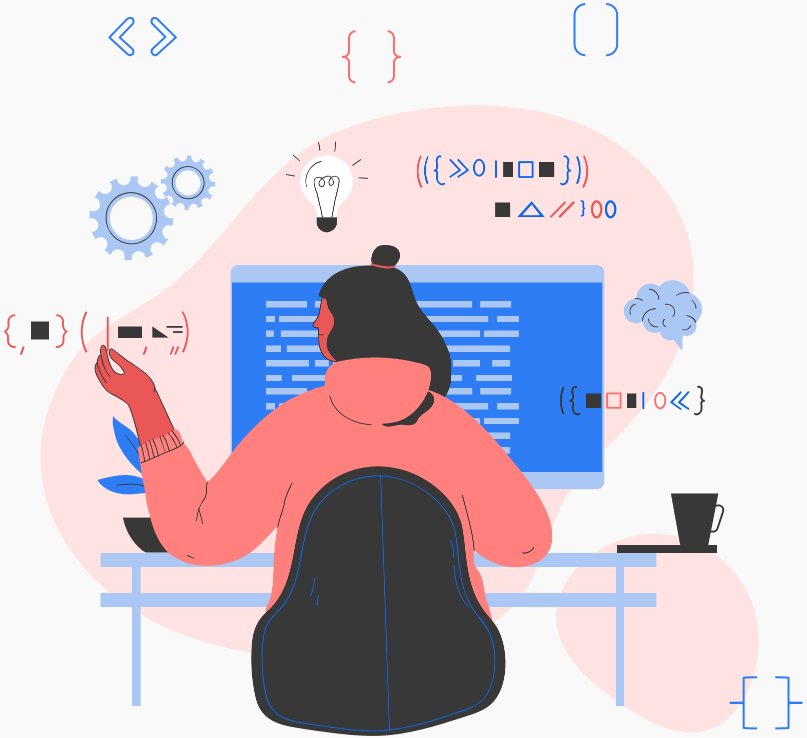
Python 3.11

What's new?

Dilyan Grigorov

EUROPYTHON 17-23 July 2023
PRAGUE & REMOTE



About me

- Python has been my favorite language since 2007
- I had a big pause from programming, but got back during a hard period of my life
- My first programming language is C++
- I'm a student at IU of Applied Sciences - Germany (Computer Science)
- Mentored by Google Engineer
- My interests include Python (Software engineering), Django, Data Science, Machine Learning, AI

Table of contents

01

TypedDicts

02

Self Type

03

Improved Error Messages

04

Asyncio

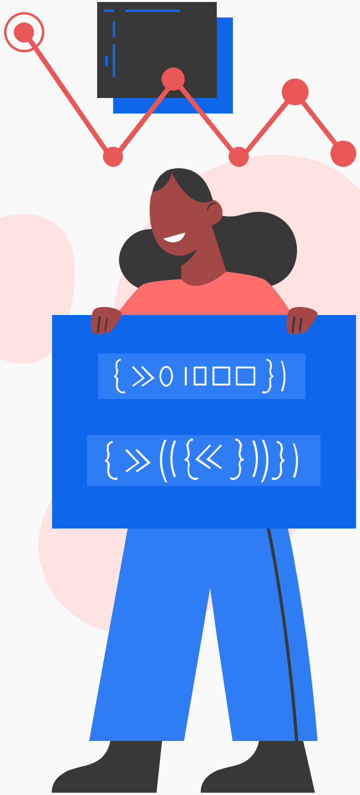


Table of contents

05

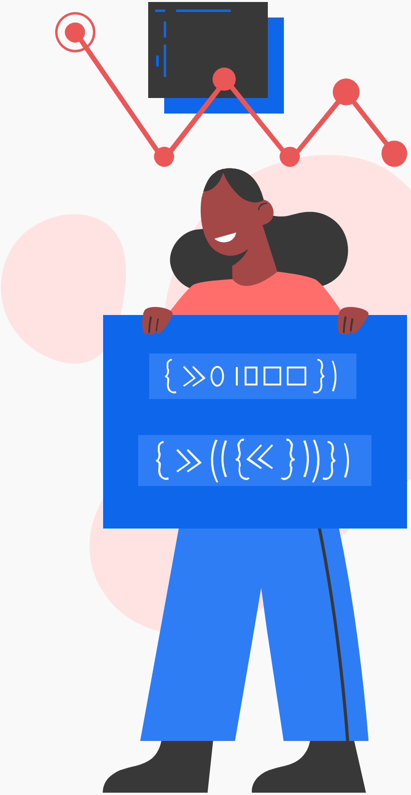
TOML read-only support in stdlib

06

Improved Type Variables

07

Speed Improvements



TypedDict

What is it? Required / NotRequired, Inheritance

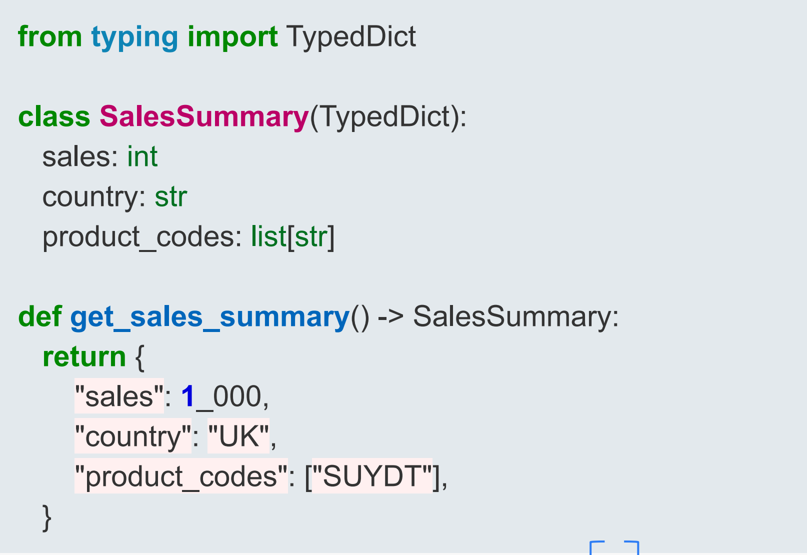
What is a TypedDict?

- TypedDict was specified in PEP 589 and introduced in Python 3.8.
- On older versions of Python you can install it from typing_extensions.
  pip install typing_extensions
- In Python 3.11 it is directly imported from typing.

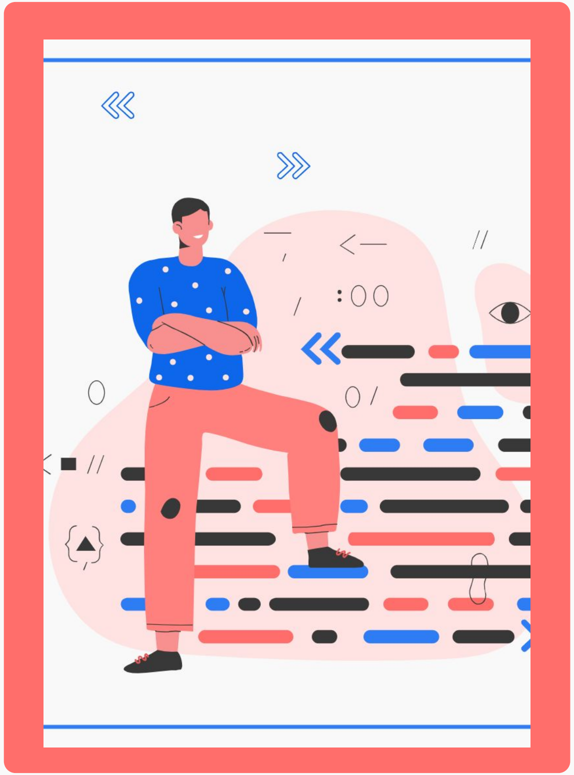
TypedDict or just a Dict?

- The dict [key: value] type lets you declare uniform dictionary types, where every value has non-defined type, and
- arbitrary keys are supported.
- But The TypedDict allows us to describe a structured dictionary
- where the type of each dictionary value depends on the key.

Example



```
from typing import TypedDict

class SalesSummary(TypedDict):
   sales: int
   country: str
   product_codes: list[str]

def get_sales_summary() -> SalesSummary:
   return {
       "sales": 1_000,
       "country": "UK",
       "product_codes": ["SUYDT"],
   }
```



Example 2

```
from typing import TypedDict

#TypedDict Attributes
class Songs(TypedDict):
    name : str
    year : int

song: Songs = {'name': 'We Will Rock You' ,
'year':1977}

#Variable Predefinition
song_info : Songs
song_info = {'name':"we will rock you",'year': 1977}
```

Inheritance: TypedDict cannot inherit from both a TypedDict type and a non-TypedDict base class

```
from typing import TypedDict

class Songs(TypedDict):
    name : str
    year : int

class Band_Name(Songs):
    band_name:str
song_info:Band_Name = {'name': we will rock
you','year':1977,'band_name':'Queen'}

#Multiple Inheritance
class A(TypedDict):
    a: str
class B:
    def __init__(self, a, b):
            self.a = a
            self.b = b
class abc(A, B):
    c: int
```

```
from typing import TypedDict

class Song_info(TypedDict):
    name: str
    year: int

class Band_info(TypedDict):
    no_of_members: int
    lead_singer: str

class Songs(TypedDict):
    songs_info:Song_info
    band: Band_info

result:Songs = {'songs_info':{'name':'we will rock
you','year':
              1977},'band':{'no_of_members':4,
             'lead_singer':'Freddie Mercury'}}
```

Nested TypedDict Example:

```
from typing import TypedDict

class Song_info(TypedDict):
    name: str
    year: int

class Band_info(TypedDict):
    no_of_members: int
    lead_singer: str

class Songs(TypedDict):
    songs_info:Song_info
    band: Band_info

result:Songs = {'songs_info':{'name':'we will rock you','year':
                 1977},'band':{'no_of_members':4,
                'lead_singer':'Freddie Mercury'}}
```

Required[] and NotRequired[]

```
from typing import TypedDict, NotRequired

class User(TypedDict):
    name: str
    age: int
    married: NotRequired[bool]

marie: User = {'name': 'Marie', 'age': 29, 'married': True}
fredrick : User = {'name': 'Fredrick', 'age': 17}  #
'married' is not required
```

Required[] and NotRequired[]

```
from typing import TypedDict, Required

# `total=False` means all fields are not required by default
class User(TypedDict, total=False):
    name: Required[str]
    age: Required[int]
    married: bool  # now this is optional

marie: User = {'name': 'Marie', 'age': 29, 'married': True}
fredrick : User = {'name': 'Fredrick', 'age': 17}  # 'married'
is not required
thomas: User = {'age': 29, 'married': True} # Will be
highlighted because a key is missing!
```

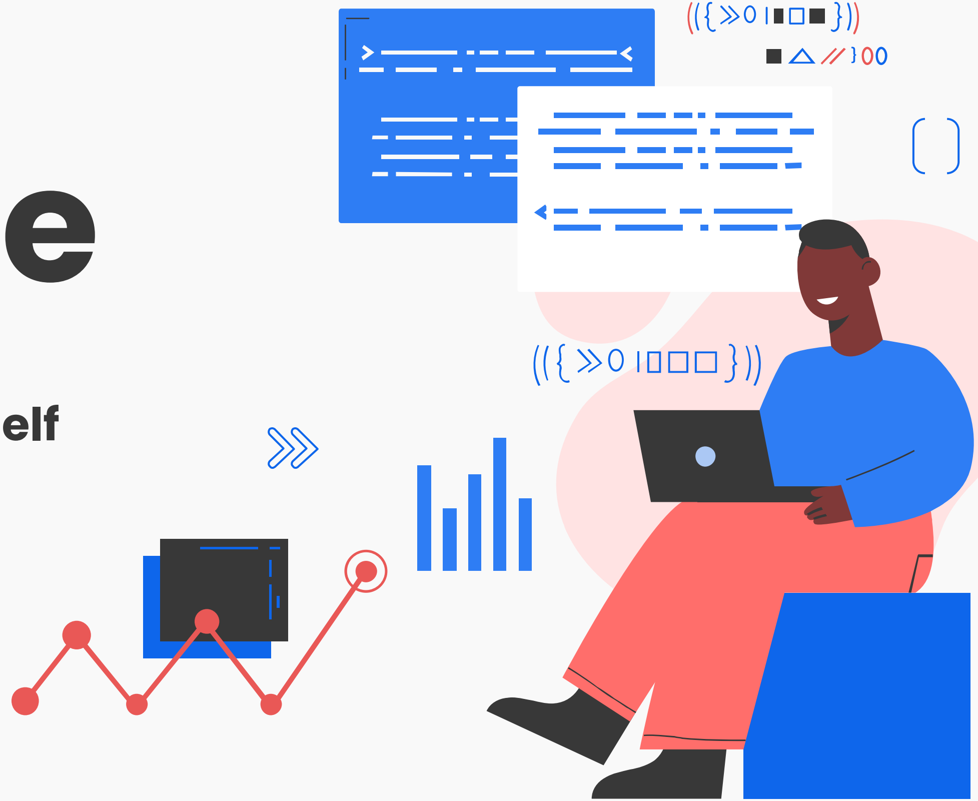
Unary operators + / - / ~ as Required / NotRequired Example

```
class MyThing(TypedDict, total=False):
    req1: +int    # + means a required key, or Required[]
    opt1: str
    req2: +float

class MyThing(TypedDict):
    req1: int
    opt1: -str    # - means a potentially-missing key, or NotRequired[]
    req2: float

class MyThing(TypedDict):
    req1: int
    opt1: ~str    # ~ means a opposite-of-normal-totality key
    req2: float
```

We could use unary + to mark a required key, unary - to mark a potentially-missing key, or unary ~ to mark a key with opposite-of-normal totality

Self type

@classmethod becomes Self



Without Self Type

- Previously, if you had to define a class method that returned an object of the class itself, it would look something like this ->

- To be able to say that a method returns the same type as the class itself, you had to define a TypeVar, and say that the method returns the same type T as the current class itself.

```python
from typing import TypeVar

T = TypeVar('T', bound=type)

class Circle:
    def __init__(self, radius: int) -> None:
        self.radius = radius

    @classmethod
    def from_diameter(cls: T, diameter) -> T:
        circle = cls(radius=diameter/2)
        return circle
```

With Self Type

```
from typing import Self

class Language:

   def __init__(self, name, version, release_date):
       self.name = name
       self.version = version
       self.release_date = release_date

   def change_version(self, version) -> Self:
       self.version = version
       return Language(self.name, self.version,
self.release_date)


lang = Language("Python", 3.11, "November")
lang.change_version(3.12)
print(lang.version)
```

```
from typing import Self

class Car:
   def set_brand(self,
              brand: str) -> Self:
       self.brand = brand
       return self

# Define a child class
class Brand(Car):
   def set_speed(self,
              speed: float) -> Self:
       self.speed = speed
       return self

# Calling object inside print statement
print(Car().set_brand("Maruti"))
print(Brand().set_brand("Maruti").set_speed(110.5))
print(type(Car().set_brand("Maruti")))
print(type(Brand().set_brand("Maruti").set_speed(110.5)))
```

Improved Exceptions

Objects, Properties and JSON

Better Exceptions in Python 3.11

Better Error Messages

Know where your error is!

Exception Notes

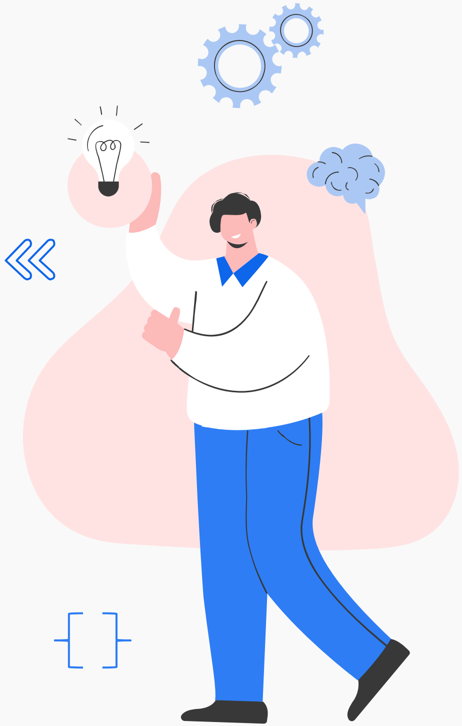
Leave yourself a note in your custom exception.

Exception Groups

Group your exceptions as you want.

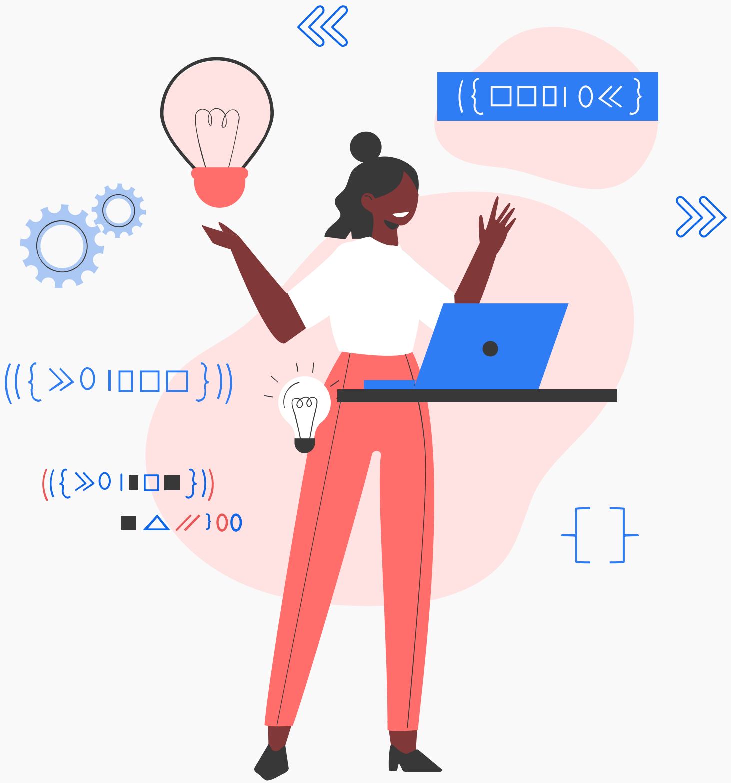
Until now, in a traceback, the only information you got about where an exception got raised was the line.

```
def get_margin(data):
    margin = data['profits']['monthly'] / 10 +
data['profits']['yearly'] / 2
    return margin

data = {
    'profits': {
        'monthly': 0.82,
        'yearly': None,
    },
    'losses': {
        'monthly': 0.23,
        'yearly': 1.38,
    },
}
print(get_margin(data))
```



Exact error locations in tracebacks

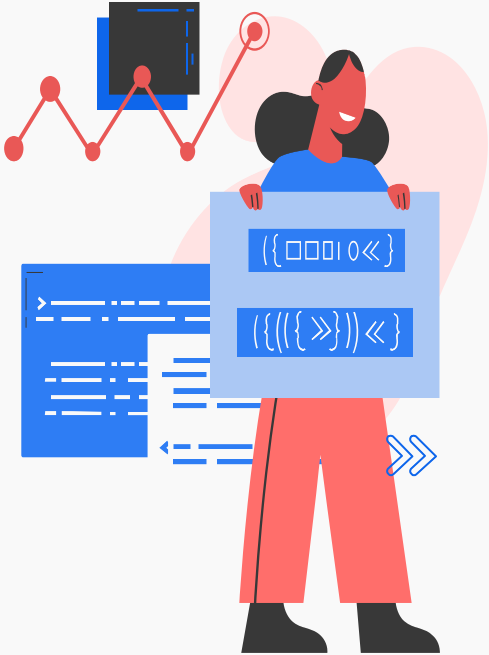
```
Traceback (most recent call last):
  File "asd.py", line 15, in <module>
    print(get_margin(data))
          ^^^^^^^^^^^^^^^^
  File "asd.py", line 2, in print_margin
    margin = data['profits']['monthly'] / 10 + data['profits']['yearly'] / 2
                                               ~~~~~~~~~~~~~~~~~~~~~~~~~~^~~
TypeError: unsupported operand type(s) for /: 'NoneType' and 'int'
```



Exception Notes

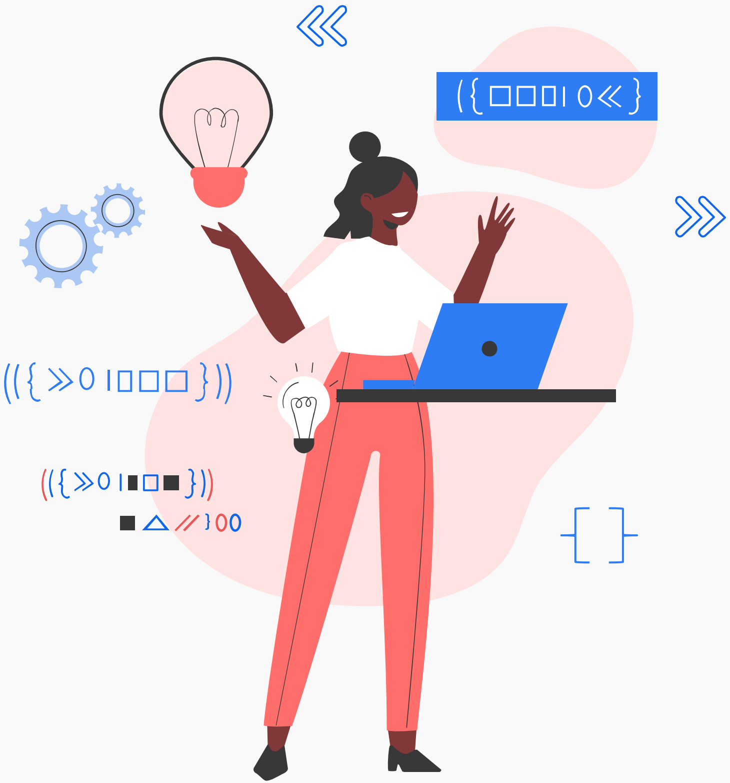
- Python 3.11 introduces exception notes (PEP 678).
- Now, inside your except clauses, you can call the add_note() function and pass a custom message when you raise an error.

```python
import math

try:
  math.sqrt(-1)
except ValueError as e:
  e.add_note("Negative value passed! Please try again.")
  raise
```



Another way to add Exception Notes

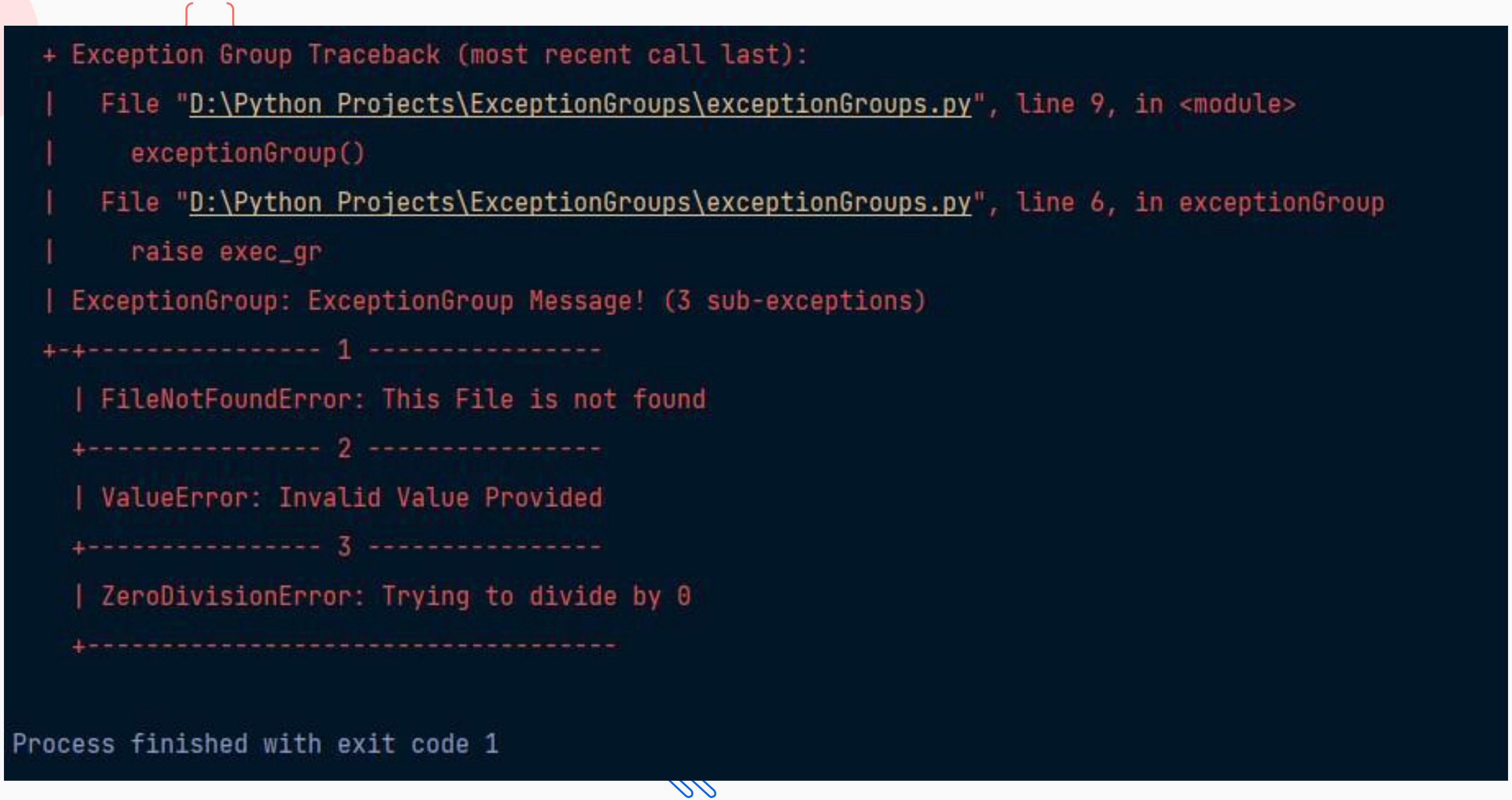
```python
import math

class MyOwnError(Exception):
  __notes__ = ["This is a custom error!"]

try:
  math.sqrt(-1)
except:
  raise MyOwnError
```



Exception Groups

- One way to think about exception groups is that they're regular exceptions wrapping several other regular exceptions.
- However, while exception groups behave like regular exceptions in many respects, they also support special syntax that helps you handle each of the wrapped exceptions effectively.
- In Python 3.11 We Group Exceptions With **ExceptionGroup()**

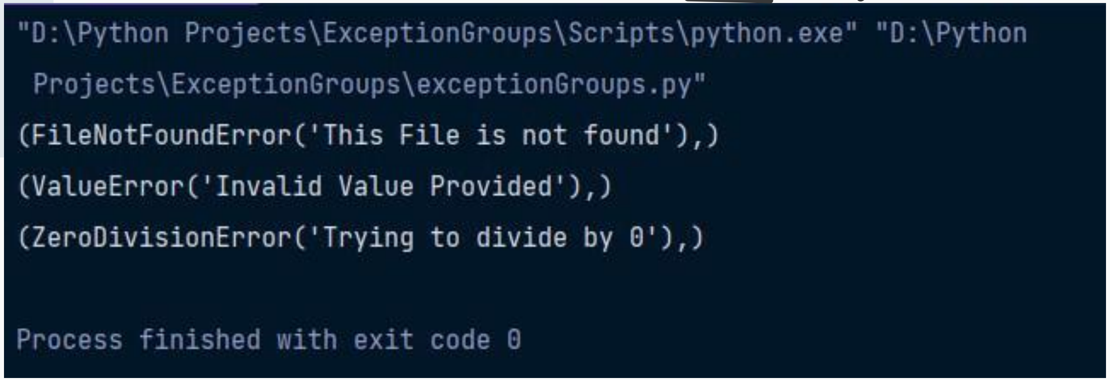
```
def exceptionGroup():
    exec_gr = ExceptionGroup('ExceptionGroup
Message!',
                [FileNotFoundError("This File is not found"),
                ValueError("Invalid Value Provided"),
                ZeroDivisionError("Trying to divide by 0")])
    raise exec_gr
```



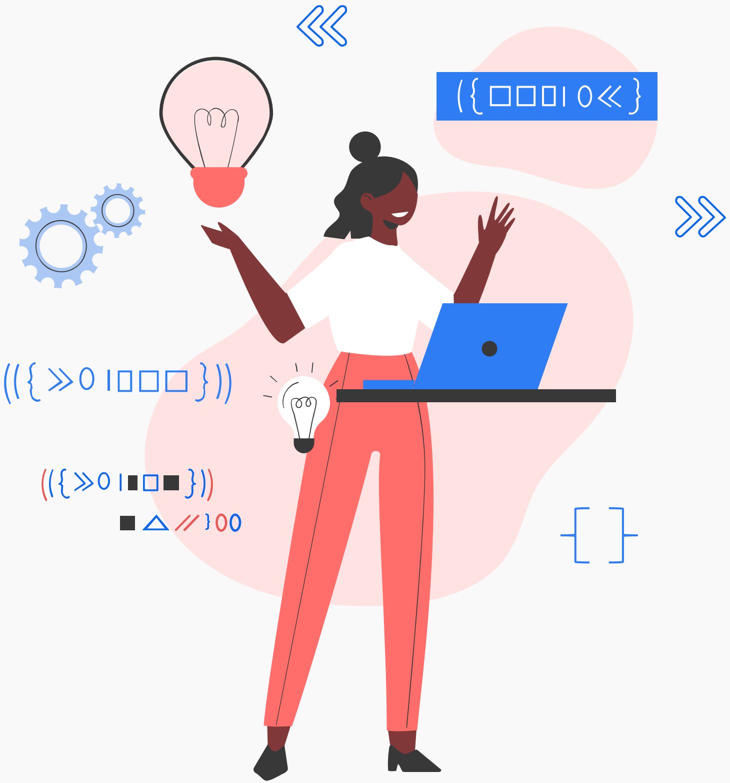
```
  + Exception Group Traceback (most recent call last):
  |   File "D:\Python Projects\ExceptionGroups\exceptionGroups.py", line 9, in <module>
  |     exceptionGroup()
  |   File "D:\Python Projects\ExceptionGroups\exceptionGroups.py", line 6, in exceptionGroup
  |     raise exec_gr
  | ExceptionGroup: ExceptionGroup Message! (3 sub-exceptions)
  +-+---------------- 1 ----------------
    | FileNotFoundError: This File is not found
    +---------------- 2 ----------------
    | ValueError: Invalid Value Provided
    +---------------- 3 ----------------
    | ZeroDivisionError: Trying to divide by 0
    +------------------------------------

Process finished with exit code 1
```

You can even call them one by one:

```
try:
    exceptionGroup()
except* FileNotFoundError as fnf:
    print(fnf.exceptions)
except* ValueError as ve:
    print(ve.exceptions)
except* ZeroDivisionError as zde:
    print(zde.exceptions)
```



```
"D:\Python Projects\ExceptionGroups\Scripts\python.exe" "D:\Python
 Projects\ExceptionGroups\exceptionGroups.py"
(FileNotFoundError('This File is not found'),)
(ValueError('Invalid Value Provided'),)
(ZeroDivisionError('Trying to divide by 0'),)

Process finished with exit code 0
```



Asyncio

Task and Exception Groups

Asyncio Task Groups

```python
import asyncio

async def f1(x: int) -> None:
    await asyncio.sleep(x / 10)
    print(f'hi from {x}')

async def amain() -> int:
    # clunky, previous way of doing this
    futures = [f1(i) for i in range(5)]
    await asyncio.gather(*futures)
    print('done')
    return 0

def main() -> int:
    return asyncio.run(amain())

if __name__ == '__main__':
    raise SystemExit(main())
```

```python
async def f1(x: int) -> None:
    await asyncio.sleep(x / 10)
    print(f'hi from {x}')

async def amain() -> int:
    async with asyncio.TaskGroup() as tg:
        # you can do loops, conditions, etc
        for i in range(5):
            tg.create_task(f1(i))

def main() -> int:
    return asyncio.run(amain())

if __name__ == '__main__':
    raise SystemExit(main())
```

TOML Support

Read TOML Files

What is TOML?

- TOML is short for Tom’s Obvious Minimal Language
- It’s a configuration file format that’s gotten popular over the last decade.
- *The Python community has embraced TOML as the format of choice when specifying metadata for packages and projects.*
- TOML has been designed to be easy for humans to read and easy for computers to parse.
- *While TOML has been used for years by many different tools, Python hasn’t had built-in TOML support. That changes in Python 3.11, when tomllib is added to the standard library. This new module builds on top of the popular tomli third-party library and allows you to parse TOML files.*

```
[second]
label   = { singular = "second", plural = "seconds" }
aliases = ["s", "sec", "seconds"]

[minute]
label      = { singular = "minute", plural = "minutes" }
aliases    = ["min", "minutes"]
multiplier = 60
to_unit    = "second"

[day]
label      = { singular = "day", plural = "days" }
aliases    = ["d", "days"]
multiplier = 24
to_unit    = "hour"

[year]
label      = { singular = "year", plural = "years" }
aliases    = ["y", "yr", "years", "julian_year", "julian years"]
multiplier = 365.25
to_unit    = "day"
```

Tomllib

- The new tomllib library brings support for parsing TOML files. tomllib does not support writing TOML. It’s based on the tomli library.
  When using tomllib.load() you pass in a file object that must be opened in binary mode by specifying mode="rb".

- *The two main functions in tomllib are:*

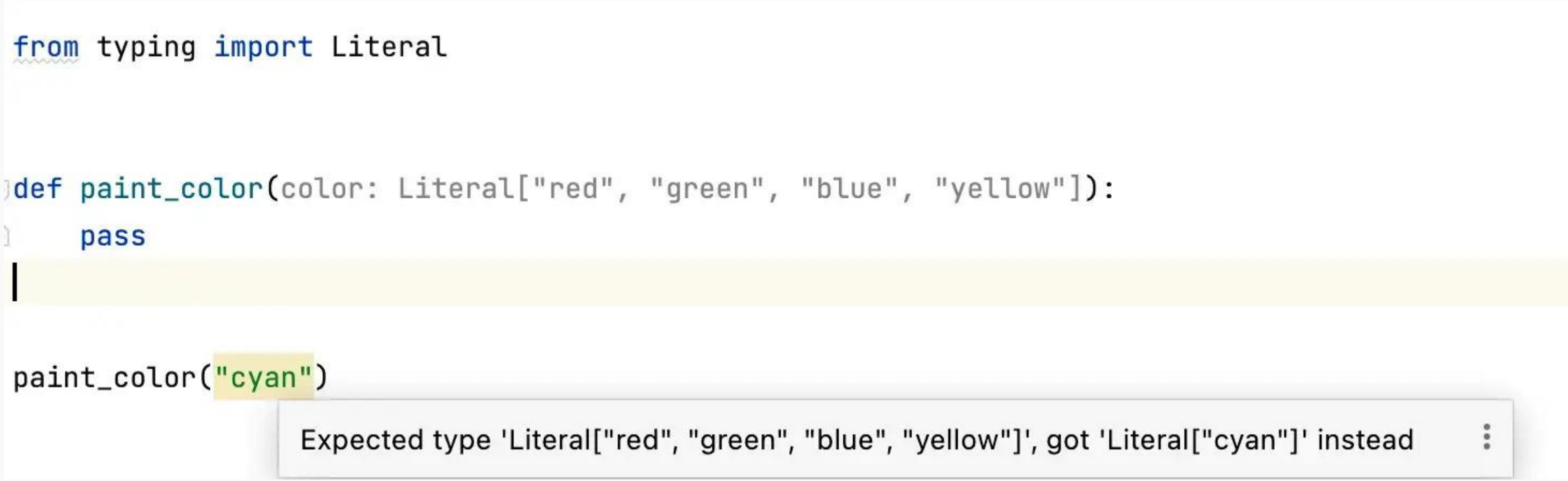
  - load(): load bytes from file
  - *loads(): load from str*

```
import tomllib

# gives TypeError, must use binary mode
with open('t.toml') as f:
    tomllib.load(f)

# correct
with open('t.toml', 'rb') as f:
    tomllib.load(f)
```

Improved Type Variables

Arbitrary literal string type, Data class transforms, Negative Zero Formatting, Improved Type Variables, Variadic generics

Arbitrary Literal String Type



```python
from typing import Literal


def paint_color(color: Literal["red", "green", "blue", "yellow"]):
    pass


paint_color("cyan")
```

Expected type 'Literal["red", "green", "blue", "yellow"]', got 'Literal["cyan"]' instead

To address this limitation, Python 3.11 introduces a new general type LiteralString, which allows the users to enter any string literals, like below:

```
from typing import LiteralString


def paint_color(color: LiteralString):
    pass

paint_color("cyan")
paint_color("blue")
```

The LiteralString type gives you the flexibility of using any string literals instead of specific string literals when you use the Literal type. For more specific use cases where LiteralString is applicable, such as constructing literal SQL query strings, you can refer to the official PEP 675.

Variadic Generics

```
from typing import Generic, TypeVar

Dim1 = TypeVar('Dim1')
Dim2 = TypeVar('Dim2')
Dim3 = TypeVar('Dim3')

class Shape1(Generic[Dim1]):
    pass
class Shape2(Generic[Dim1, Dim2]):
    pass
class Shape3(Generic[Dim1, Dim2, Dim3]):
    pass
```

```
from typing import Generic, TypeVarTuple
Dim = TypeVarTuple('Dim')
class Shape(Generic[*Dim]):
    pass
```

As shown, for three dimensions, we'll have to define three types and their respective classes, which isn't clean and represents a high level of repetition that we should be cautious about. Python 3.11 is introducing the TypeVarTuple that allows you to create generics using multiple types.

Negative Zero Formatting

```
small_num = -0.00321
print(f"{small_num:.2f}")

# -0.00
```

```
small_num = -0.00321
print(f"{small_num:z.2f}")
# 0.00
```

- *Normally, there’s only one zero, and it’s neither positive nor negative.*
- *One weird concept that you may run into when doing calculations with floating-point numbers is negative zero.*

Data Class Transforms

```
from dataclasses import dataclass

@dataclass
class InventoryItem:
    """Class for keeping track of an item in
inventory."""
    name: str
    unit_price: float
    quantity_on_hand: int = 0

    def total_cost(self) -> float:
        return self.unit_price *
self.quantity_on_hand
```

Dataclasses (since Python 3.7) are a metaclass that helps you deal with data oriented classes. Using dataclass decorator, __init__, __hash__, __eq__ and other dunder methods can be generated.

```
@dataclasses.dataclass(*, init=True,
repr=True, eq=True, order=False,
unsafe_hash=False, frozen=False,
match_args=True, kw_only=False,
slots=False, weakref_slot=False
```

Data Class Transforms

```python
from dataclasses import dataclass

@dataclass
class InventoryItem:
    """Class for keeping track of an item in inventory."""
    name: str
    unit_price: float
    quantity_on_hand: int = 0

    def total_cost(self) -> float:
        return self.unit_price * self.quantity_on_hand
```

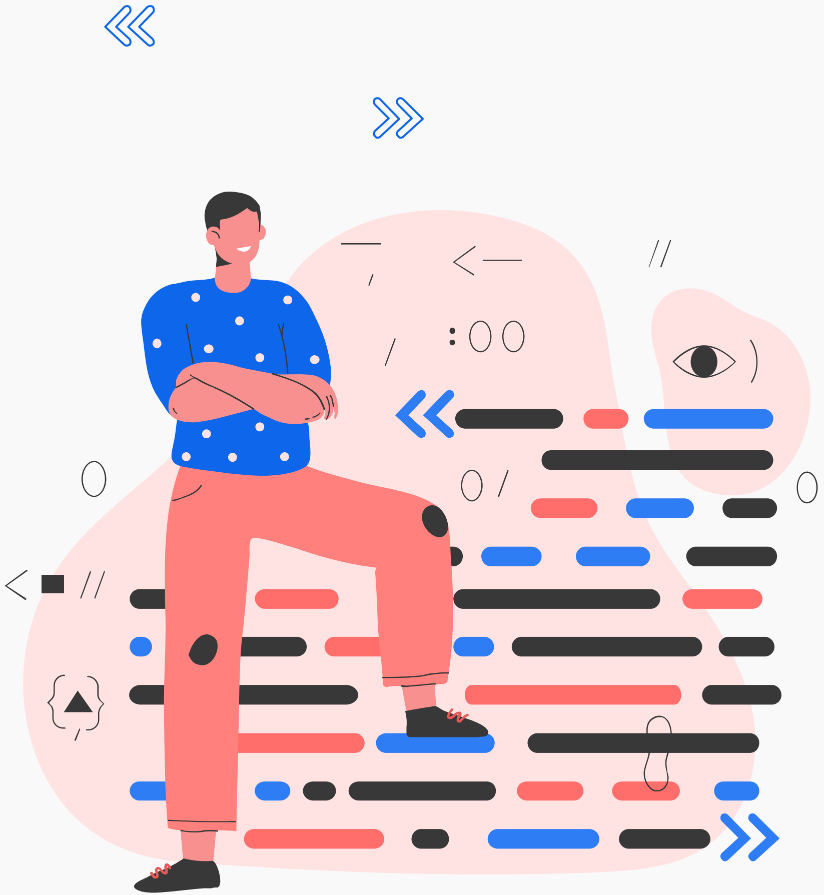
Dataclasses (since Python 3.7) are a metaclass that helps you deal with data oriented classes. Using dataclass decorator, __init__, __hash__, __eq__ and other dunder methods can be generated.

The second code on the next slide will add to the class, among other things, an __init__()

Data Class Transforms

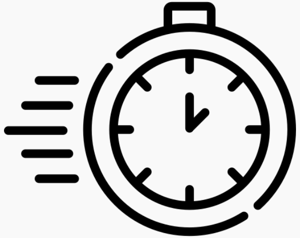
```
from dataclasses import dataclass

@dataclass
class InventoryItem:
    """Class for keeping track of an item in
inventory."""
    name: str
    unit_price: float
    quantity_on_hand: int = 0

    def total_cost(self) -> float:
        return self.unit_price *
self.quantity_on_hand
```

```
from typing import dataclass_transform, Type, TypeVar

T = TypeVar("T")

@dataclass_transform(kw_only_default=True)
def as_model(cls: Type[T]) -> Type[T]:
    print(f"__dataclass_transform__:
{as_model.__dataclass_transform__}")

    def init_instance_with_kwargs(instance: T, **kwargs):
        for item, _type in instance.__annotations__.items():
            argument = kwargs.get(item)
            assert isinstance(argument, _type)
            setattr(instance, item, argument)

    cls.__init__ = init_instance_with_kwargs
    return cls

@as_model
class BookModel:
    id: int
    title: str
    author: str
```



Speed Improvements

Really Better Performance Improvement?!

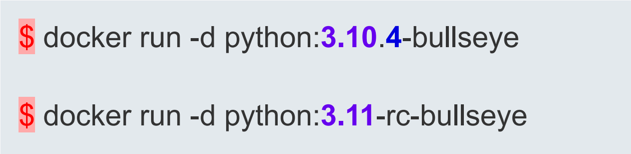


Tomllib

- The first significant change that will excite data scientists is speed improvement—the standard benchmark suite runs about 25% faster compared to 3.10.

- *The Python docs claim 3.11 can be up to 60% faster in some instances.*

- To compare the speeds of Python 3.10 and 3.11, you will need a Docker installation.



```
$ docker run -d python:3.10.4-bullseye

$ docker run -d python:3.11-rc-bullseye
```

Types of brackets

Function	py310(ms)	py311(ms)
scimark_monte_carlo	0.12392	0.08052
mako	0.01967	0.01289
chameleon	0.01131	0.00843
float	0.14265	0.09133
regex_effbot	0.00315	0.00261

Python 3.12?

- *Even better error messages*
- *Missing module suggestions ("did you forget to import X?")*
- *Support for the Linux perf profiler*
- *Buffer protocol dunders*
- *Running a profiler or attaching a debugger to a Python program gives you visibility and insight into what the program's doing.*
- *In Python 3.12, you can use a TypedDict as source of types to hint keyword arguments used in a function.*

Python 3.12?

- *Better Garbage Collection*
- *Better Parallelism*
- *Immortal Objects - Every object in Python has a reference count that tracks how many times other objects refer to it, including built-in objects like None.*
- *Comprehensions, a syntax that lets you quickly construct lists, dictionaries, and sets, are now constructed "inline" rather than by way of temporary objects.*
- *Still noteworthy performance improvement.*

Thanks!