WHAT THE STRUCT?!

> whoami

> DIRECTOR, AI ENGINEERING @ S&P GLOBAL

> PYTHON FANATIC

> NEW DAD

Main page
Contents
Current events
Random article
About Wikipedia
Contact us
Donate

Contribute
Help
Learn to edit
Community portal
Recent changes
Upload file

Tools
What links here
Related changes
Special pages
Permanent link
Page information
Cite this page
Wikidata item

Print/export
Download as PDF
Printable version

In other projects
Wikiversity

Languages
Čeština

struct (C programming language)

From Wikipedia, the free encyclopedia

A **struct** in the C programming language (and many derivatives) is a composite data type (or record) declaration that defines a physically grouped list of variables under one name in a block of memory, allowing the different variables to be accessed via a single pointer or by the struct declared name which returns the same address. The struct data type can contain other data types so is used for mixed-data-type records such as a hard-drive directory entry (file length, name, extension, physical address, etc.), or other mixed-type records (name, address, telephone, balance, etc.).

The C struct directly references a *contiguous block* of physical memory, usually delimited (sized) by word-length boundaries. It corresponds to the similarly named feature available in some assemblers for Intel processors. Being a block of contiguous memory, each field within a struct is located at a certain fixed offset from the start.

Because the contents of a struct are stored in contiguous memory, the sizeof operator must be used to get the number of bytes needed to store a particular type of struct, just as it can be used for primitives. The alignment of particular fields in the struct (with respect to word boundaries) is implementation-specific and may include padding, although modern compilers typically support the `#pragma pack` directive, which changes the size in bytes used for alignment.[1]

In the C++ language, a struct is identical to a C++ class but has a different default visibility: class members are private by default, whereas struct members are public by default.

struct

Contents [hide]

In other languages [edit]

The struct data type in C was derived from the ALGOL 68 struct data type.[2]

Like its C counterpart, the struct data type in C# (*Structure* in Visual Basic .NET) is similar to a class. The biggest difference between a struct and a class in these languages is that when a struct is passed as an argument to a function, any modifications to the struct in that function will not be reflected in the original variable (unless pass-by-reference is used).[3]

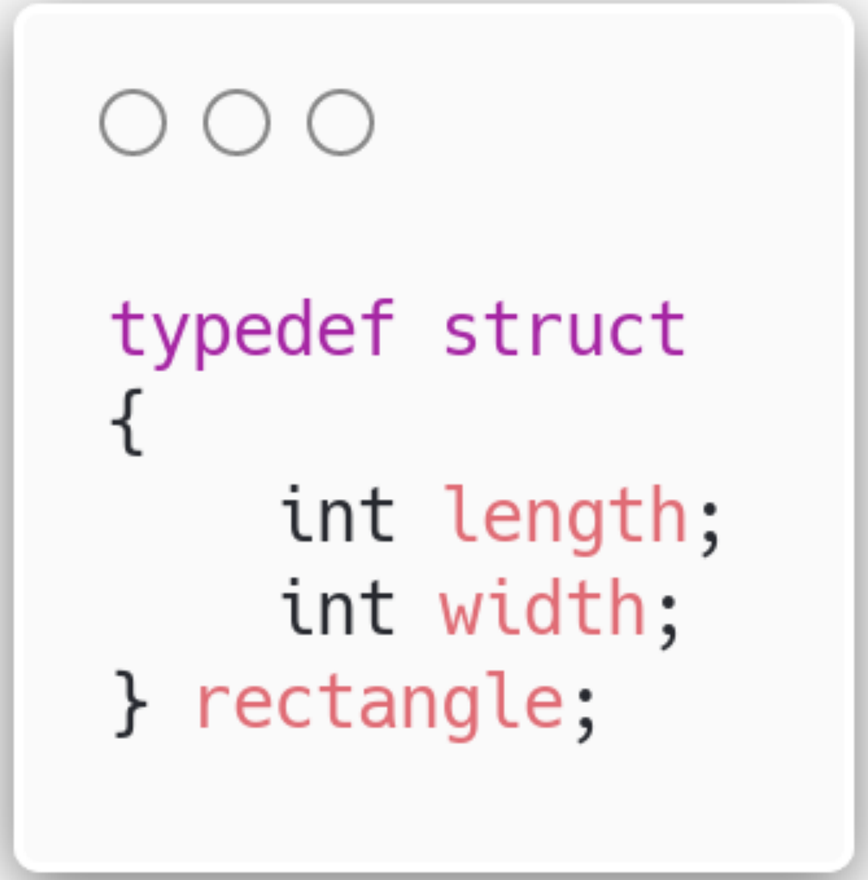
This differs from C++, where classes or structs can be statically allocated or dynamically allocated either on the stack (similar to C#) or on the heap, with an explicit pointer. In C++, the only difference between a struct and a class is that the members and base classes of a struct are public by default. (A class defined with the `class` keyword has private members and base classes by default.)

C

- MINIMAL PROGRAMMING LANGUAGE. CAME AFTER B
- PROCEDURAL. NO OBJECTS!

```c
#include <stdio.h>

int main(){
    int rect_length = 5;
    int rect_width = 4;
    int area = rect_length * rect_width;
    printf("Area is %d!\n", area);
}

// Area is 20!
```

```c
#include <stdio.h>

int main(){
    int rect_length_one = 5;
    int rect_width_one = 4;
    int rect_length_two = 6;
    int rect_width_two = 7;
    int area_one = rect_length_one * rect_width_one;
    int area_two = rect_length_two * rect_width_two;
    int area_diff = area_two - area_one;
    printf("Area difference is %d!\n", area_diff);
}

// Area difference is 22!
```



```
typedef struct
{
    int length;
    int width;
} rectangle;
```

```c
#include <stdio.h>

typedef struct
{
    int length;
    int width;
} rectangle;

int main(){
    rectangle firstRectangle = {4, 5};
    rectangle secondRectangle = {6, 7};
    int area_one = firstRectangle.length * firstRectangle.width;
    int area_two = secondRectangle.length * secondRectangle.width;
    int area_diff = area_two - area_one;
    printf("Area difference is %d!\n", area_diff);
}
```

```c
#include <stdio.h>

typedef struct
{
    int length;
    int width;
} rectangle;


int area(rectangle rect){
    return rect.length * rect.width;
}


int main(){
    rectangle firstRectangle = {4, 5};
    rectangle secondRectangle = {6, 7};
    int area_diff = area(secondRectangle) - area(firstRectangle);
    printf("Area difference is %d!\n", area_diff);
}

// Area difference is 22!
```

```c
#include <stdio.h>

typedef struct
{
    int length;
    int width;
} rectangle;

typedef struct
{
    rectangle first;
    rectangle second;
} rectangles;

int area(rectangle rect){
    return rect.length * rect.width;
}

int area_diff(rectangles rects){
    return area(rects.second) - area(rects.first);
}


int main(){
    rectangle firstRectangle = {4, 5};
    rectangle secondRectangle = {6, 7};
    rectangles twoRectangles = {firstRectangle, secondRectangle};
    printf("Area difference is %d!\n", area_diff(twoRectangles));
}

// Area difference is 22!
```

STRUCTURE++

READABILITY ++

REUSABILITY ++

ASIDE: STRUCTS AND MEMORY

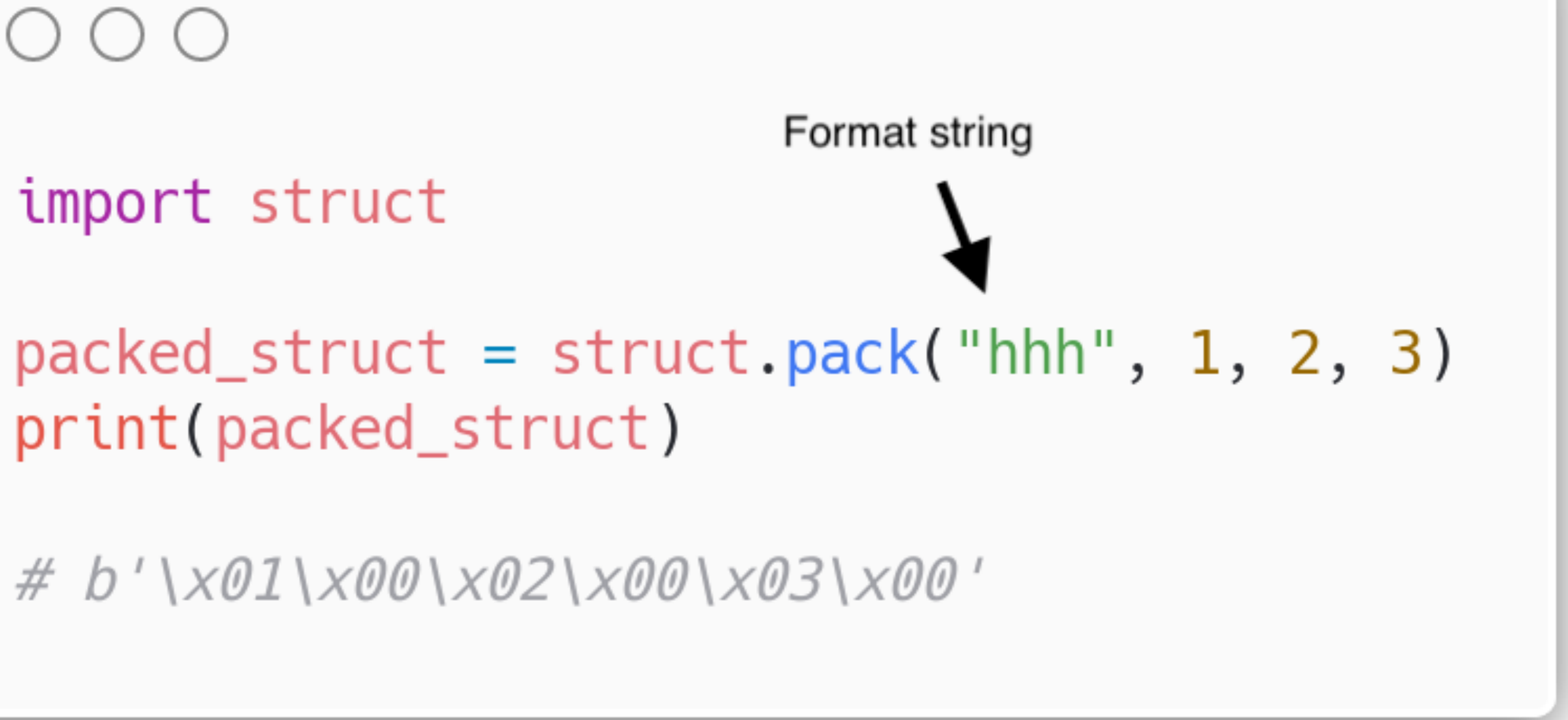
- C VARIABLES ARE STORED IN SPECIFIC MEMORY ADDRESSES
- USING A STRUCT, YOU CAN ENSURE THAT ADDRESSES FOR THE ELEMENTS IN THE STRUCT ARE AT CONTIGUOUS LOCATIONS IN MEMORY
- YOU CAN THEN READ AND WRITE THE STRUCT IN BINARY AS A SINGLE UNIT

PYTHON: STRUCT

`struct`

- PYTHON MODULE FOR UNPACKING AND REPACKING STRUCTS STORED AS BINARY DATA
- ONLY USEFUL FOR HIGH-PERFORMANCE CODE AND INTEROPERATING WITH C
- REQUIRES DECLARING FORMAT TYPES

```python
import struct

packed_struct = struct.pack("hhh", 1, 2, 3)
print(packed_struct)

# b'\x01\x00\x02\x00\x03\x00'
```


Format string
import struct
packed_struct = struct.pack("hhh", 1, 2, 3)
print(packed_struct)
# b'\x01\x00\x02\x00\x03\x00'

Format Characters

Format characters have the following meaning; the conversion between C and Python values should be obvious given their types. The 'Standard size' column refers to the size of the packed value in bytes when using standard size; that is, when the format string starts with one of `'<'`, `'>'`, `'!'` or `'='`. When using native size, the size of the packed value is platform-dependent.

Format	C Type	Python type	Standard size	Notes
`x`	pad byte	no value		
`c`	`char`	bytes of length 1	1	
`b`	`signed char`	integer	1	(1), (2)
`B`	`unsigned char`	integer	1	(2)
`?`	`_Bool`	bool	1	(1)
`h`	`short`	integer	2	(2)
`H`	`unsigned short`	integer	2	(2)
`i`	`int`	integer	4	(2)
`I`	`unsigned int`	integer	4	(2)
`l`	`long`	integer	4	(2)
`L`	`unsigned long`	integer	4	(2)
`q`	`long long`	integer	8	(2)
`Q`	`unsigned long long`	integer	8	(2)
`n`	`ssize_t`	integer		(3)
`N`	`size_t`	integer		(3)
`e`	(6)	float	2	(4)
`f`	`float`	float	4	(4)
`d`	`double`	float	8	(4)
`s`	`char[]`	bytes		
`p`	`char[]`	bytes		
`P`	`void *`	integer		(5)

Format Characters

Format characters have the following meaning; the conversion between C and Python values should be obvious given their types. The 'Standard size' column refers to the size of the packed value in bytes when using standard size; that is, when the format string starts with one of `'<'`, `'>'`, `'!'` or `'='`. When using native size, the size of the packed value is platform-dependent.

Format	C Type	Python type	Standard size	Notes
`x`	pad byte	no value		
`c`	`char`	bytes of length 1	1	
`b`	`signed char`	integer	1	(1), (2)
`B`	`unsigned char`	integer	1	(2)
`?`	`_Bool`	bool	1	(1)
`h`	`short`	integer	2	(2)
`H`	`unsigned short`	integer	2	(2)
`i`	`int`	integer	4	(2)
`I`	`unsigned int`	integer	4	(2)
`l`	`long`	integer	4	(2)
`L`	`unsigned long`	integer	4	(2)
`q`	`long long`	integer	8	(2)
`Q`	`unsigned long long`	integer	8	(2)
`n`	`ssize_t`	integer		(3)
`N`	`size_t`	integer		(3)
`e`	(6)	float	2	(4)
`f`	`float`	float	4	(4)
`d`	`double`	float	8	(4)
`s`	`char[]`	bytes		
`p`	`char[]`	bytes		
`P`	`void *`	integer		(5)

```python
import struct

packed_struct = struct.pack("hhh", 1, 2, 3)
print(packed_struct)

# b'\x01\x00\x02\x00\x03\x00'
```


import struct
packed_struct = struct.pack("lll", 1, 2, 3)
print(packed_struct)

```python
import struct

packed_struct = struct.pack("lll", 1, 2, 3)
print(packed_struct)

#
b'\x01\x00\x00\x00\x00\x00\x00\x00\x02\x00\x00\x00\x00\x00\x00\x00\x03\x00\x00\x00\x00\x00\x00\x00'
```

```python
import struct

packed_struct = struct.pack("cccc", b"z", b"a", b"c", b"h")
print(packed_struct)

# b'zach'
```

```python
import struct

packed_struct = struct.pack("cccc", b"z", b"a", b"c", b"h")
unpacked_struct = struct.unpack("cccc", packed_struct)
print(unpacked_struct)

# (b'z', b'a', b'c', b'h')
```

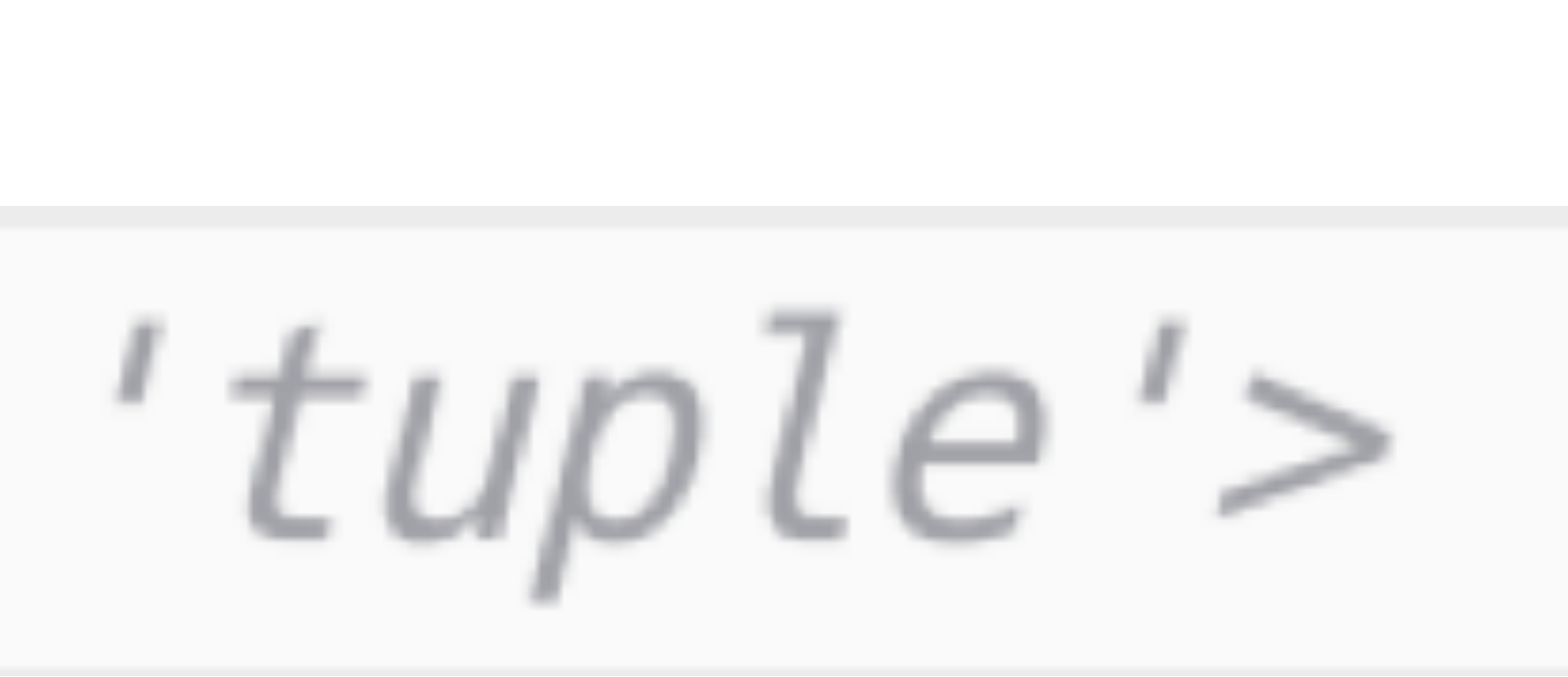
```python
import struct

packed_struct = struct.pack("cccc", b"z", b"a", b"c", b"h")
unpacked_struct = struct.unpack("cccc", packed_struct)
print(unpacked_struct)

# (b'z', b'a', b'c', b'h')
```

```python
import struct

packed_struct = struct.pack("cccc", b"z", b"a", b"c", b"h")
unpacked_struct = struct.unpack("cccc", packed_struct)

print(type(unpacked_struct))

# <class 'tuple'>
```

```python
import struct

packed_struct = struct.pack("cccc", b"z",
unpacked_struct = struct.unpack("cccc", pa

print(type(unpacked_struct))

# <class 'tuple'>
```

```
packed_struct = struct.pack(
unpacked_struct = struct.unpack(

print(type(unpacked_struct))

# <class 'tuple'>
```

```
# <class 'tuple'>
```


'tuple'>

WHAT THE STRUCT

```c
#include <stdio.h>

typedef struct
{
    int length;
    int width;
} rectangle;

int main(){
    rectangle firstRectangle = {4, 5};
    rectangle secondRectangle = {6, 7};
    int area_one = firstRectangle.length * firstRectangle.width;
    int area_two = secondRectangle.length * secondRectangle.width;
    int area_diff = area_two - area_one;
    printf("Area difference is %d!\n", area_diff);
}
```

```python
import struct

packed_struct = struct.pack("cccc", b"z", b"a", b"c", b"h")
unpacked_struct = struct.unpack("cccc", packed_struct)

print(unpacked_struct[0])

# b'z'
```

NAMEDTUPLE

`collections.namedtuple`

IT'S A TUPLE. WITH NAMES!

```python
import collections
import struct

packed_struct = struct.pack("cccc", b"z", b"a", b"c", b"h")
unpacked_struct = struct.unpack("cccc", packed_struct)

FourLetters = collections.namedtuple("FourLetters", "first second third
fourth")
my_name = FourLetters(*unpacked_struct)
print(my_name)

# FourLetters(first=b'z', second=b'a', third=b'c', fourth=b'h')
```

```python
import collections
import struct

packed_struct = struct.pack("cccc", b"z", b"a", b"c", b"h")
unpacked_struct = struct.unpack("cccc", packed_struct)

FourLetters = collections.namedtuple("FourLetters", "first second third
fourth")
my_name = FourLetters(*unpacked_struct)

print(my_name.first, my_name.second, my_name.third)

# b'z' b'a' b'c'
```

`namedtuple(typename, field_names)`

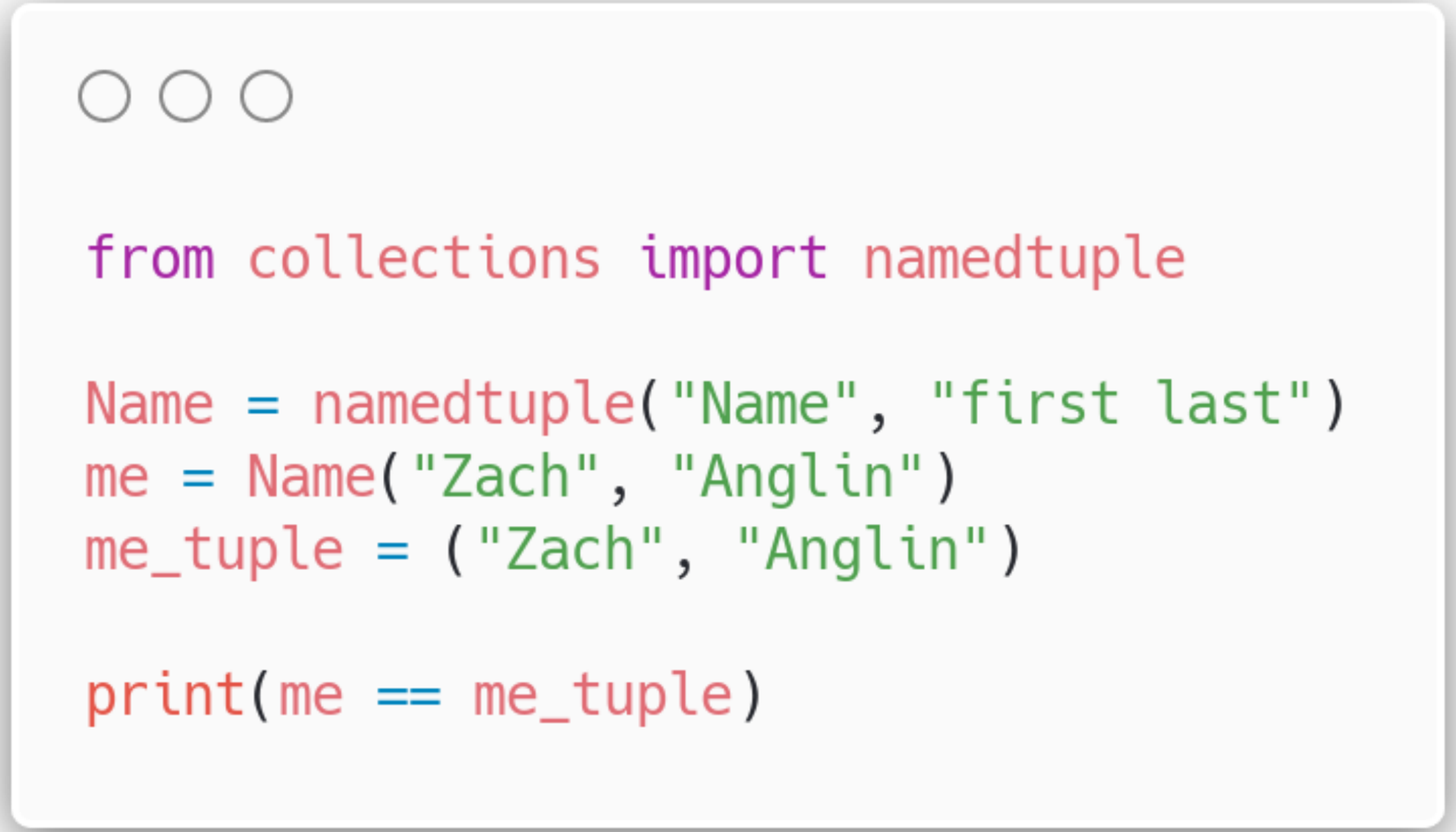
- CREATES A SUBTYPE OF `tuple` NAMED `typename`
- INHERITS ALL TUPLE METHODS
- GIVES NAMED ATTRIBUTE ACCESS TO FIELDS IN TUPLE

STILL A TUPLE

...STILL IMMUTABLE
...SAME MEMORY FOOTPRINT (NO `dict`)
...COMPARES EQUALLY TO `tuple`

```python
from collections import namedtuple

Name = namedtuple("Name", "first last")
me = Name("Zach", "Anglin")
print(me)

# Name(first='Zach', last='Anglin')
```

```python
from collections import namedtuple

Name = namedtuple("Name", "first last")
me = Name("Zach", "Anglin")
me_tuple = ("Zach", "Anglin")

print(me_tuple)
# ('Zach', 'Anglin')
```



```python
from collections import namedtuple

Name = namedtuple("Name", "first last")
me = Name("Zach", "Anglin")
me_tuple = ("Zach", "Anglin")

print(me == me_tuple)
```

```python
from collections import namedtuple

Name = namedtuple("Name", "first last")
me = Name("Zach", "Anglin")
me_tuple = ("Zach", "Anglin")

print(me == me_tuple)
# True
```

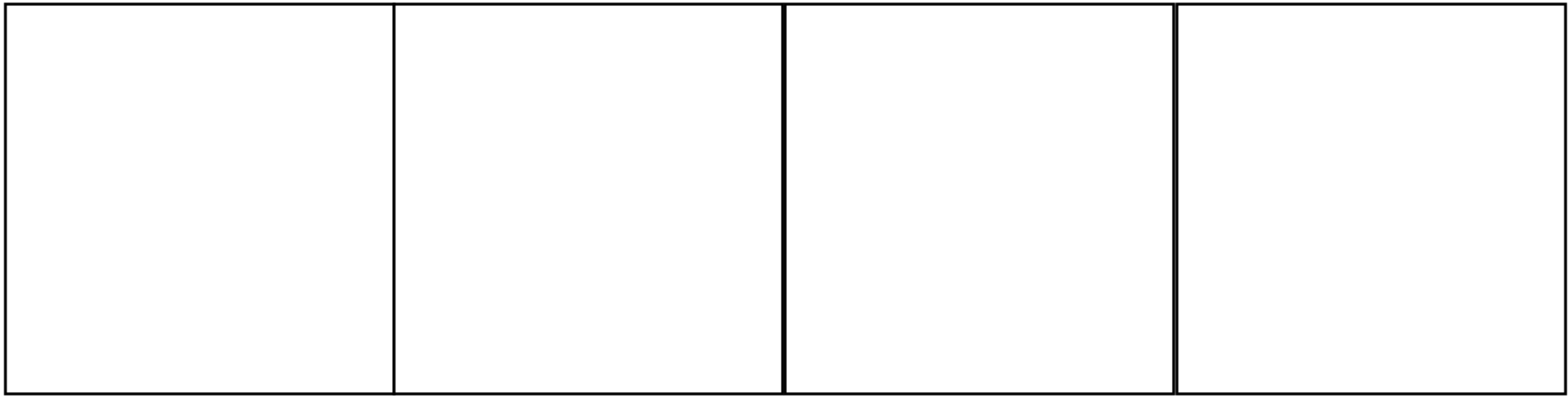
DEFAULT VALUES

```
from collections import namedtuple

Name = namedtuple("Name", "first last middle", defaults=["X"])
me = Name("Zach", "Angelin")

print(me)
# Name(first='Zach', last='Angelin', middle='X')
```

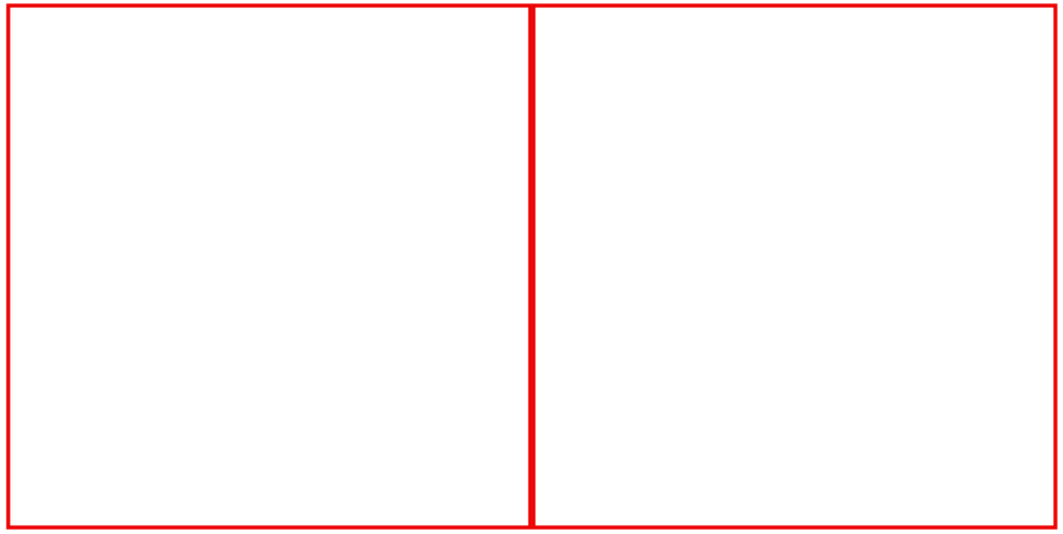
```python
from collections import namedtuple

Name = namedtuple("Name", "first last middle", defaults=["X"])
me = Name("Zach", "Anglin")

print(me)
# Name(first='Zach', last='Anglin', middle='X')
```

```
Student = namedtuple("Student", "name id age year", defaults=[10, 5])
```

Fields

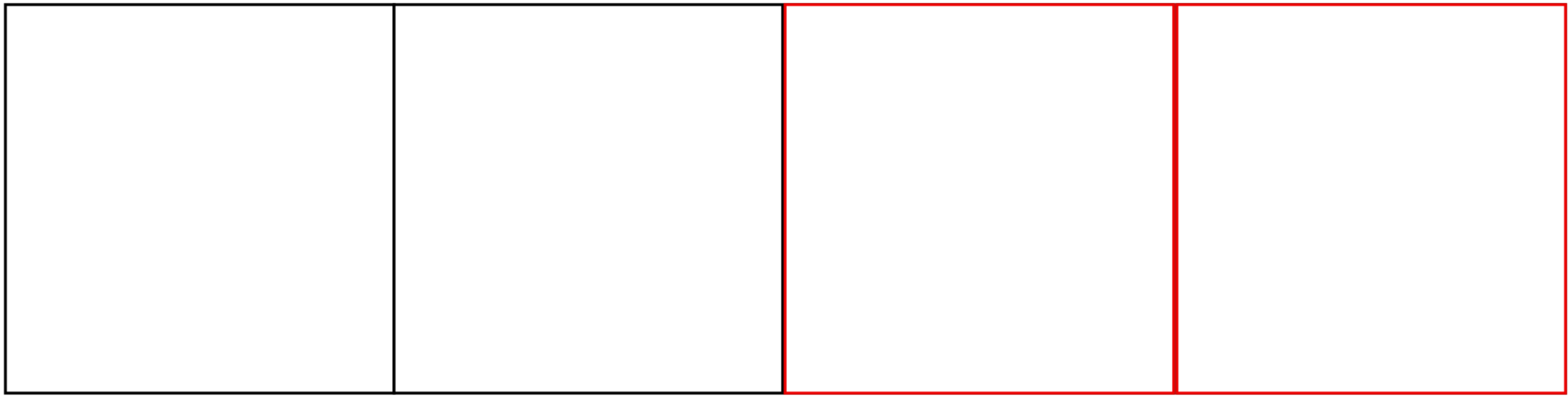


Defaults



Defaults

Fields



```python
from collections import namedtuple

Student = namedtuple("Student", "name id age year", defaults=[10, 5])
tommy = Student("tommy", 12345)

print(tommy)
# Student(name='tommy', id=12345, age=10, year=5)
```

DICT CONVERSION

```
print(tommy._asdict())
# OrderedDict([('name', 'tommy'), ('id', 12345), ('age', 10), ('year', 5)])
```

```python
from collections import namedtuple

Student = namedtuple("Student", "name id age year", defaults=[10, 5])
tommy = Student("tommy", 12345)

print(tommy._asdict())
# OrderedDict([('name', 'tommy'), ('id', 12345), ('age', 10), ('year', 5)])
```

IMMUTABILITY

```
from collections import namedtuple

Student = namedtuple("Student", "name id age year", defaults=[10, 5])
tommy = Student("tommy", 1245)
print(tommy)
tommy.age = 11
print(tommy)
```

```python
from collections import namedtuple

Student = namedtuple("Student", "name id age year", defaults=[10, 5])
tommy = Student("tommy", 12345)
print(tommy)
# Student(name='tommy', id=12345, age=10, year=5)

tommy.age = 11
print(tommy)
```

```python
from collections import namedtuple

Student = namedtuple("Student", "name id age year", defaults=[10, 5])
tommy = Student("tommy", 12345)
print(tommy)
# Student(name='tommy', id=12345, age=10, year=5)

tommy.age = 11
print(tommy)
```

AttributeError

TUPLES ARE IMMUTABLE

namedtuple._replace()

```python
from collections import namedtuple

Student = namedtuple("Student", "name id age year", defaults=[10, 5])
tommy = Student("tommy", 12345)
print(tommy)
# Student(name='tommy', id=12345, age=10, year=5)

tommy_plus_one = tommy._replace(age=11)
print(tommy_plus_one)
# Student(name='tommy', id=12345, age=11, year=5)
```

```python
from collections import namedtuple

Student = namedtuple("Student", "name id age year", defaults=[10, 5])
tommy = Student("tommy", 12345)
print(tommy)
# Student(name='tommy', id=12345, age=10, year=5)

tommy_plus_one = tommy._replace(age=11)
print(tommy_plus_one)
# Student(name='tommy', id=12345, age=11, year=5)
```

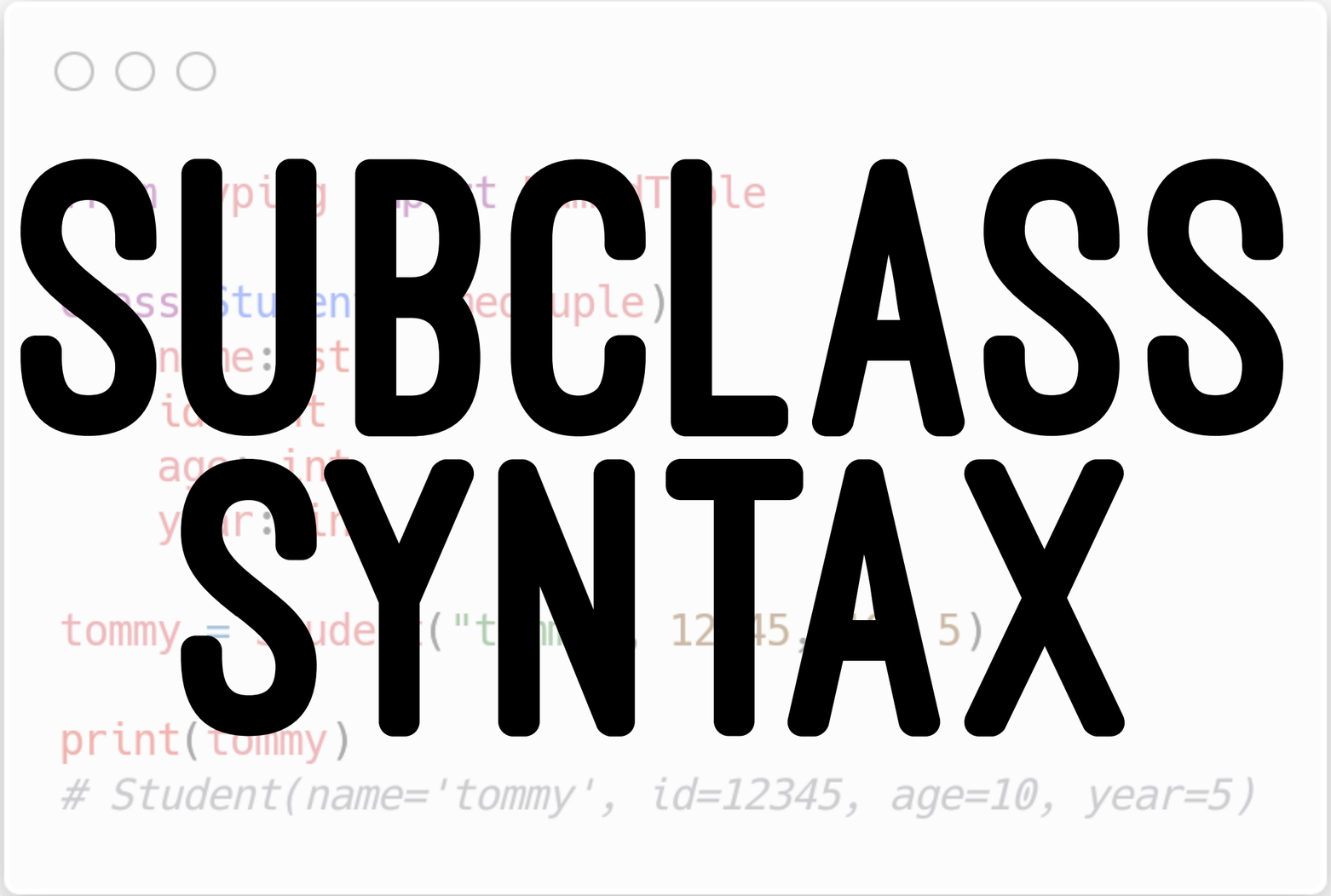
WHAT ABOUT TYPES?

typing.NamedTuple

`typing.NamedTuple`

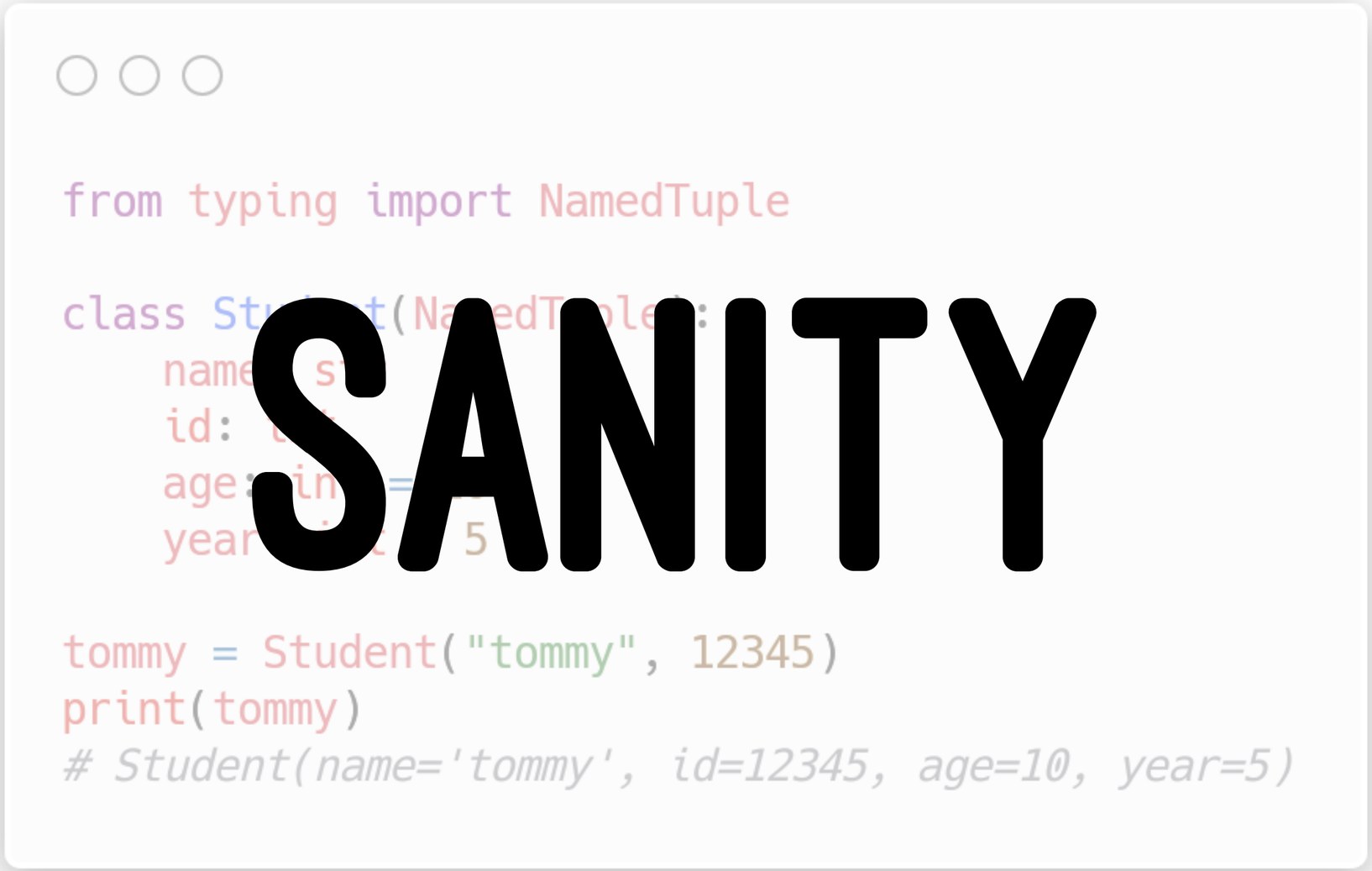
- INTRODUCED IN PYTHON 3.6
- PROVIDES STRUCTURAL SUBCLASS OF `tuple` WITH TYPED FIELDS
- INTRODUCES CLASS-BASED DECLARATION SYNTAX

```python
from typing import NamedTuple

Student = NamedTuple("Student", [("name", str),
                                 ("id", int),
                                 ("age", int),
                                 ("year", int)])
tommy = Student("tommy", 12345, 10, 5)

print(tommy)
# Student(name='tommy', id=12345, age=10, year=5)
```


SUBCLASS
SYNTAX

```python
from typing import NamedTuple

class Student(NamedTuple):
    name: str
    id: int
    age: int
    year: int

tommy = Student("tommy", 12345, 10, 5)

print(tommy)
# Student(name='tommy', id=12345, age=10, year=5)
```

```python
from typing import NamedTuple

class Student(NamedTuple):
    name: str
    id: int
    age: int = 10
    year: int = 5

tommy = Student("tommy", 12345)
print(tommy)
# Student(name='tommy', id=12345, age=10, year=5)
```

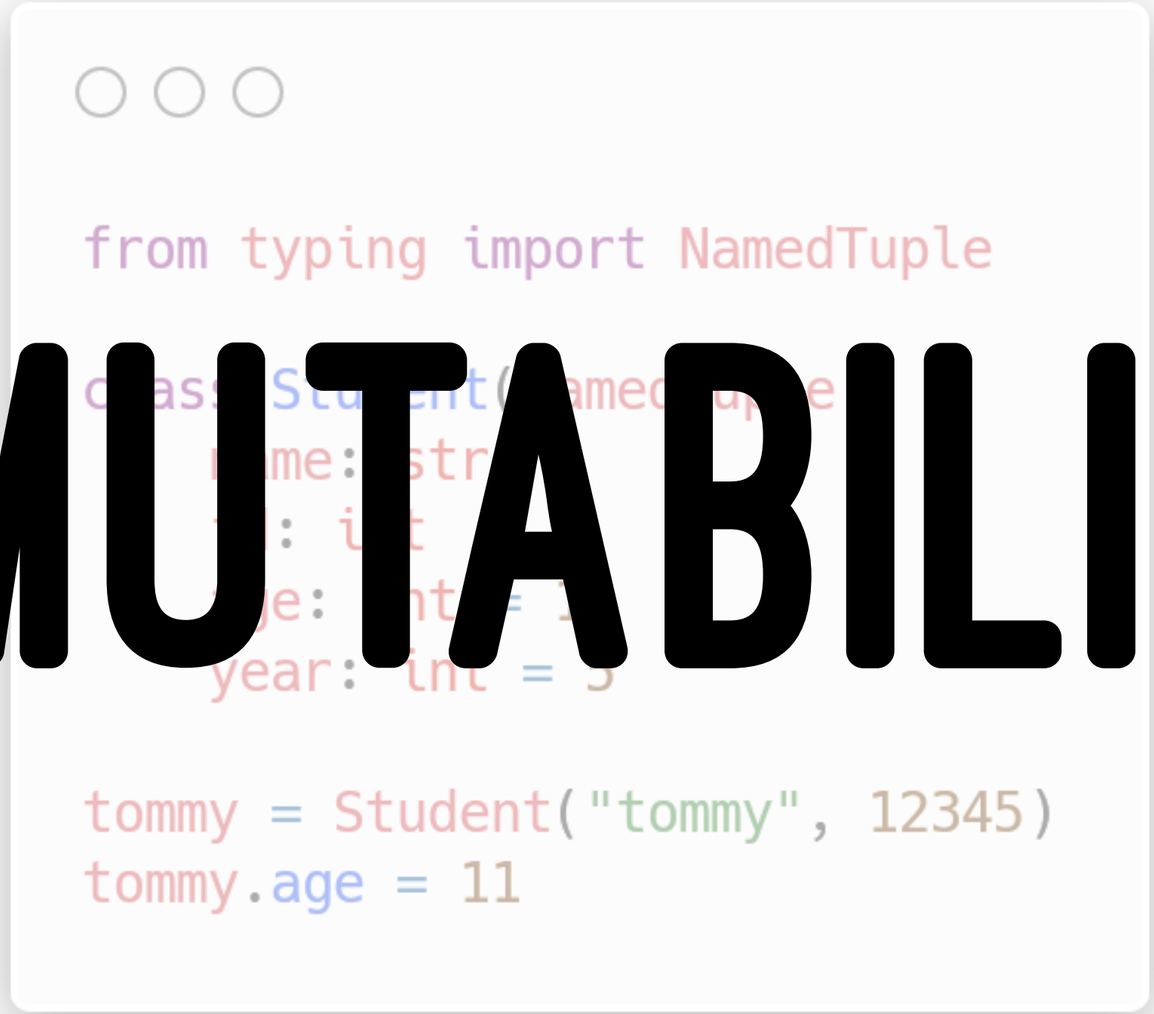

from typing import NamedTuple
class Student(NamedTuple):
name: str
id: int
age: int = 10
year: int = 5
tommy = Student("tommy", 12345)
print(tommy)
# Student(name='tommy', id=12345, age=10, year=5)
SANITY

COMPARISON WITH TUPLE

```python
from typing import NamedTuple

class Student(NamedTuple):
    name: str
    id: int
    age: int = 10
    year: int = 5

tommy = Student("tommy", 12345)

print(tommy == ("tommy", 12345, 10, 5))
# True
```

```
class Car(NamedTuple):
    color: str
    vin: int
    wheels: int
    tons: int

ginger = Student("ginger", id=321, age=4, year=1)
pickup = Car("ginger", vin=321, wheels=4, tons=1)

print(ginger == pickup)
# True
```

IMMUTABILITY



```python
from typing import NamedTuple

class Student(NamedTuple):
    name: str
    id: int
    age: int = 10
    year: int = 5

tommy = Student("tommy", 12345)
tommy.age = 11
```

AttributeError: can't set attribute

```
from typing import NamedTuple

class Student(NamedTuple):
    name: str
    id: int
    age: int = 10
    year: int = 5

tommy = Student("tommy", 12345)
tommy.age = 11
```

WHEN TO USE NAMED TUPLES?

- TO ADD STRUCTURE TO PROCESSED IMMUTABLE RECORDS
- TO GIVE DESCRIPTIVE NAMES IN COMPOUND DATA TYPES
- TO GET RID OF "MAGIC NUMBERS" (INDICES)

WHEN TO USE `typing.NamedTuple` OVER `collections.namedtuple`?

- BASICALLY ALL THE TIME
- OR, AT LEAST WHEN YOU'RE WORKING IN A TYPED CODEBASE

WHEN TO USE `typing.NamedTuple` OVER `collections.namedtuple`?

- BASICALLY ALL THE TIME
- OR, AT LEAST WHEN YOU'RE WORKING IN A TYPED CODEBASE
- (ALSO PLEASE WORK IN TYPED CODEBASES)

EXITING TUPLE-LAND

dataclasses

`dataclasses`

- INTRODUCED IN PYTHON 3.7
- SHORTHAND FOR CREATING LIGHTWEIGHT CLASSES
- CREATE INDEPENDENT TYPES. NO RELATION TO TUPLES

```python
from dataclasses import dataclass

@dataclass
class Student:
    name: str
    id: int
    age: int = 10
    year: int = 5

tommy = Student("tommy", 12345)

print(tommy)
# Student(name='tommy', id=12345, age=10, year=5)
```

```python
from dataclasses import dataclass

@dataclass
class Student:
    name: str
    id: int
    age: int = 10
    year: int = 5

tommy = Student("tommy", 12345)
tommy.age += 1

print(tommy)
# Student(name='tommy', id=12345, age=11, year=5)
```

```python
from dataclasses import dataclass

@dataclass
class Student:
    name: str
    id: int
    age: int = 10
    year: int = 5

tommy = Student("tommy", 12345)
tommy.age += 1

print(tommy)
# Student(name='tommy', id=12345, age=11, year=5)
```

NO AttributeError!

```python
from dataclasses import dataclass

@dataclass
class Student:
    name: str
    id: int
    age: int = 10
    year: int = 5

tommy = Student("tommy", 12345)

print(tommy[2])
```

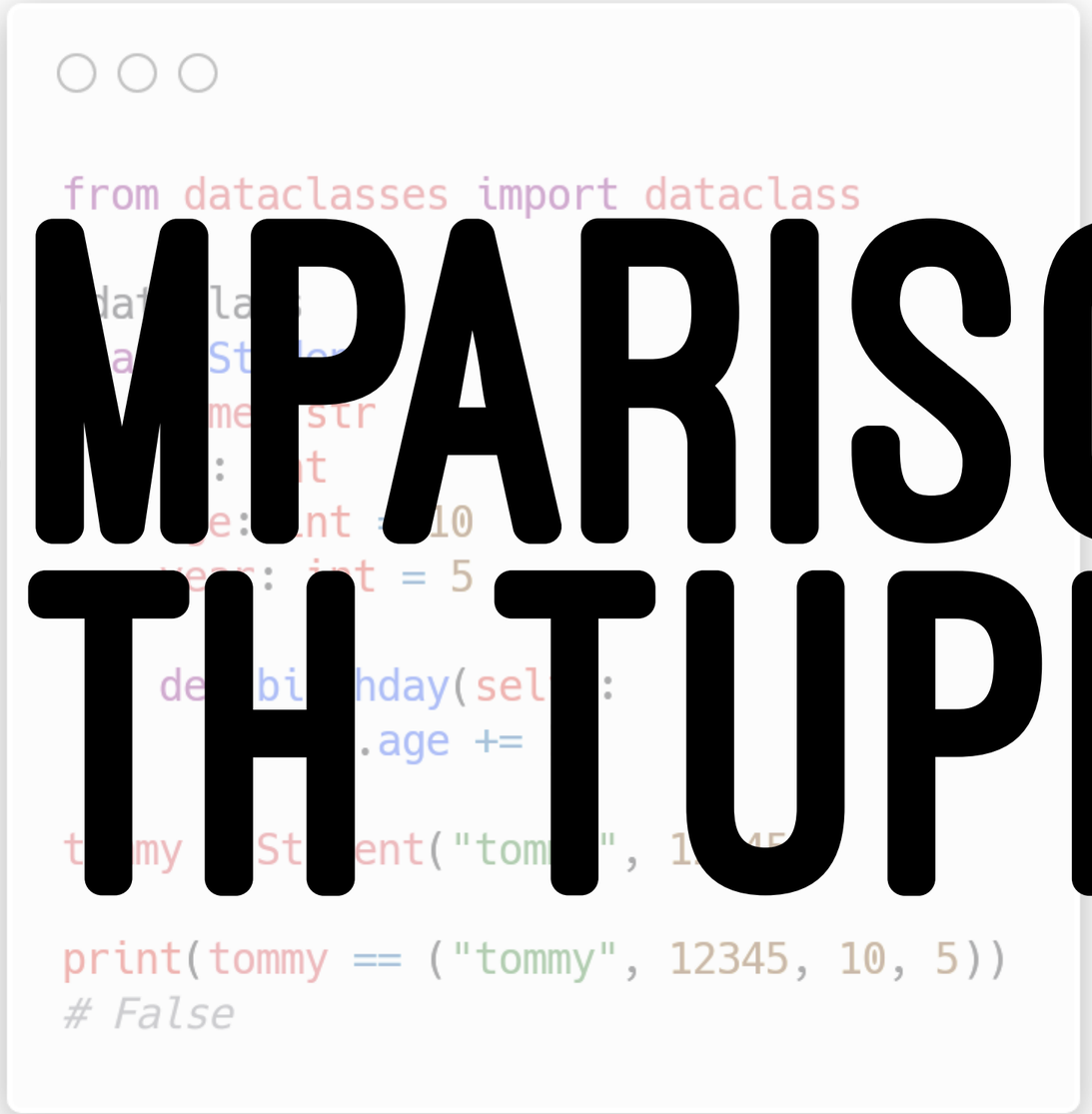
TypeError: 'Student' object is not subscriptable

```python
from dataclasses import dataclass

@dataclass
class Student:
    name: str
    id: int
    age: int = 10
    year: int = 5

tommy = Student("tommy", 12345)

print(tommy[2])
```

- MUTABLE BY DEFAULT
- NOT SUBSCRIPTABLE
- JUST PLAIN CLASSES

```python
from dataclasses import dataclass

@dataclass
class Student:
    name: str
    id: int
    age: int = 10
    year: int = 5

    def birthday(self):
        self.age += 1

tommy = Student("tommy", 12345)
tommy.birthday()

print(tommy)
# Student(name='tommy', id=12345, age=11, year=5)
```

COMPARISON WITH TUPLE

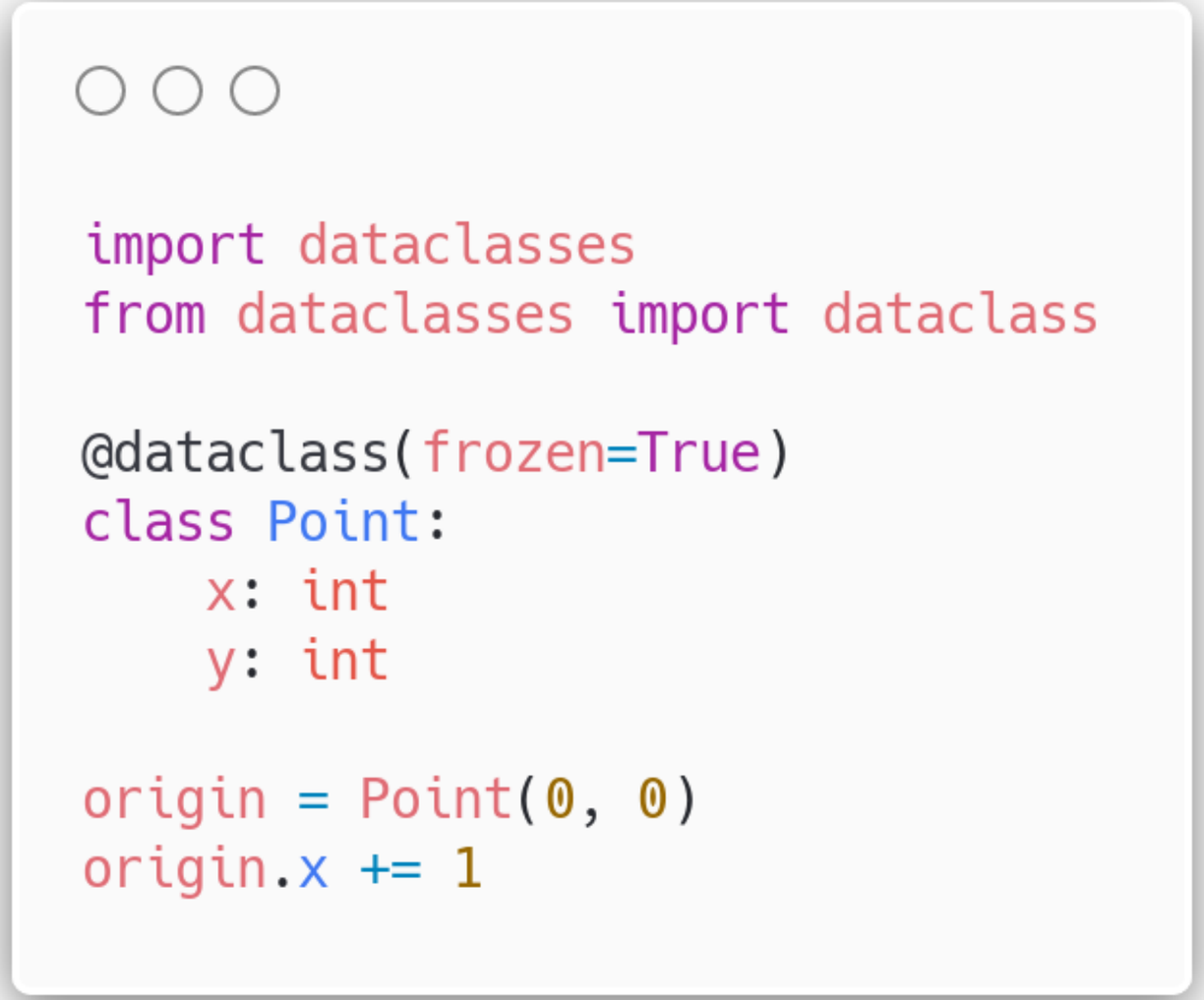


```python
from dataclasses import dataclass

@dataclass
class Student:
    name: str
    id: int
    age: int = 10
    year: int = 5

    def birthday(self):
        self.age += 1

tommy = Student("tommy", 12345)

print(tommy == ("tommy", 12345, 10, 5))
# False
```

```python
import dataclasses
from dataclasses import dataclass

@dataclass
class Student:
    name: str
    id: int
    age: int = 10
    year: int = 5

    def birthday(self):
        self.age += 1

tommy = Student("tommy", 12345)

print(dataclasses.astuple(tommy) == ("tommy", 12345, 10, 5))
# True
```

EMULATING IMMUTABILITY

```
import dataclasses
from dataclasses import dataclass

@dataclass(frozen=True)
class Point:
    x: int
    y: int

origin = Point(0, 0)
origin.x += 1
```



```python
import dataclasses
from dataclasses import dataclass

@dataclass(frozen=True)
class Point:
    x: int
    y: int

origin = Point(0, 0)
origin.x += 1
```

```
import dataclasses
from dataclasses import dataclass

@dataclass(frozen=True)
class Point:
    x: int
    y: int

origin = Point(0, 0)
origin.x += 1
```

FrozenInstanceError: cannot assign to field 'x'

```python
import dataclasses
from dataclasses import dataclass

@dataclass(frozen=True)
class Point:
    x: int
    y: int

origin = Point(0, 0)

origin.__dict__["x"] = 1

print(origin)
# Point(x=1, y=0)
```

"EMULATING"

```python
import dataclasses
from dataclasses import dataclass

@dataclass(frozen=True)
class Point:
    x: int
    y: int

origin = Point(0, 0)

origin.__dict__["x"] = 1

print(origin)
# Point(x=1, y=0)
```

dataclasses == python objects

dataclasses == python objects

tuples == C artifacts

Why not just use namedtuple?

- Any namedtuple can be accidentally compared to any other with the same number of fields. For example: `Point3D(2017, 6, 2) == Date(2017, 6, 2)`. With Data Classes, this would return False.
- A namedtuple can be accidentally compared to a tuple. For example `Point2D(1, 10) == (1, 10)`. With Data Classes, this would return False.
- Instances are always iterable, which can make it difficult to add fields. If a library defines:

```
Time = namedtuple('Time', ['hour', 'minute'])
def get_time():
    return Time(12, 0)
```

  Then if a user uses this code as:

```
hour, minute = get_time()
```

  then it would not be possible to add a `second` field to `Time` without breaking the user's code.

- No option for mutable instances.
- Cannot specify default values.
- Cannot control which fields are used for `__init__`, `__repr__`, etc.
- Cannot support combining fields by inheritance.

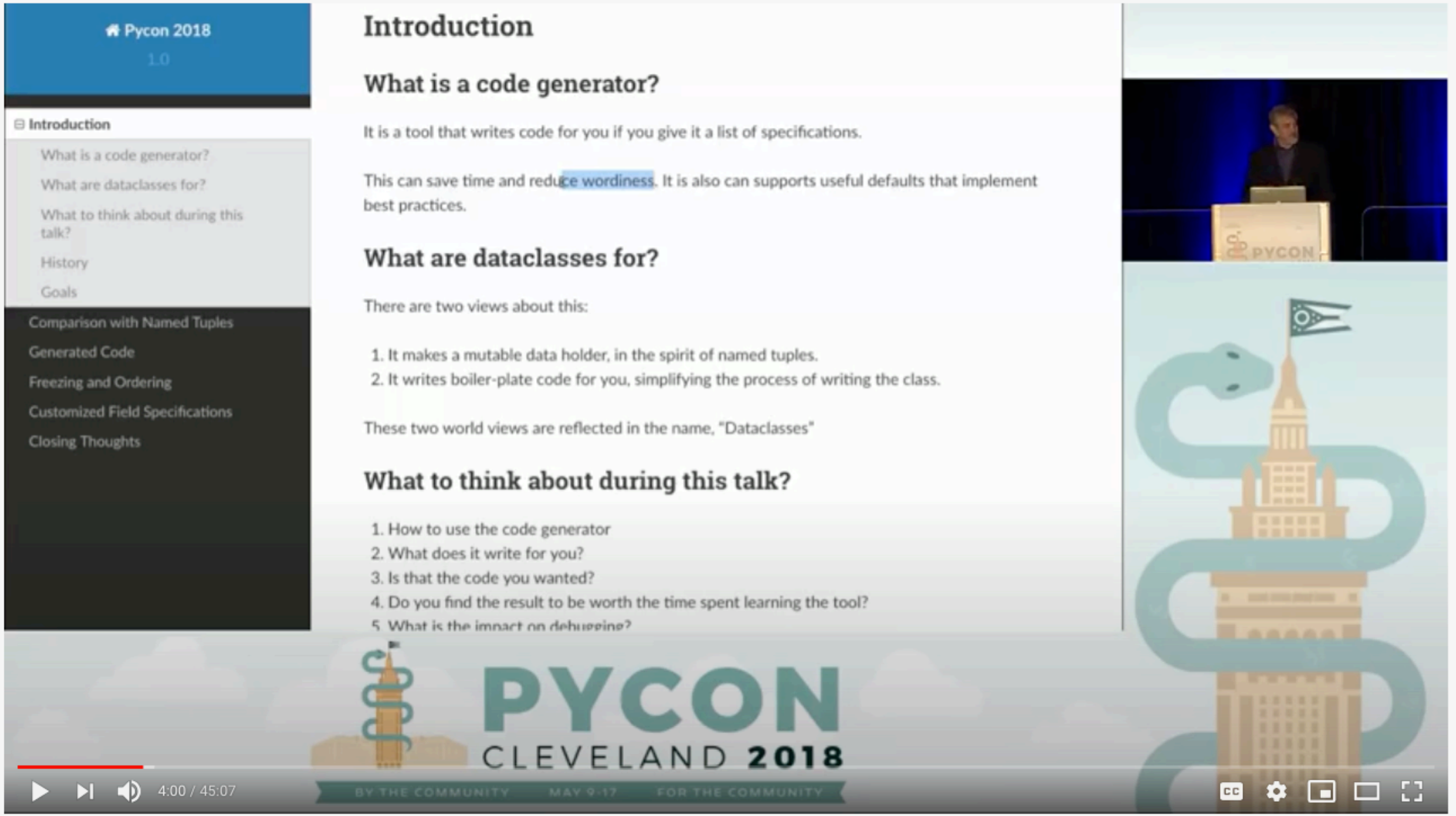
IF YOU WANT A SIMPLE TUPLE, USE TYPING.NAMEDTUPLE

IF YOU WANT A PYTHON OBJECT THAT CAN GROW IN COMPLEXITY, USE DATACLASSES

```python
import dataclasses
from dataclasses import dataclass
import math

@dataclass
class Point:
    x: int
    y: int
    magnitude: float = dataclasses.field(init=False)

    def __post_init__(self):
        self.magnitude = math.sqrt(self.x**2 + self.y**2)

pythag = Point(3, 4)

print(pythag)
# Point(x=3, y=4, magnitude=5.0)
```

__post_init__

```python
import dataclasses
from dataclasses import dataclass
import math

@dataclass
class Point:
    x: int
    y: int
    magnitude: float = dataclasses.field(init=False)

    def __post_init__(self):
        self.magnitude = math.sqrt(self.x**2 + self.y**2)

pythag = Point(3, 4)

print(pythag)
# Point(x=3, y=4, magnitude=5.0)
```



Raymond Hettinger - Dataclasses: The code generator to end all code generators - PyCon 2018

73,696 views • May 13, 2018

1K 30 SHARE SAVE

PyCon 2018
19.9K subscribers

SUBSCRIBE

TYPEDDICT

`typing.TypedDict`

- NEW IN PYTHON 3.8
- ADD TYPE HINTS TO DICTIONARY
- DOESN'T DEFINE NEW RUNTIME CLASS
- USEFUL FOR TYPECHECKERS LIKE `mypy`

```python
from typing import TypedDict

class Student(TypedDict):
    name: str
    id: int
    age: int
    year: int

tommy = Student(name="tommy", id=12345, age=10, year=5)
print(tommy)
# {'name': 'tommy', 'id': 12345, 'age': 10, 'year': 5}

print(isinstance(tommy, dict))
# True
```

WHEN TO USE `TypedDict`?

- WHEN YOU ARE PROCESSING DICTIONARIES (E.G., JSON)
- WHEN YOU WANT TO ADD STRUCTURE TO `Dict` TYPE ANNOTATIONS

EXITING THE STANDARD LIBRARY

attrs

attrs: Classes Without Boilerplate

- LIGHTWEIGHT SYNTAX FOR DECLARING CUSTOM TYPES
- BEEN AROUND SINCE PYTHON 2
- INSPIRED AND CONSULTED ON `dataclasses`

```python
import attr

@attr.s
class Point():
    x = attr.ib()
    y = attr.ib()

origin = Point(0, 0)

print(origin)
# Point(x=0, y=0)
```

VALIDATORS

```
import attr

@attr.s
class FirstQuadrantPoint():
    x = attr.ib()
    y = attr.ib()

    @x.validator
    @y.validator
    def check(self, attribute, value):
        if value <= 0:
            raise ValueError("Coordinates must be positive.")

origin = FirstQuadrantPoint(0, 0)
```

```python
import attr

@attr.s
class FirstQuadrantPoint():
    x = attr.ib()
    y = attr.ib()

    @x.validator
    @y.validator
    def check(self, attribute, value):
        if value <= 0:
            raise ValueError("Coordinates must be positive.")

origin = FirstQuadrantPoint(0, 0)
```

```
import attr

@attr.s
class FirstQuadrantPoint():
    x = attr.ib()
    y = attr.ib()
    @x.validator
    @y.validator
    def check(self, attribute, value):
        if value <= 0:
            raise ValueError("Coordinates must be positive.")

origin = FirstQuadrantPoint(0, 0)
```

ValueError: Coordinates must be positive.

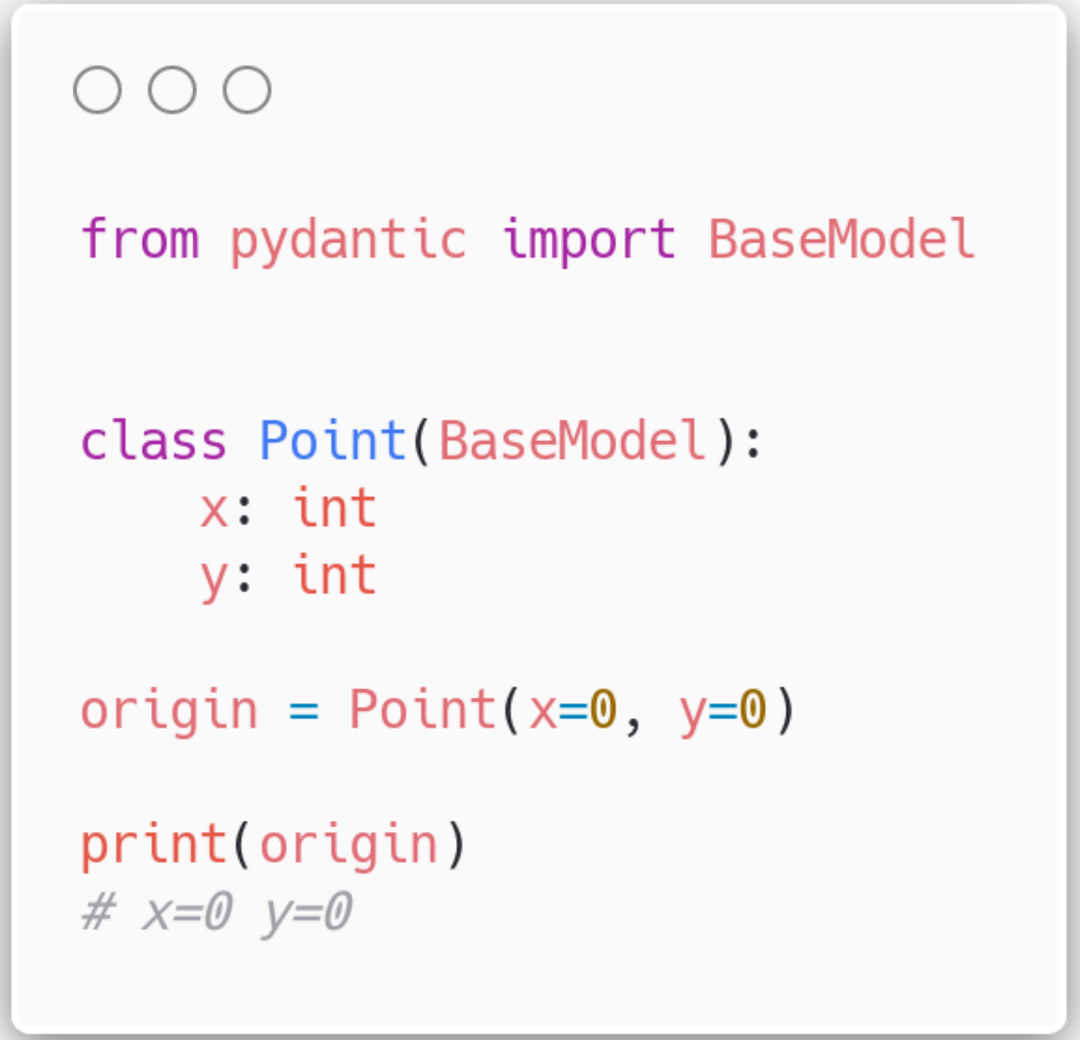
FEATURES OVER `dataclasses`:

- PYTHON 2/PYPY SUPPORT
- VALIDATORS
- CONVERTERS
- `__slots__`
- AND MORE

WHEN TO USE `attrs`:

- YOU NEED TO SUPPORT PYTHON 2 (:/)
- YOU NEED MORE ADVANCED FEATURES THAN `dataclasses` CAN PROVIDE
- YOU WANT TO BE ON THE BLEEDING EDGE


from pydantic import BaseModel
class Point(BaseModel):
x: int
y: int
origin = Point(x=0, y=0)
print(origin)
# x=0 y=0
pydantic



```python
from pydantic import BaseModel


class Point(BaseModel):
    x: int
    y: int

origin = Point(x=0, y=0)

print(origin)
# x=0 y=0
```

VALIDATORS

```python
from pydantic import BaseModel, ValidationError, validator

class FirstQuadrantPoint(BaseModel):
    x: int
    y: int

    @validator('x', 'y', allow_reuse=True)
    def must_be_positive(cls, i):
        if i <= 0:
            raise ValueError('Must be positive')
        return int(i)

origin = FirstQuadrantPoint(x=0, y=1)
```

```python
from pydantic import BaseModel, ValidationError, validator

class FirstQuadrantPoint(BaseModel):
    x: int
    y: int

    @validator('x', allow_reuse=True)
    @validator('y', allow_reuse=True)
    def must_be_positive(cls, i):
        if i <= 0:
            raise ValueError('Must be positive')
        return int(i)

origin = FirstQuadrantPoint(x=0, y=1)
```

```python
from pydantic import BaseModel, ValidationError, validator

class FirstQuadrantPoint(BaseModel):
    x: int
    y: int

    @validator('x', allow_reuse=True)
    @validator('y', allow_reuse=True)
    def must_be_positive(cls, i):
        if i <= 0:
            raise ValueError('Must be positive')
        return int(i)

origin = FirstQuadrantPoint(x=0, y=1)
```

Must be positive (type=value_error)

`pydantic`

- RECENT PROJECT TAKING ADVANTAGE OF TYPE ANNOTATIONS
- STRONG SUPPORT FOR DATA VALIDATION
- FULL `dataclasses` API AND STANDARD LIBRARY INTEROPERABILITY

dataclasses API

```python
import datetime

from pydantic.dataclasses import dataclass

@dataclass
class Conference:
    name: str
    time: datetime.datetime

pyjamas = Conference(name="PyJamas 2020", time="2020-12-05T21:00")

print(pyjamas)
# Conference(name='PyJamas 2020', time=datetime.datetime(2020, 12, 5, 21,
0))
```

```python
import datetime

from pydantic.dataclasses import dataclass

@dataclass
class Conference:
    name: str
    time: datetime.datetime

pyjamas = Conference(name="PyJamas 2020", time="2020-12-05T21:00")

print(pyjamas)
# Conference(name='PyJamas 2020', time=datetime.datetime(2020, 12, 5, 21,
0))
```

Complex serializations

```python
import datetime
from typing import List

from pydantic.dataclasses import dataclass

@dataclass
class Presentation:
    name: str
    title: str

@dataclass
class Conference:
    name: str
    time: datetime.datetime
    presentations: List[Presentation]

serialized_conference = {
    "name": "PyJamas 2020",
    "time": "2020-12-05T21:00",
    "presentations": [
        {
            "name": "Zach Anglin",
            "title": "What the struct?!",
        },
        {
            "name": "Mark Smith",
            "title": "Stupid Things I've Done with Python",
        }
    ],
}

pyjamas = Conference(**serialized_conference)

print(pyjamas.presentations[0].title)
# What the struct?!
```

```python
import datetime
from typing import List

from pydantic.dataclasses import dataclass

@dataclass
class Presentation:
    name: str
    title: str

@dataclass
class Conference:
    name: str
    time: datetime.datetime
    presentations: List[Presentation]

serialized_conference = {
        "name": "PyJamas 2020",
        "time": "2020-12-05T21:00",
        "presentations": [
            {
                "name": "Zach Anglin",
                "title": "What the struct?!",
            },
            {
                "name": "Mark Smith",
                "title": "Stupid Things I've Done with Python",
            }
        ],
    }

pyjamas = Conference(**serialized_conference)

print(pyjamas.presentations[0].title)
# What the struct?!
```

WHEN TO USE `pydantic`:

- WHEN YOU NEED POWERFUL DATA VALIDATION
- WHEN YOU'RE VALIDATING EXTERNAL DATA THROUGH AN API
- WHEN YOU NEED TO DESERIALIZE A COMPOUND STRUCTURE
- IF YOU'RE IN A TYPED CODEBASE :)

RECAP

- C structs
- Python structs
- collection.namedtuple
- typing.NamedTuple
- dataclasses
- typing.TypedDict
- attrs
- pydantic

THERE SHOULD BE ONE-- AND PREFERABLY ONLY ONE --OBVIOUS WAY TO DO IT.

- THE ZEN OF PYTHON

FIN