Topic #1

Advanced Programming Concepts

Type annotations

What is it?

Type annotations specify which type a variable is.

It‘s optional, but very useful!

Example:

```
a: int = 0
b: float = 0.0
a = b     Incompatible types in assignment (expression has type "float", variable has type "int")
```

What is it good for?

PEP 484 – Type Hints (29-Sep-2014):

"This PEP aims to provide a standard syntax for type annotations, opening up Python code to easier static analysis and refactoring, potential runtime type checking, and (perhaps, in some contexts) code generati

Of these

such as n

and refac

[…]

It should

have no c

David's redefinition:

- It is a part of your automatic documentation (like with meaningful variable names). If another person gets your source code they understand it easier.
- Your editor might thank you. Do to some new features in Python 3.10, the modern editors that do syntax highlighting and error checking have a harder time to infer what you mean. The more it need to think about what you mean, the slower your editor might get or even fail to show you syntax highlighting.
- Static code analysis is really helpful. It showed me any problems ahead that I would have figured out the hard way otherwise.
- Packages like the just-in-time compiler numba can produce better results if you can tell it what the variables are.

When and how do we do it?

…mostly when we introduce new variables or define functions or classes:

```python
a: int
b: int = 0
```

```python
def this_is_a_function() -> None:
    pass

def this_is_a_function() -> int:
    return 5

def this_is_a_function(a: int) -> int:
    return a

def this_is_a_function(a: int, b: int = 8) -> int:
    return a + b

def this_is_a_function(a: int, b: int = 8) -> tuple[int, int]:
    return a, b
```

Common types

Simple types:

```
a: int = 0
b: float = 0.0
c: bool = True
d: str = "LaLa"
```

Variables with different types:

```
from typing import Any
a: Any = 0
b: float = 0.0
a = b
```

…could also be explicit list with ‚or' separators…

```
a: None | int = None
```

Generic types:

```
la: list = ["a", 1, 3.3]


ta: tuple = ("a", 1, 3.3)
tb: tuple[str, int, float] = ("a", 1, 3.3)
```

```
la: list[str | int | float] = ["a", 1, 3.3]
```

Functions, more complex types…

```python
import numpy as np
import torch
from typing import Callable


def func() -> None:
    return


a: int = 0
b: np.ndarray = np.zeros((10,))
c: torch.Tensor = torch.zeros((10, 1))
d: Callable = func
```

```python
Callable[[Arg1Type, Arg2Type], ReturnType]
```

```python
from typing import Callable


def function_a(x: int) -> int:
    return x + 1


def function_a_bad(x: int, y: int) -> int:
    return x + y


def function_b(x, other_function: Callable[[int], int]) -> int:
    return other_function(x) ** 2


print(function_b(1, function_a))  # -> 4
```

More information:

https://davrot.github.io/pytutorial/python_basics/python_typing/

Classes

What is a class?

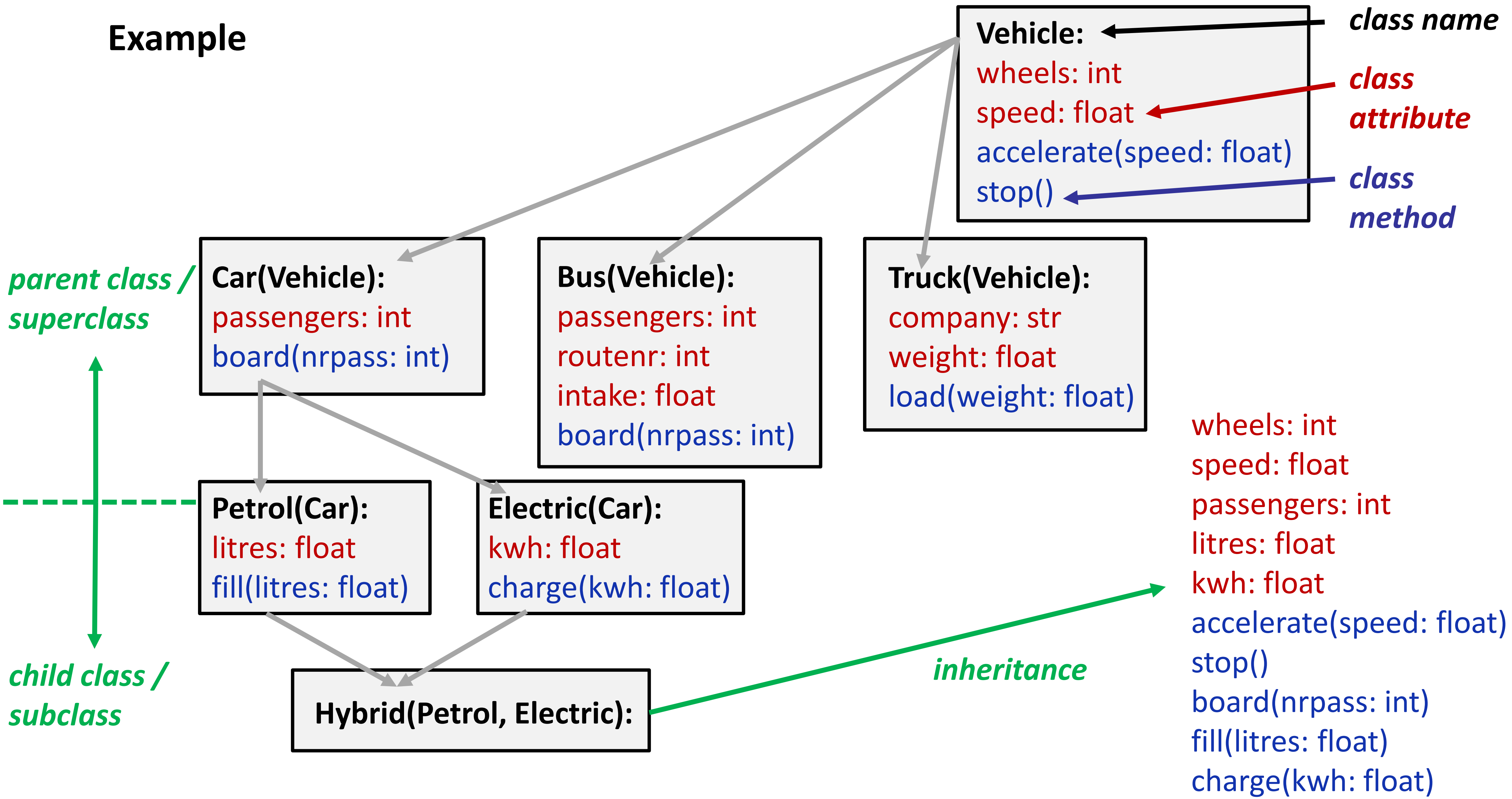
A class is a **container for data (attributes)** and **functions (methods)**. They bundle properties of ‚objects' and actions that can be performed with/on them.

Classes help to **modularize and organize your code** even more than functions.

Class hierarchies are useful for re-using code which is common for different ‚objects'

Dataclasses are specific classes in Python intended to represent data that ‚belongs together'.

Example


Vehicle:
wheels: int
speed: float
accelerate(speed: float)
stop()
class name
class
attribute
class
method
parent class /
superclass
Car(Vehicle):
passengers: int
board(nrpass: int)
Bus(Vehicle):
passengers: int
routenr: int
intake: float
board(nrpass: int)
Truck(Vehicle):
company: str
weight: float
load(weight: float)
Petrol(Car):
litres: float
fill(litres: float)
Electric(Car):
kwh: float
charge(kwh: float)
child class /
subclass
Hybrid(Petrol, Electric):
inheritance
wheels: int
speed: float
passengers: int
litres: float
kwh: float
accelerate(speed: float)
stop()
board(nrpass: int)
fill(litres: float)
charge(kwh: float)

Basic usage

a) Defining and instantiating a class:

```
class SimpleClass(object):
    variable_a: int

instance_a = SimpleClass()
instance_a.variable_a = 1

instance_b = SimpleClass()
instance_b.variable_a = 2

print(instance_a.variable_a)   # -> 1
print(instance_b.variable_a)   # -> 2
```

b) Including and setting attributes (variables):

```
class MostSimplestClass:
    a: int
    b: list

    def __init__(self):
        self.a = 0
        self.b = []
```

please don‘t initialize in attribute section!

c) Including and using methods (functions):

```
class SimpleClass:
    variable_a: int

    def __init__(self) -> None:
        self.variable_a = 1

    def some_method(self, input: int) -> int:
        return self.variable_a + input


instance = SimpleClass()
print(instance.some_method(678)) # -> 679
```

Inheritance

a) adding

```
class BaseClassA:
    a: int = 0


class ClassA(BaseClassA):
    b: int = 1


instance = ClassA()
print(instance.a)   # -> 0
print(instance.b)   # -> 1
```

b) replacing

```
class BaseClassA:
    def print_something(self):
        print("BaseClassA")


class ClassA(BaseClassA):
    def print_something(self):
        print("ClassA")


instance = ClassA()
instance.print_something() # -> ClassA
```

c) multiple inheritance

```
class BaseClassA:
    def print_something(self):
        print("BaseClassA")


class BaseClassB:
    def print_something(self):
        print("BaseClassB")


class ClassA(BaseClassA, BaseClassB):
    pass


class ClassB(BaseClassB, BaseClassA):
    pass


instance_a = ClassA()
instance_a.print_something()   # -> BaseClassA
instance_b = ClassB()
instance_b.print_something()   # -> BaseClassB
```

What else?

There are special methods for certain purposes:

- **__init__:** called always when a class is instantiated. Good for initalizing and setting up a meaningful state for a class instance.
- **__str__:** called by **str(object)** and the built-in functions **format()** and **print()** to compute the “informal” or nicely printable string representation of an object. See also **__repr__** for the built-in function **repr()**.
- **super():** refers to the parent class. Good to call functions from that class. **Example:**

```
class BaseClassA:
    a = 1

    def print_something(self):
        print(f"BaseClassA {self.a}")


class BaseClassB(BaseClassA):
    a = 2

    def print_something(self):
        super().print_something()
        print(f"BaseClassB {self.a}")
```

More information:

https://davrot.github.io/pytutorial/python_basics/class/

Dataclasses

The **dataclass** is very similar to normal classes, but it requires **type annotations** thus serving a good programming style. You have to **import from dataclasses** and use a decorator **@dataclass** when you define a dataclass:

```
from dataclasses import dataclass


@dataclass
class TestClassA:
    name: str
    number_of_electrodes: int
    sample_rate_in_hz: float
    dt: float


data_1 = TestClassA("Dataset A", 100, 1000, 1 / 1000)
print(data_1)
```

there will be an error without these annotations!

A dataclass has an automatic __init__() method which can be used to populate the attributes…

Further Features I

- **default_factory** can be used to specify automatic generation of default values.

```
from dataclasses import dataclass, field


@dataclass
class TestClassA:
    name: str = field(default_factory=str)
    number_of_electrodes: int = field(default_factory=int)
    dt: float = field(default_factory=float)
    sample_rate_in_hz: float = field(default_factory=float)


data_1 = TestClassA()
print(data_1)
```

Output:

```
TestClassA(name='', number_of_electrodes=0,
dt=0.0, sample_rate_in_hz=0.0)
```

Further Features I

```python
@dataclass
class TestClassA:
    name: str
    number_of_electrodes: int = field(kw_only=True, default=42)
    dt: float = field(init=False)
    sample_rate_in_hz: float = 1000.0

    def __post_init__(self) -> None:
        self.dt = 1.0 / self.sample_rate_in_hz

    def __str__(self) -> str:
        output: str = (
            f"Name: {self.name}"
            "\n"
            f"Number of electrodes: {self.number_of_electrodes}"
            "\n"
            f"dt: {self.dt:.4f}s"
            "\n"
            f"Sample Rate: {self.sample_rate_in_hz:.2f}Hz"
        )
        return output
```

- defaults can also be specified in the class definiton (please do not do this to mutables!)
- attributes can be spared from initialization
- attributes can explicitly be specified as keywords

```python
data_1 = TestClassA("Dataset A", 500)
print(data_1)
print("")

data_2 = TestClassA("Dataset B", 500, number_of_electrodes=33)
print(data_2)
```

```
Name: Dataset A
Number of electrodes: 42
dt: 0.0020s
Sample Rate: 500.00Hz

Name: Dataset B
Number of electrodes: 33
dt: 0.0020s
Sample Rate: 500.00Hz
```

- **__post_init__()** if you have to do some init of your own…
- **__str__()** for a nice printout!

Why dataclasses?

- putting data together into meaningful containers…
- appropriate type handling…
- versatile and safe initialization methods…
- **makes comparing data sets easy…!**

```
@dataclass
class MyDataset:
    x: int
    y: int


data_1a = MyDataset(x=1, y=1)
data_1b = MyDataset(x=1, y=1)
print(data_1a == data_1b)
data_2 = MyDataset(x=1, y=2)
print(data_1a == data_2)
```

```
True
False
```

…compare everything

```
@dataclass
class MyDataset:
    x: int
    y: int = field(compare=False)


data_1a = MyDataset(x=1, y=1)
data_1b = MyDataset(x=1, y=1)
print(data_1a == data_1b)
data_2 = MyDataset(x=1, y=2)
print(data_1a == data_2)
```

```
True
True
```

…compare part

More information:

https://davrot.github.io/pytutorial/python_basics/dataclass/

Catch me if you can!

Aspects:

- syntax errors
- logical errors
- data inconsistencies
- exceptions
- inadequate usage or user input
- …

Assert

Assert checks if a condition is true. If not, it issues and error and stops program execution. Example:

```
a: int = 0
assert isinstance(a, float), f"a (value={a}) is not a float."
```

Use it often to make your code safe to use, or to discover inconsistencies in input data!

```
import numpy as np

def solve_quadratic(a: float, b: float, c: float) -> tuple[float, float]:

    assert isinstance(a, float), "argument 'a' must be float!"
    assert isinstance(b, float), "argument 'b' must be float!"
    assert isinstance(c, float), "argument 'c' must be float!"
    assert a != 0, "argument a must be non-zero, otherwise it's not a quadratic equation!"

    sqrt_arg = -4*a*c+b**2
    assert sqrt_arg >= 0, "root argument must be positive for non-imaginary solutions!"

    x1 = (+np.sqrt(-4*a*c+b**2)-b)/2*a
    x2 = (-np.sqrt(-4*a*c+b**2)-b)/2*a
    return x1, x2
```

Try … Except … Else … Finally …

Errors need not terminate your program. Each error raises an exception, and you can catch that exception and handle it properly!

Example for different exceptions…

```
10 * (1 / 0) # -> ZeroDivisionError: division by zero

4 * not_there_variable + 1 # -> NameError: name 'not_there_variable' is not defined

"1" + 1 # -> TypeError: can only concatenate str (not "int") to str

with open("file_that_is_not_there.nope", "r") as fid: # -> FileNotFoundError: [Errno 2] No such file or directory: 'file_that_is_not_there.nope'
    pass
```

…and example for handling an exception nicely:

```
try:
    x = 10 * (1 / 0)
except ZeroDivisionError:
    x = 0

print(x)  #-> 0
```

General form:

```
try:
    ...
except Exception:
    ...
else:
    ...
finally:
    ...
```

try this piece of code…

…**if the specified exception occurred**, execute this piece of code **(‚error handling‘)**…

…**otherwise**, execute this piece of code. You can put code here which runs only correctly when the exception did not occur…

…**in any case**, execute this piece of code, irrespectively of errors having occurred/not occurred **(‚clean-up‘)**. When an unhandled exception occurred, execution stop after this code!

General form, example:

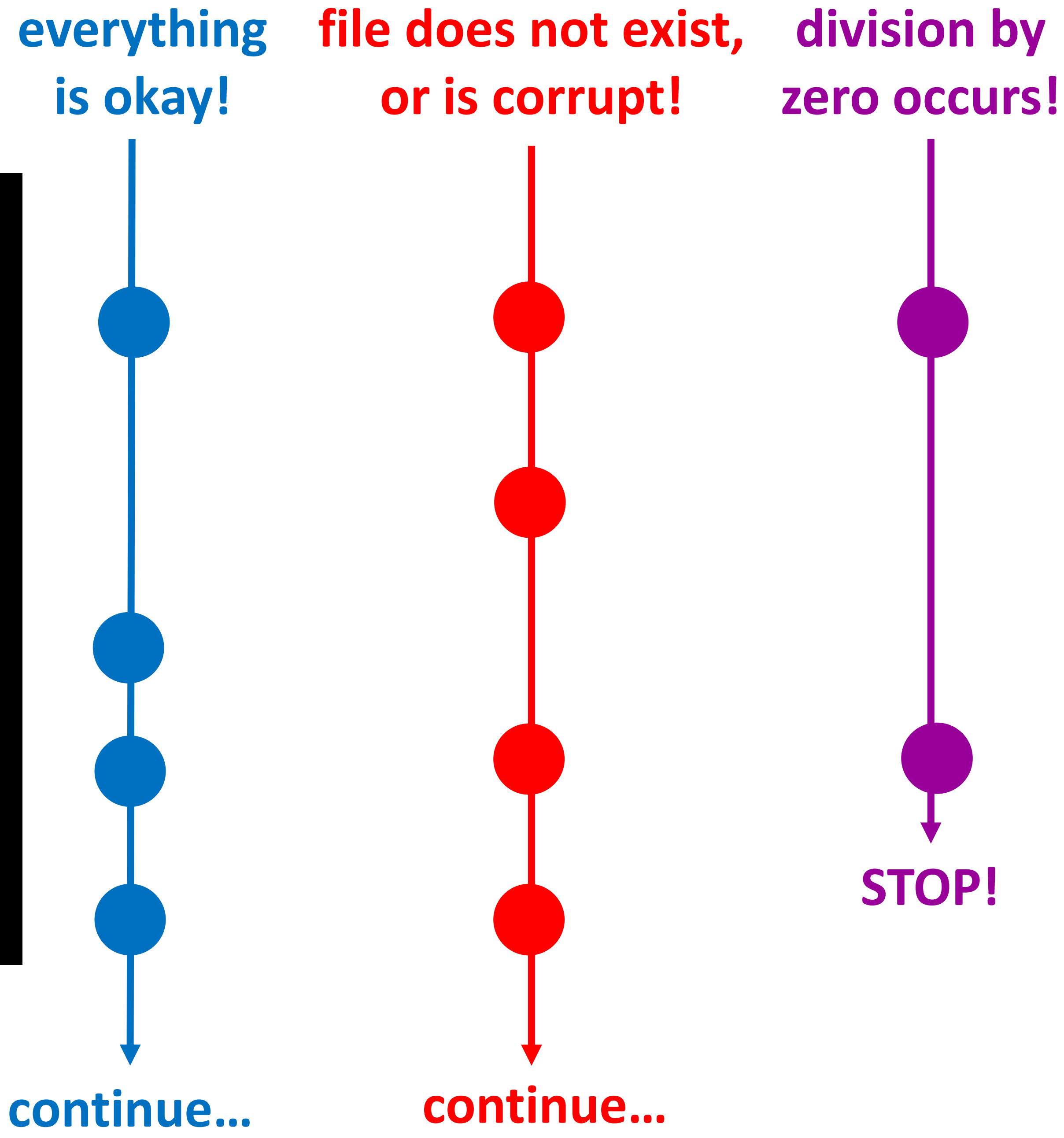
```
for i in range(n_files):
    try:
        # open file[i] for read
        # read neural activity into temp array
        # normalize temp array by its sum

    except OSError:
        # assign None to data[i]

    else:
        # assign temp array to data[i]

    finally:
        # close file if open


print("Successfully read existing data files...")
```



Possible exceptions…

```
BaseException
 ├── BaseExceptionGroup
 ├── GeneratorExit
 ├── KeyboardInterrupt
 ├── SystemExit
 └── Exception
      ├── ArithmeticError
      │    ├── FloatingPointError
      │    ├── OverflowError
      │    └── ZeroDivisionError
      ├── AssertionError
      ├── AttributeError
      ├── BufferError
      ├── EOFError
      ├── ExceptionGroup [BaseExceptionGroup]
      ├── 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
      │    ├── NotImplementedError
      │    └── RecursionError
      ├── StopAsyncIteration
      ├── StopIteration
      ├── SyntaxError
      │    └── IndentationError
      │         └── TabError
      ├── SystemError
      ├── TypeError
      ├── ValueError
      │    └── UnicodeError
      │         ├── UnicodeDecodeError
      │         ├── UnicodeEncodeError
      │         └── UnicodeTranslateError
      └── Warning
           ├── BytesWarning
           ├── DeprecationWarning
           ├── EncodingWarning
           ├── FutureWarning
           ├── ImportWarning
           ├── PendingDeprecationWarning
           ├── ResourceWarning
           ├── RuntimeWarning
           ├── SyntaxWarning
           ├── UnicodeWarning
           └── UserWarning
```

…use those!

…and you can define your own!

Exceptions have associated information in their attributes

```
try:
    x = "1" + 1
except TypeError as e:
    print(type(e))
    print("")
    print(dir(e))
    print("")
    print(e)
    print("")
    print(e.args)
```

```
<class 'TypeError'>

['__cause__', '__class__', '__context__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__', '__gt__', '__hash__', '__init__', '__init_subclass__', '__le__', '__lt__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__setstate__', '__sizeof__', '__str__', '__subclasshook__', '__suppress_context__', '__traceback__', 'args', 'with_traceback']

can only concatenate str (not "int") to str

('can only concatenate str (not "int") to str',)
```

Raising exceptions

When you determine that something goes wrong, you can yourself raise an exception…

…either a **matching, predefined one**

```
raise NameError("My name is NameError")
```

```
# -> NameError: My name is NameError
```

…or a **newly defined exception**…

```
class BaseError(Exception):
    def __init__(self, *args: object):
        super().__init__(*args)


class E1Error(BaseError):
    message: str
    error_value: str

    def __init__(self, *args: object):
        super().__init__(*args)
        self.message = str(args[0])
        self.error_value = str(args[1])


try:
    raise E1Error("User-Error1", "X is not a U")
except E1Error as e:
    print(e.message)
    print("")
    print(e.error_value)
```

Using the debugger

The VSCode debugger lets you follow, monitor, and manipulate the execution of program code…

Examples of actions possible:

- step over, step in, step out
- continue
- breakpoints, conditional breakpoints, function breakpoints
- inspection and change of variables
- monitoring
- …

Interactive demonstration…

More information:

https://davrot.github.io/pytutorial/workflow/vscode_debug/
https://davrot.github.io/pytutorial/python_basics/exceptions/
https://davrot.github.io/pytutorial/python_basics/assert/