters: `[abc]` match `a`, `b` or `c`
  - Ranges: `[a-c]`, same as above
  - Character classes
- Character sets can be complemented by adding a `^` in the beginning:
  - `[^...]` matches all characters not matched by `[...]`.

Character sets

- Example:

```
file_[a-zA-Z]
```

- This matches:

```
file_art_2353.txt
file_ted_12.txt
file_zae_8.txt
file_lpi_9.txt
```

Repetitions

- `*`: Matches 0 or more repetitions of the previous expression
  - Example `[a-z]*` matches `""`, `"word"` but not `123`.
- `+`: Matches 1 or more repetitions of the previous expression
- `{n}`: Match exactly `n` repetitions of the previous pattern.

Repetitions

- Example:

```
file_[a-zA-Z]{3}
```

- This matches:

```
file_art_2353.txt
file_ted_12.txt
file_zae_8.txt
file_lpi_9.txt
```

Repetitions

- Example:

```
file_[a-zA-Z]{3}_\d+.txt
```

- This matches:

```
file_art_2353.txt
file_ted_12.txt
file_zae_8.txt
file_lpi_9.txt
```

Using regular expressions

- Python provides built-in support for regular expression via the `re` module.
- Since the \ character has special meaning in Python strings as well as regexps, it is common to use a raw string to define a regular expression.

```
import re
expr = re.compile(r"file_[a-zA-Z]{3}_\d+.txt")
match = expr.match("file_art_2353.txt")
if match:
    print("Filename matches!")
```

Extracting substrings

- Parentheses `(...)` can be used to define groups in a match:
- Example:

```
file_([a-zA-Z]{3})_(\d+).txt
```

  - Defines two groups identified by indices 1 and 2.
- Can be used to extract substrings from match:

```
import re
expr = re.compile(r"file_([a-zA-Z]{3})_(\d+).txt")
match = expr.match("file_art_2353.txt")
print(match.group(1)) # Prints: art
print(match.group(2)) # Prints: 2353
```

Exercise 4

- Exercise 4
- Time: 15 minutes

The problem

- File system paths look different Windows and Linux:
  - Windows:

```
C:\Documents\Report.pdf
```

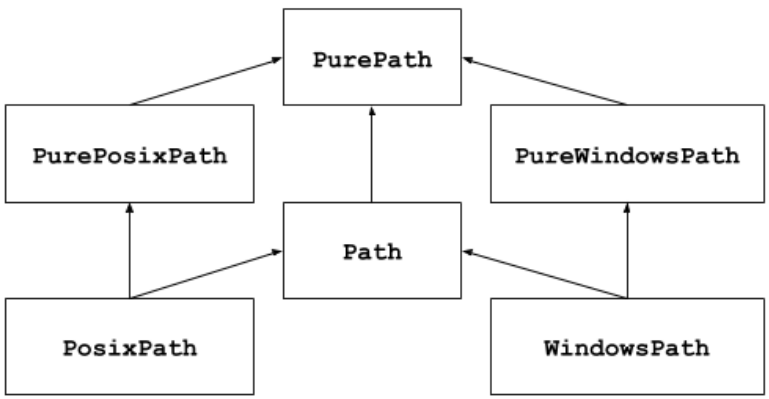
  - Linux:

```
/home/simon/Documents/Report.pdf
```

- By using strings to handle paths your code will likely become *platform dependent* (or/and messy)

The solution

- The `pathlib` module provides an object oriented solution to handle file system paths in a (mostly) platform independent way
- The documentation[8] even contains a simplified UML diagram:



[8] Taken from `https://docs.python.org/3/library/pathlib.html`

The `Path` class

- For mosts tasks simply using the `Path` class is suffcient

Examples:

- A common requirement is to determine the directory that a source file is located in:

```
from pathlib import Path
this_directory = Path(__file__).parent
```

- Or to get the current working directory:

```
this_directory = Path.cwd()
```

More `Path` functionality

- Concatenating paths (/ operator):

```
sub_dir = current_dir / "directory_name"
```

- Iterate over directory content:

```
for p in current_dir.iterdir():
    print(p)
```

- Creating directories:
  - Avoids having to check whether directory already exists

```
sub_dir.mkdir(parents=True, exist_ok=True)
```

Handling dates

`datetime`

- The `datetime` module provides two handy classes to handle dates and times:
  - `datetime`: Represents a point in time defined by date and time
  - `timedelta`: Represents a time difference between to points in time

Useful functions

- Date arithmetic:

```
from datetime import datetime, timedelta
date_1 = datetime(2020, 2, 28)
date_2 = date_1 + timedelta(day=1)
print(date_2) # Prints: 2020-02-29 00:00:00
```

Useful functions

- Parsing of dates using `strptime`[9]:

```
a_date = "27.10.2020" # Germans and their silly dates.
parsed_date = datetime.strptime(a_date, "%d.%m.%Y")
print(parsed_date) # Prints: 2020-10-27 00:00:00
```

- Parsing of dates using `strftime`:

```
a_date = parsed_date.strftime("%d.%m.%Y")
print(a_date) # Prints: 27.10.2020
```

[9] See `https://docs.python.org/3.6/library/datetime.html#strftime-strptime-behavior` for full reference.

The `argparse` module

- Provides an object oriented interface to build command line application.
- Automatically parses command line arguments and displays help messages.

Example

- From the smhpy source code:

```
description = """Display SMHI weather forecasts on the command line."""
parser = argparse.ArgumentParser(prog="smhpy",
                                 description=description)
parser.add_argument('--days',
                    nargs=1,
                    type=int,
                    default=[1],
                    help="Number of days (< 10) to display.")
args = parser.parse_args()
days = args.days[0]
```

Example

- Resulting interface:

```
$ smhpy --help
usage: smhpy [-h] [--days DAYS]

Display SMHI weather forecasts on the command line.

optional arguments:
  -h, --help   show this help message and exit
  --days DAYS  Number of days (< 10) to display
```

The Python standard library

- Contributes a lot to the effectiveness of Python
- Don’t try to reinvent the wheel: A lot of thinking went into designing it, so use it.
- Too complex to cover completely here, so keep your eyes open.

Advantages of the standard library

- Platform independence
- No need to install external packages
- Proven solutions
- Helps you get more done with less code