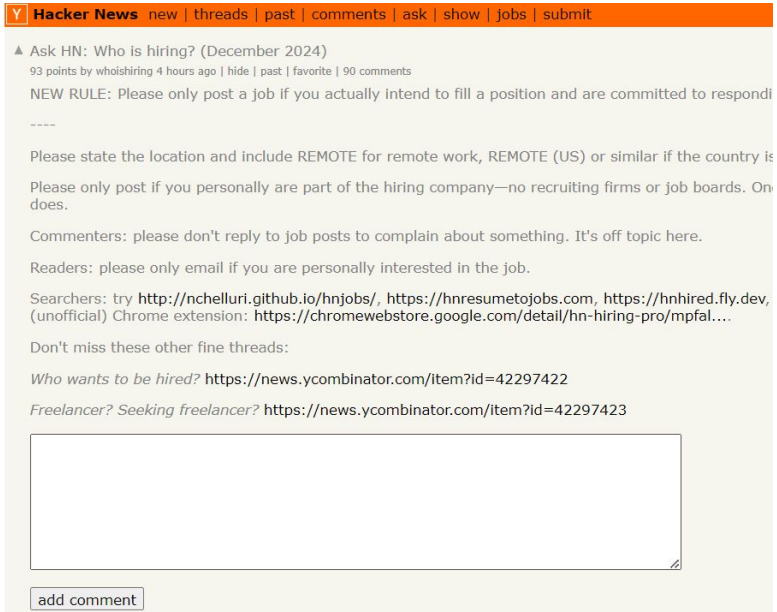
Advanced Abstraction

Agenda - Schedule

1. Warm-Up
2. OOP
3. Dataclasses
4. TLAB



Rediscover the lost art of finding [jobs on forums](#).

Agenda - Goals

- Learn about the different coding paradigms and the utility of OOP
- Implement data classes in Python
- Learn about the distinction between classes and objects

Warm-Up

```
def modify(prices: dict, adjusted: float) -> list:
        former = []

        for k in prices:
          former.append(prices[k])
          prices[k] *= (1 + adjusted)

        return former

imports = {
  “RTX 3060”: 299.99,
  “1 kg coffee”: 6.102,
  “1 lb avocados”: 2.01,
  “Thinkpad X13”: 889.01,
}

result = modify(imports, -0.06)
print(imports)
```

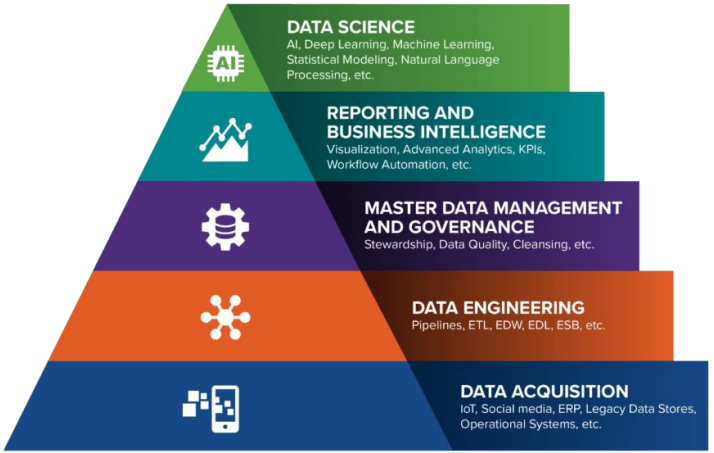
Join your pod groups and evaluate this chunk of code. Work together to figure out what will occur when we run this code.

Phase 1 Recap

Course Goals - Abstract

In the shallow sense of things, this fellowship aims to teach you different skills that **build off of each other**. In the abstract, the goals entail:

Phase 1: *Learn how to engineer, pipeline, and collect*
Phase 2: *Learn how to analyze, predict, model, and innovate*
Phase 3: *Learn how to manage, collaborate, visualize, and inform*



Stack & Heap

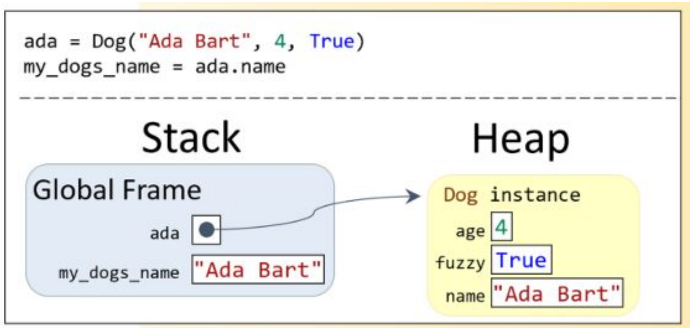
An Aside - Stack vs Heap

Both the stack and heap exist in your RAM (random access memory).

Often times, we have shown you this diagram without an explanation. Let's rectify this and formally define a "stack" and "heap", and **furthermore define their differences**.

Stack: Memory set dedicated for local variables and function calls.

Heap: Dynamic memory set with no defined pattern of memory allocation/deallocation.



The name, “stack”, defines the First-in-Last-out (FILO) structure of this memory concept.

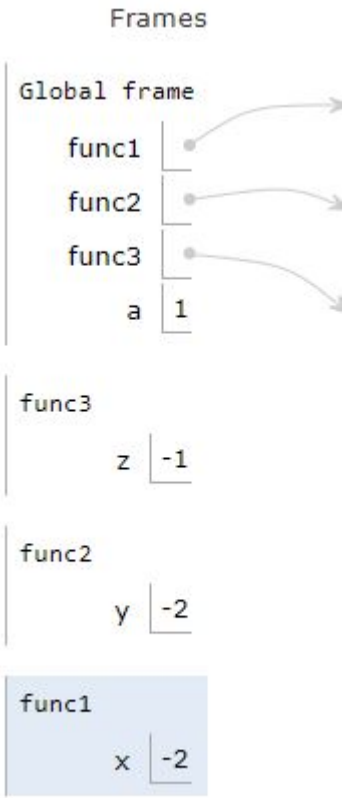
An Aside - Stack

Our program utilizes the stack to keep track of the order of function calls and initialize local variables that exist in specific **function bodies**.

Notice that each stack frame defines **variables, their value**, and **which scope the variables are instantiated in.**

Python 3.6
known limitations

```
1  def func1(x):
2      x = x + 2
3      return x
4
5  def func2(y):
6      y = y * 2
7      return func1(y)
8
9  def func3(z):
10     z = z - 2
11     return func2(z)
12
13 a = 1
14 print(func3(a))
```



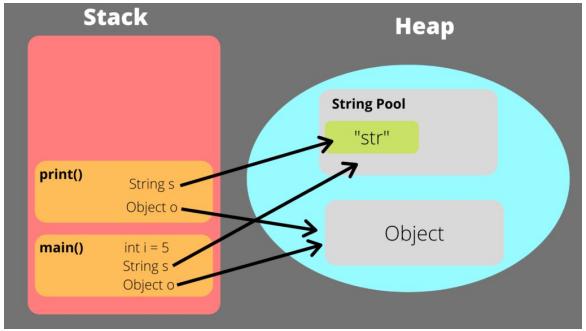
The name, “heap”, defines the unstructured nature of this memory concept.

An Aside - Heap

The heap, on the other hand, does not use any intrinsic pattern to “allocate/deallocate” (create/delete) memory.

Python handles interaction with the heap **automatically**, so we do not have to worry about things like memory size or freeing up memory.

However, we should understand this concept to handle issues involving **copies vs deep copies**.

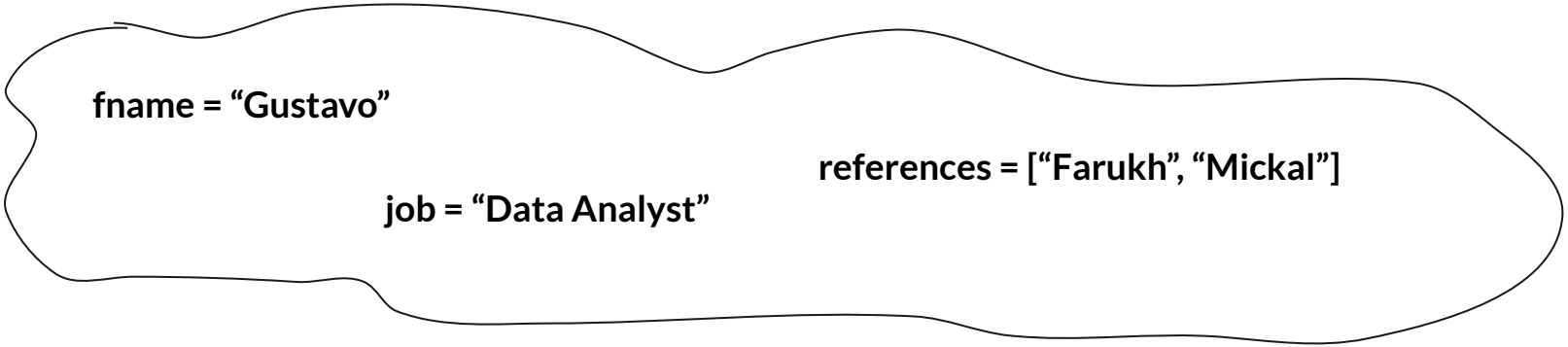


Abstraction

Conceptual OOP

While we could use data-structures to satisfy this, could you think how this could introduce its' own complexity?

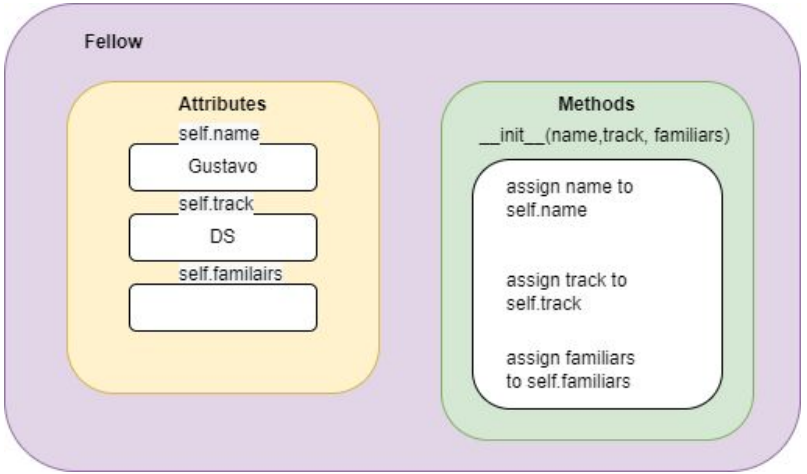
Now that we've learned about various ways to store data, a challenge that you should be considering is "**how do we associate and label these bits of data"?** For example, let's say we are developing an applicant tracking system. This will entail the storage of a variety of data-types whose purpose might get lost without sufficient context.



Conceptual OOP - data_intake.py

Instead what we want to do is **bundle all this information in a neat package.** We want to **abstract and encapsulate. AKA we want to create a class.**

Before we get into this however, let's go over the different types of **programming paradigms**.



Programming Paradigms

It turns out that there are many different “styles” of programming aka programming paradigms

Imperative (aka von Neumann)

- **Procedural:** instructions are executed in strict order
- **Object-Oriented**: groups of objects change state

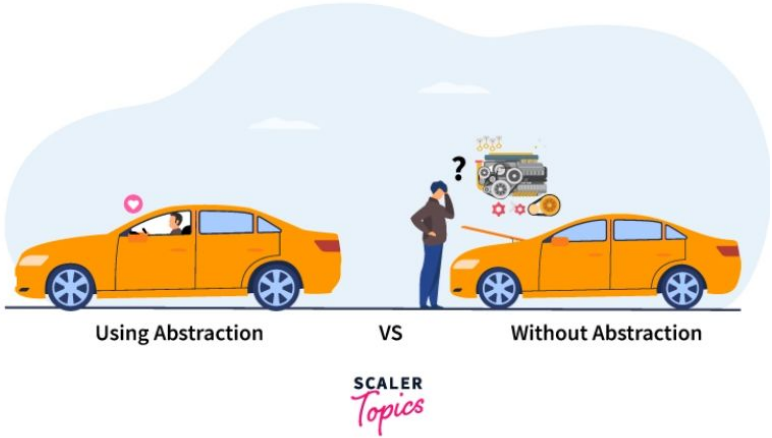
Declarative

- **Functional**: series of functional applications
- **Logical**: query-based programming
- **Reactive**: based on data streams and change

John von Neumann

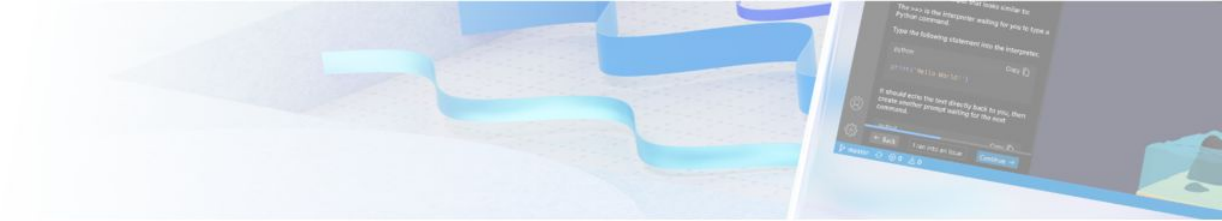
Programming Paradigms - OOP

The concept of **abstraction** (hide all irrelevant details) and **encapsulation** (bundling all data and functions into one unit) using classes, objects, and methods.



Dataclasses

Introduction



Composite types are composed of other types, unlike *primitive* types. The first kind of composite type you'll learn about is the dataclass.

- A *dataclass* is a way to make new types that collect multiple values into one place.
- Dataclasses are data structures that you can use to combine other types to create a new type.
- A dataclass is a kind of type, not a type itself.

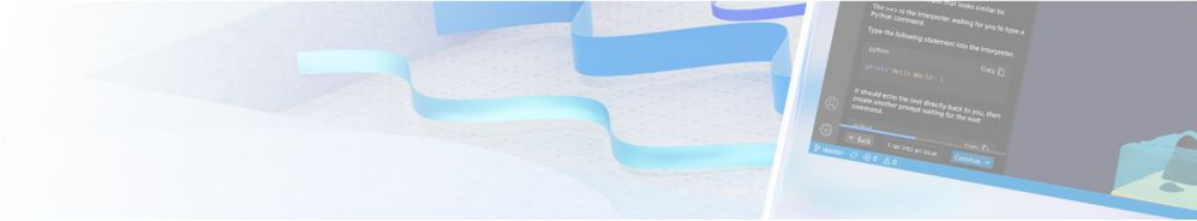
An Example Dataclass

- **Example**: You must import a decorator from the built-in dataclasses module.

- Create a definition for the dataclass by writing the header and body of the dataclass.

```
# Required import
from dataclasses import dataclass
# Definition
@dataclass
class Box:
    width: int
    length: int
```

An Example Dataclass

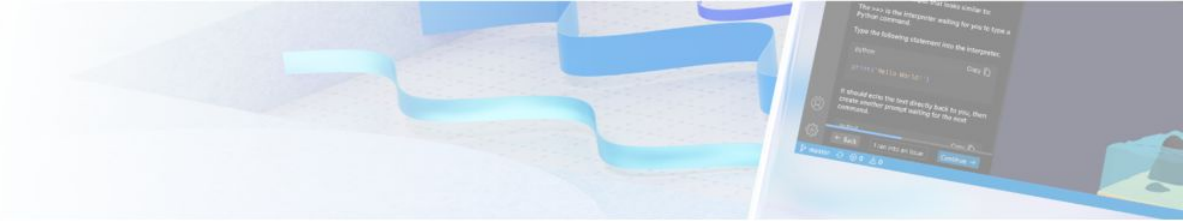


- The header requires:
  - The decorator
  - The `class` keyword
  - The name of the new dataclass
  - A colon (:)

- The body requires that you specify the fields, each on its own line.

```
# Required import
from dataclasses import dataclass
# Definition
@dataclass
class Box:
    width: int
    length: int
```

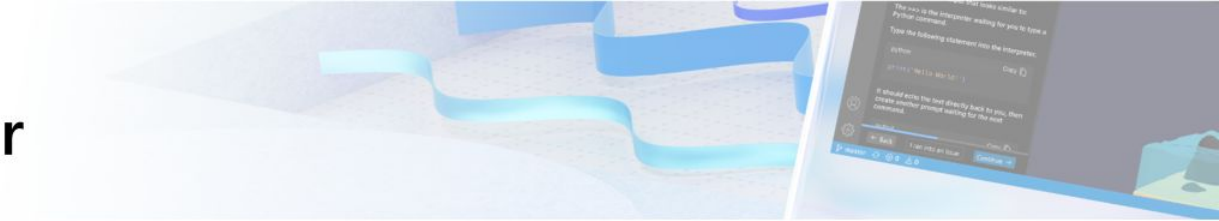
An Example Dataclass



```
# Instance creation
my_box = Box(5, 10)

# Using the instance
print(my_box)
```

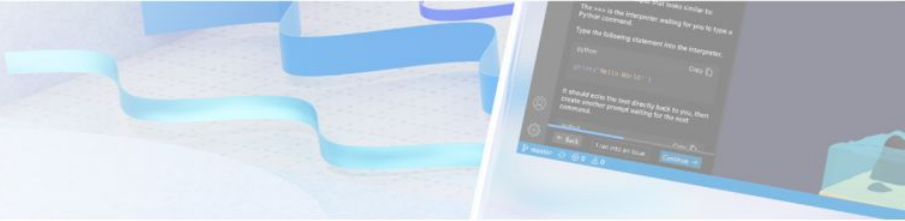
- You can create instances of the dataclass by using the *constructor* function.
- **Example**: Store the result in a variable to use the instance later.

@dataclass Decorator



- The *decorator* is the `@dataclass` annotation.
- The "at" symbol (@) is the syntax that makes this a decorator.
- A decorator adds extra functionality on top of classes — in this case, to make the classes into dataclasses.
- The decorated dataclasses are far more powerful and convenient than regular classes.

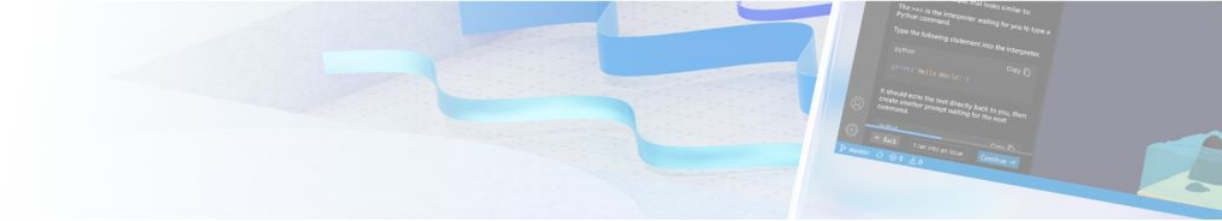
@dataclass Decorator



- You must `import` the `dataclass` decorator from the `dataclasses` module without including the @ symbol.
- When you put the decorator above the class definition, you must include the @ symbol.
- Doing either part incorrectly prevents Python from understanding that you want to create a dataclass.

```
from dataclasses import dataclass

@dataclass
class Dog:
    pass
```

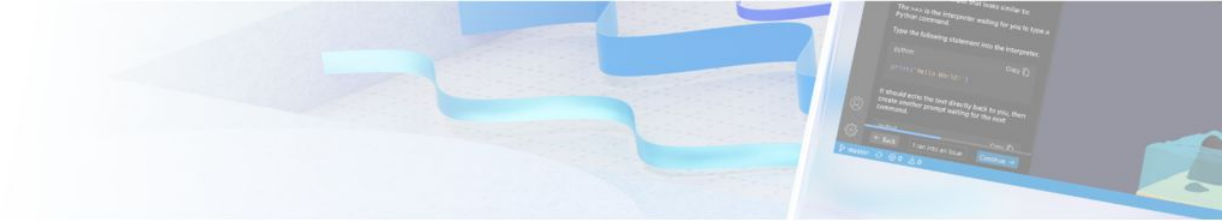
Dataclass Fields



- You can use the fields of a dataclass to bundle up related data into a single place.
- Each field is like a variable inside of the dataclass.
- The rules for naming fields are:
  - The name must have only letters, digits, and underscores.
  - The name must only begin with letters or underscores.

```
from dataclasses import
dataclass

@dataclass
class Dog:
    name: str
    age: int
    is_fuzzy: bool
```

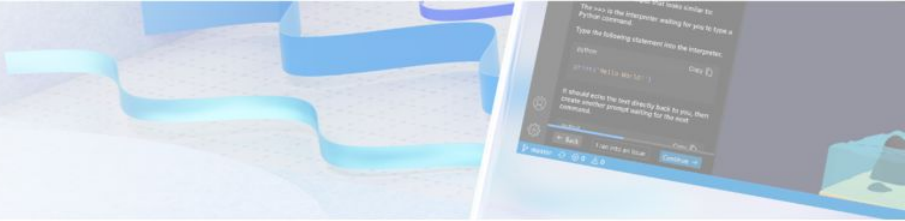
Dataclass Fields



```
from dataclasses import
dataclass

@dataclass
class Dog:
    name: str
    age: int
    is_fuzzy: bool
```

- The rules for specifying the field type are like the rules for specifying the function parameter type.
- Instead of using commas, place each field on its own line.

Check Your Understanding



Question 1

How many fields are in the dataclass shown here?

```
from dataclasses import
dataclass

@dataclass
class Dog:
    name: str
    age: int
    is_fuzzy: bool
```

- ❑ 1
- ❑ 6
- ❑ 3

Check Your Understanding

Question 1

How many fields are in the dataclass shown above?

```
from dataclasses import
dataclass

@dataclass
class Dog:
    name: str
    age: int
    is_fuzzy: bool
```

- ❑ 1
- ❑ 6
- ✓ 3

  The three fields are `name`, `age`, and `is_fuzzy`.

The Constructor Function

- After you define a dataclass, you can create instances that represent concrete versions of the original dataclass.
- You create instances of a dataclass by calling the constructor function.
- The constructor is a special function created by the dataclass definition and given the same name.

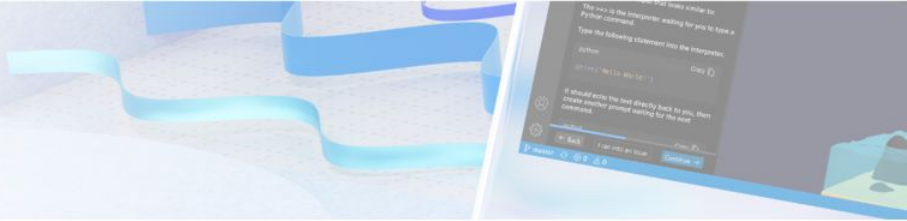
```
from dataclasses import dataclass
@dataclass
class Dog:
    name: str
    age: int
    is_fuzzy: bool
ada = Dog("Ada Bart", 4, True)
```

The Constructor Function

```
from dataclasses import dataclass
@dataclass
class Dog:
    name: str
    age: int
    is_fuzzy: bool
ada = Dog("Ada Bart", 4, True)
```

- Calling a function and *instantiating* ("creating an instance of") a dataclass are similar.
- Each field of the dataclass definition corresponds to an argument for the constructor.
- The order and type of each field must match the order and type of each argument.

Check Your Understanding

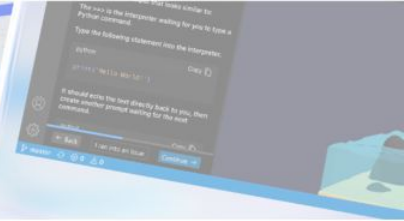


Question 2

How many arguments do you need for a constructor function of a dataclass with five fields?

- ❑ Five arguments
- ❑ One argument
- ❑ Three arguments

Check Your Understanding



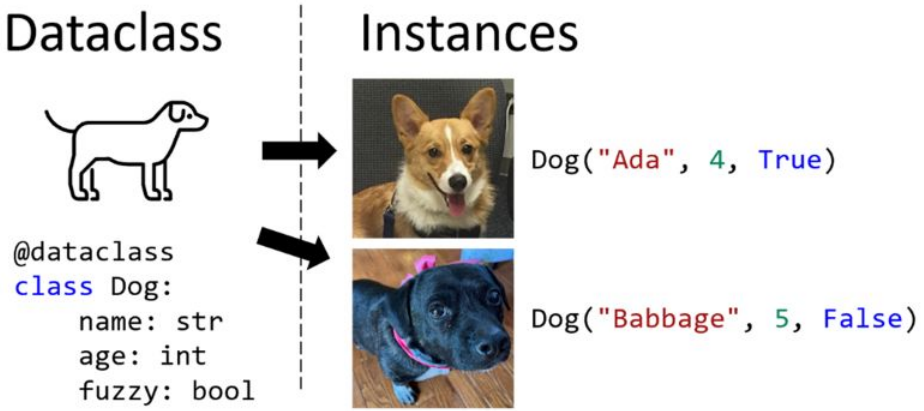
Question 2

How many arguments do you need for a constructor function of a dataclass with five fields?

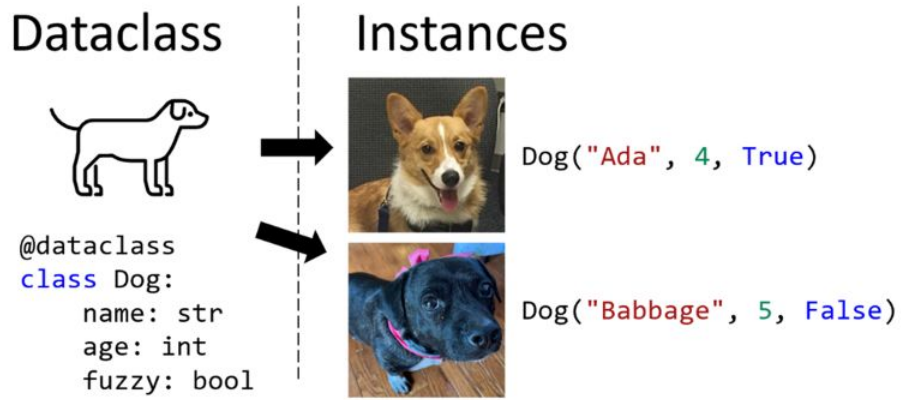
- ✓ Five arguments
  Each field needs a corresponding argument.
- ❑ One argument
- ❑ Three arguments

Class vs. Instance

- What does a class represent compared to an instance of the class? One of these is an *idea*, the other is a concrete thing.
- **Example**: When you look only at the definition of the `Box`, it's impossible to say what color a box is.
- Although the box color is a string value, there's no specific string value associated with the idea of a `Box`.
- You need to create an instance of a `Box`.

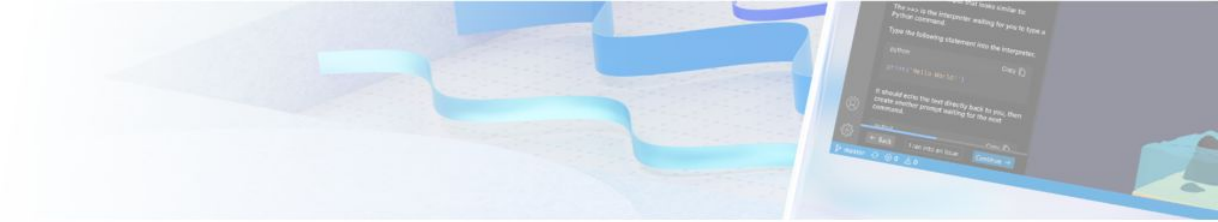


Class vs. Instance



- You can create an instance by using the constructor function that is `"red"` or `"blue"`, but the original `Box` object itself doesn't inherently have a color.
- **Dataclass**: The pattern used as a base for instances.
- **Instance**: A specific, concrete example of a class.

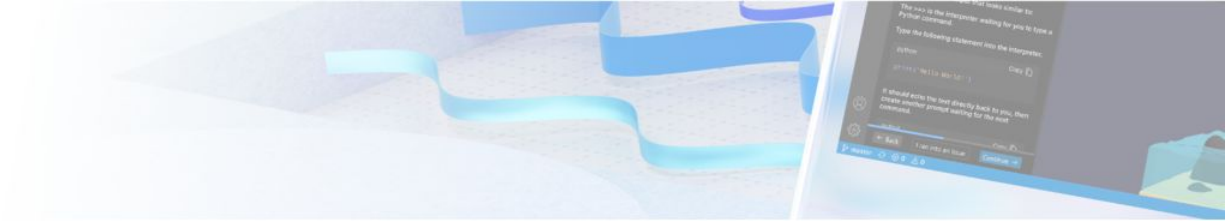
Accessing Fields



- You use the period and the name of the field to access fields. Fields are variables that live inside an instance.
- **Example**: Access the `name` field of the `Dog` instance stored in `ada`.
- You can only access the fields of an instance, not a dataclass.

```
from dataclasses import dataclass
@dataclass
class Dog:
    name: str
    age: int
    is_fuzzy: bool
ada = Dog("Ada Bart", 4, True)
print("Ada's name is", ada.name)
```

Accessing Fields



```
from dataclasses import dataclass
@dataclass
class Dog:
    name: str
    age: int
    is_fuzzy: bool
ada = Dog("Ada Bart", 4, True)
print("Ada's name is", ada.name)
```

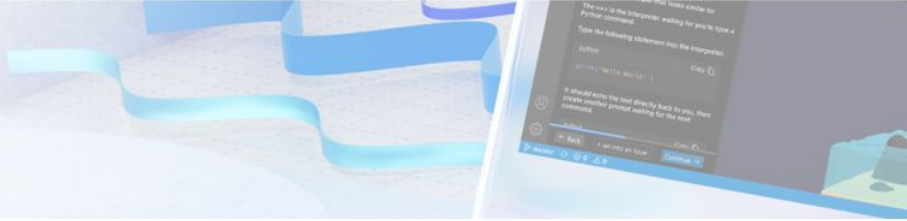
- You can't try to access the fields of the `Dog` dataclass, only the `ada` instance.
- If you try changing the code, you'll get an error when you try to access `Dog.name` instead of `ada.name`.
- `Dog` is an abstract idea, while the `ada` instance is a concrete bit of data.

Accessing Fields

- The term *field* can be used interchangeably with *attribute*, *property*, and *member*.

- You'll use the terms *field* and *attribute* most often, although you might refer to properties or members.

Check Your Understanding



Question 3

Which of the following correctly accesses the age of the `Dog` instance shown in the example?

```
from dataclasses import dataclass
@dataclass
class Dog:
    name: str
    age: int
    is_fuzzy: bool
ada = Dog("Ada Bart", 4, True)
print("Ada's name is", ada.name)
```

- ❑ `age`
- ❑ `Dog.age`
- ❑ `ada.age`

Check Your Understanding

Question 3

Which of the following correctly accesses the age of the `Dog` instance shown in the example?

```
from dataclasses import dataclass
@dataclass
class Dog:
    name: str
    age: int
    is_fuzzy: bool
ada = Dog("Ada Bart", 4, True)
print("Ada's name is", ada.name)
```

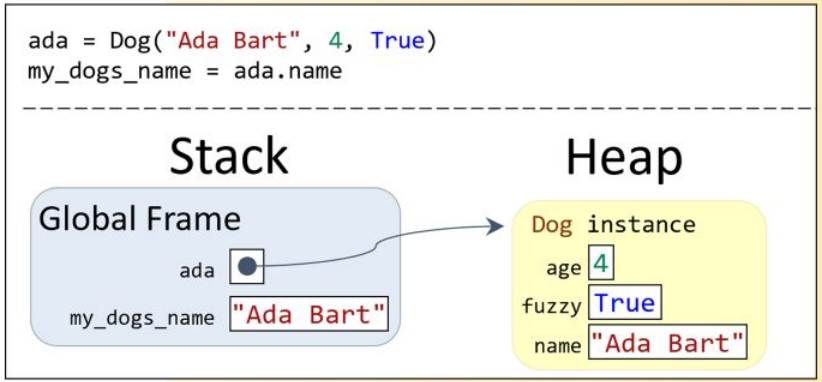
- ❑ `age`
- ❑ `Dog.age`
- ✓ `ada.age`
  The variable `ada` holds an instance of a `Dog`, so you can access its `age` attribute.

Tracing Dataclasses

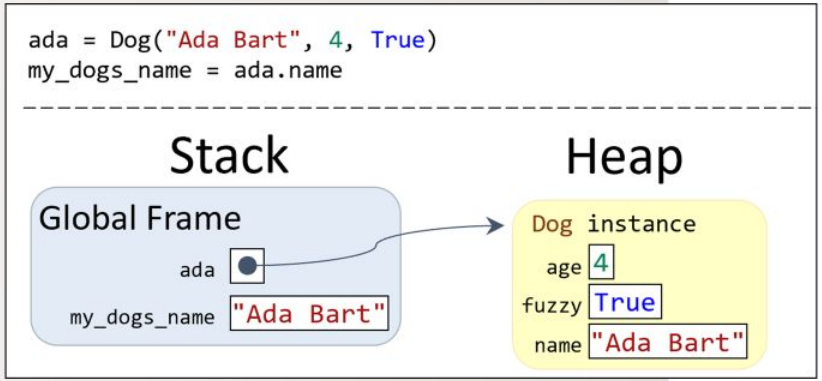
- The data stored in a dataclass includes multiple pieces of data (the fields), all stored in a single object.
- The data for an instance lives in an area of memory called the *heap*.
- The heap is a special region of memory with many interesting properties. Variables on the stack can hold references to values on the heap.

Tracing Dataclasses

- Dataclasses are traced differently compared to regular primitive types of data.
- Calling the constructor creates a new dataclass instance.
- That instance is a special kind of value with multiple fields, visible on the right side of the diagram.
- When code accesses one of those fields, like on the second line (`ada.name`), the arrow is followed, and the named field's value is reached.



Tracing Dataclasses

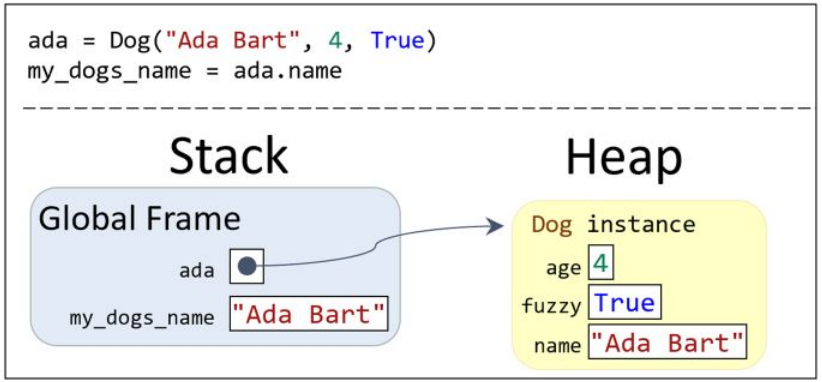


- Because that value is a simple primitive string, you can store it directly in the `my_dogs_name` variable.
- The `ada` variable points to the entire `Dog` instance, unlike the `my_dogs_name` variable, which points to a simple primitive value.
- Keep track of the difference between primitive values stored in the stack and special instances stored on the heap.

Check Your Understanding

Question 4

In the stack/heap diagram shown here, what does the arrow represent?



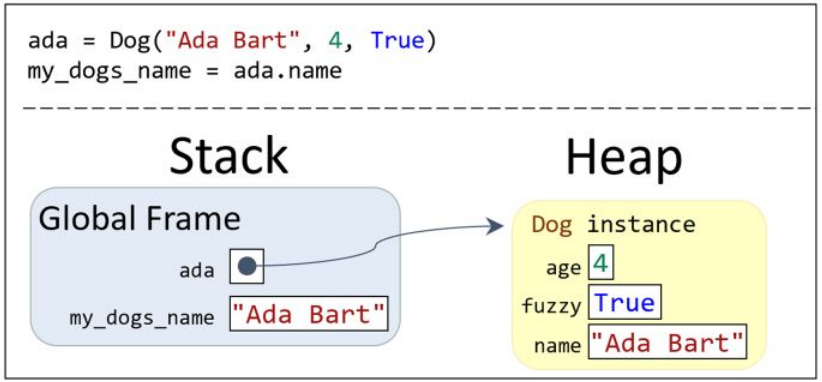
- ❑ The `ada` variable is equal to the string `"Ada Bart"`
- ❑ The variable `ada` is holding an instance of a `Dog` dataclass.
- ❑ The `ada` function calls the `Dog` constructor.

Check Your Understanding

Question 4

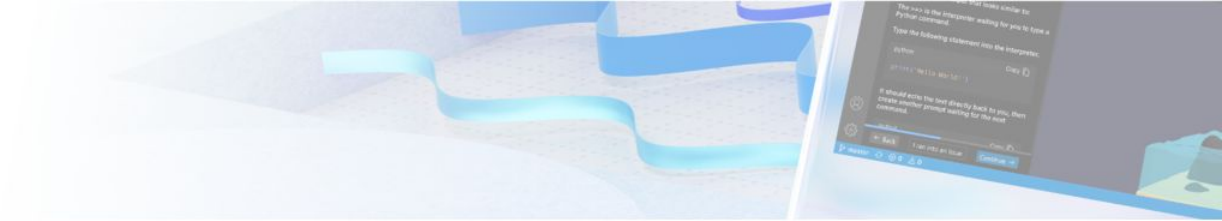
In the stack/heap diagram shown here, what does the arrow represent?

- ❑ The `ada` variable is equal to the string `"Ada Bart"`
- ✓ The variable `ada` is holding an instance of a `Dog` dataclass.
- ❑ The `ada` function calls the `Dog` constructor.



Dataclass Operations

Using Dataclasses

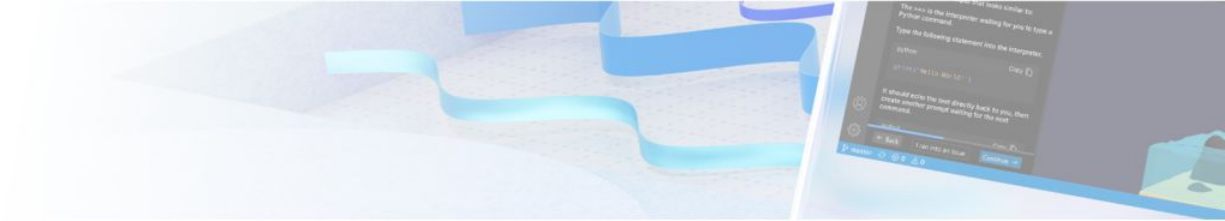


- The *equality* (`==`) and *inequality* (`!=`) operators do work with dataclasses.

- When you check whether two dataclasses are equal, each field is checked to make sure they are equal.

```
# Both of these are True
print(red_circle != blue_circle)
print(red_circle == Circle(5, "red"))
```

Using Dataclasses

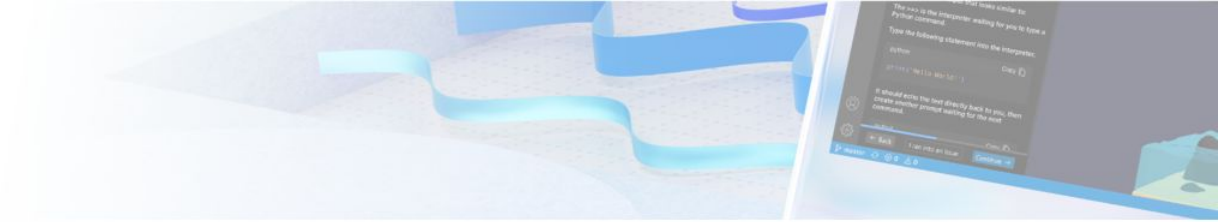


- You won't need to think about equality with dataclasses directly.

- Instead, you'll *access the fields of a dataclass* to perform operations using the data stored inside.

```
from dataclasses import dataclass
@dataclass
class Circle:
    radius: int
    color: str
```

Using Attributes



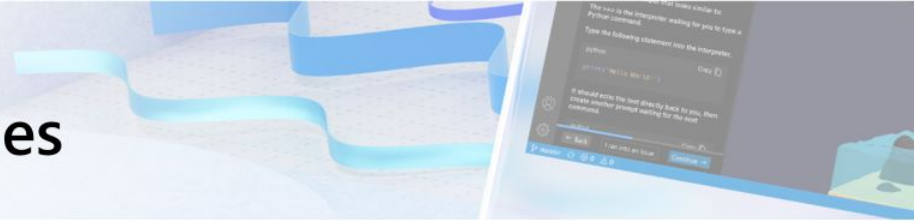
- You can access the fields of instances to use their values.
- You need:
  - The name of the variable holding the instance
  - A period (`.`)
  - The name field

```
from dataclasses import dataclass
@dataclass
class Rectangle:
    length: int
    width: int
box = Rectangle(5, 3)
print("The box area is", box.length * box.width)
```

Using Attributes

- A *method* is just an attribute that holds a function.
- The important thing to focus on is that you can access fields from instances.
- This is especially useful when you pass dataclasses into functions.

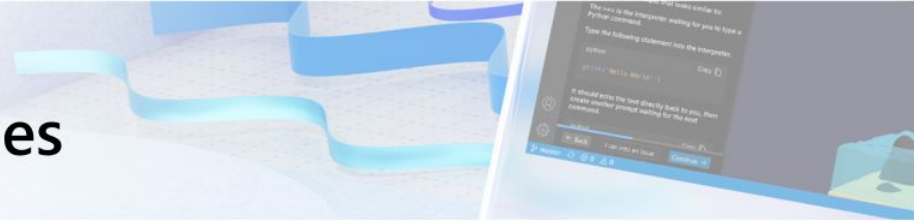
Functions Consuming Dataclasses



Example: The multiplication is inside a function to create a reusable `area` function that consumes a `Rectangle` instance and produces an integer.

```
from bakery import assert_equal
from dataclasses import dataclass
@dataclass
class Rectangle:
    length: int
    width: int
def area(rect: Rectangle) -> int:
    return rect.length * rect.width
box = Rectangle(5, 3)
assert_equal(area(box), 15)
```

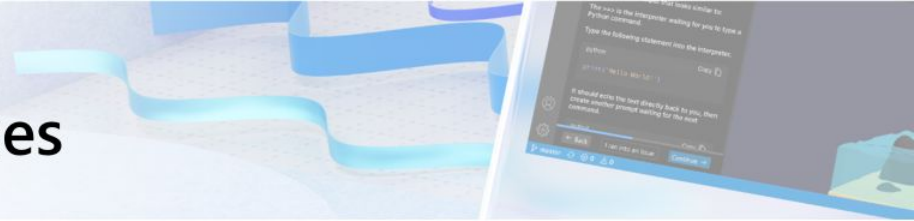
Functions Consuming Dataclasses



```
from bakery import assert_equal
from dataclasses import dataclass
@dataclass
class Rectangle:
    length: int
    width: int
def area(rect: Rectangle) -> int:
    return rect.length * rect.width
box = Rectangle(5, 3)
assert_equal(area(box), 15)
```

1. Import the `assert_equal` function and the `dataclass` decorator. Imports always go at the top of a program.
2. Define a dataclass named `Rectangle`, which will be the name of the constructor function *and* the newly created type `Rectangle`. The dataclass has two fields: `length` and `width`.
3. Define a function named `area` that consumes a `Rectangle` and produces an integer.
4. Call the constructor function (`Rectangle`) with the values 5 and 2, which creates a new `Rectangle` instance and assigns the result to a new variable `box`. Give `box` whatever name you want.

Functions Consuming Dataclasses



Continued

5. In the last line of code, unit-test the `area` function by calling the function inside an `assert_equal` call. The `Rectangle` instance associated with the variable box is passed as an argument.
6. In the header of the `area` function, one parameter named `rect` is a `Rectangle`. The dataclass type name `Rectangle` to the right of the colon indicates the parameter type.
7. On the left, choose a name for the local variable that holds the instance. The concise name `rect` could have been clearer, like `Rectangle` or `a_rectangle`.

```
from bakery import assert_equal
from dataclasses import dataclass
@dataclass
class Rectangle:
    length: int
    width: int
def area(rect: Rectangle) -> int:
    return rect.length * rect.width
box = Rectangle(5, 3)
assert_equal(area(box), 15)
```

Functions Consuming Dataclasses

Continued

8. When the `area` function is called, the `Rectangle` instance used as an argument is assigned to the `rect` parameter.
9. The body of the function runs, and the function can use the fields of the instance stored in `rect` to calculate and return the area.
10. The returned integer from the `area` function call is compared against the expected value 15, which is correct, causing the unit test to pass.

```
from bakery import assert_equal
from dataclasses import dataclass
@dataclass
class Rectangle:
    length: int
    width: int
def area(rect: Rectangle) -> int:
    return rect.length * rect.width
box = Rectangle(5, 3)
assert_equal(area(box), 15)
```

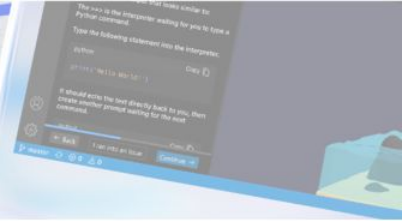
- Notice that the `rect` parameter matches the previously defined dataclass.
- The `assert_equal` function/call matches that parameter to the corresponding argument, in this case the variable `box`, which contains an instance of the `Rectangle` dataclass created using its constructor.

Functions Consuming Dataclasses

- The function expects a Rectangle, so you couldn't pass in a `Circle` to the `area` function. The types don't match, and the field accesses would be wrong.

- The `Circle` dataclass doesn't have `length` or `width` fields — only the `radius` field.

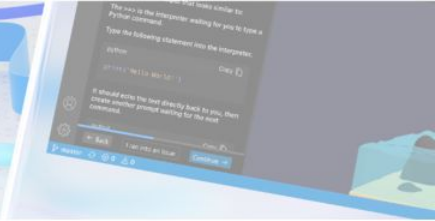
Common Mistakes with Dataclass Functions



- When you call a function that consumes a dataclass, pass in an *instance* instead of a *type*.
- You can pass in an instance directly or assign the instance to a variable and use the variable instead.
- Don't try to pass the type directly into the function — the type is an abstract idea instead of a concrete instance.
- Don't pass the fields into the dataclass directly. You need to pass a single instance, not the individual parts.

```
# Good
area(Rectangle(5, 3))

# Also good
my_box = Rectangle(5, 3)
area(my_box)

# Does not work!
area(Rectangle)

# Also does not work!
area(my_box.length, my_box.width)
```

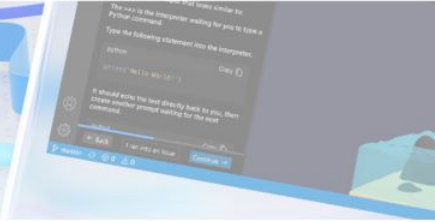
Common Mistakes with Dataclass Functions



- Don’t use the parameter type inside of the function definition instead of the parameter itself.
- Avoid mistakenly using the fields without referring to the instance.
- Use the parameter variable when you need to access the fields of an instance.

```
# This is good
def area(rect: Rectangle) -> int:
    return rect.length * rect.width

# This does not work
def area(rect: Rectangle) -> int:
    return Rectangle.length * Rectangle.width

# This does not work either!
def area(rect: Rectangle) -> int:
    return length * width
```

Common Mistakes with Dataclass Functions

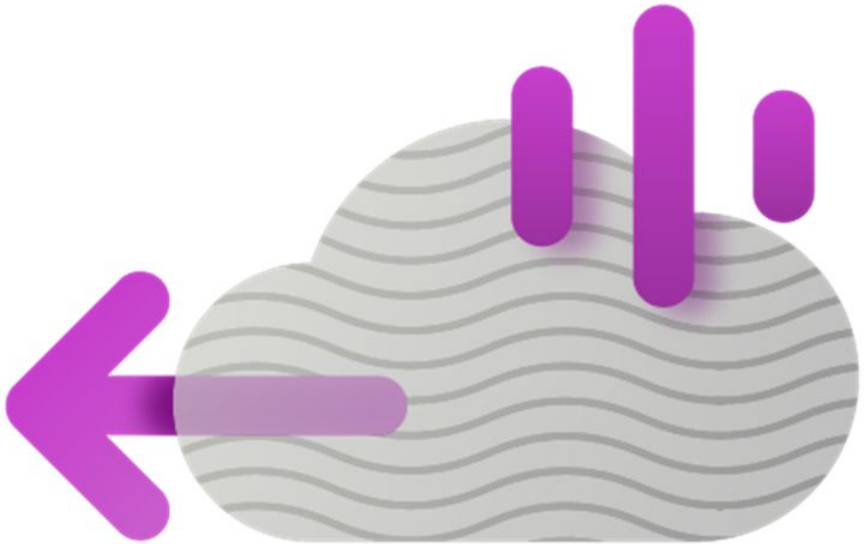


- The recurring theme is the same for both kinds of mistakes: You can use only the *instance* instead of the *type*.

- In the same way that you can add `5+3` but not `5+int`, you can access the fields of an instance but not of a dataclass type. The type is a theoretical idea instead of a concrete value.

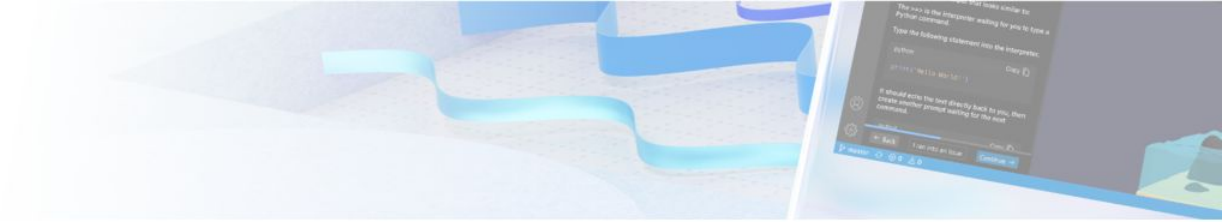
```
# This is good
def area(rect: Rectangle) -> int:
    return rect.length * rect.width

# This does not work
def area(rect: Rectangle) -> int:
    return Rectangle.length * Rectangle.width

# This does not work either!
def area(rect: Rectangle) -> int:
    return length * width
```

Fields as Arguments



- There's nothing special about the values stored in the fields of instances.
- The attribute access expression using the period is no different from other expressions, such as a variable lookup, math operation, or function call.
- That means you can pass the result of such expressions to other function calls or math operations, or you can even return them.

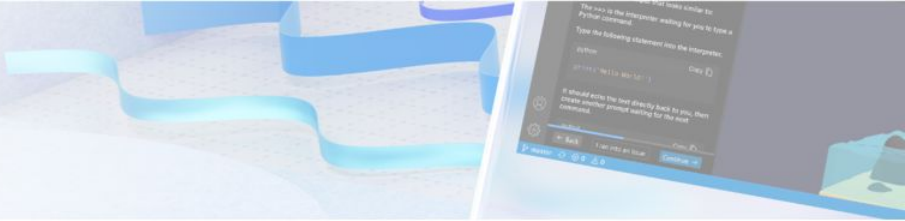
Fields as Arguments



- **Example**: Use the `str` function by passing in the values associated with the fields of the given `rect` instance to create a nice string representation of the instance.
- Think about whether the function needs the *entire* instance or just a specific field from the instance.
- Then, decide whether to define a function that consumes the instance or a function that consumes a field from the instance.

```
from dataclasses import dataclass
@dataclass
class Rectangle:
    length: int
    width: int
def as_string(rect: Rectangle) -> str:
    # Accessing fields is just another kind of expression
    return str(rect.length) + "x" + str(rect.width)
box = Rectangle(4, 5)
print("The box size is " + as_string(box))
```

Check Your Understanding



Question 2

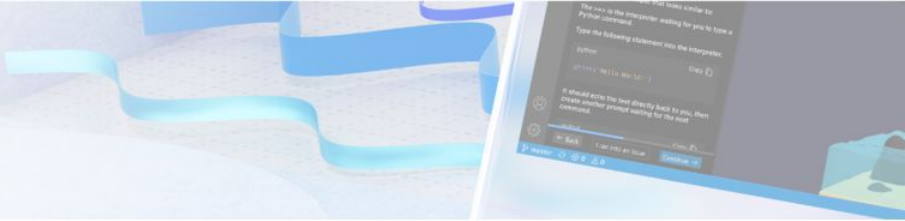
Given the following code:

```
from dataclasses import dataclass
from bakery import assert_equal
@dataclass
class Fruit:
  kind: str
  sweetness: int
def is_sweet(fruit: Fruit) -> bool:
  return fruit.sweetness > 10
assert_equal(is_sweet(Fruit("Orange", 15)), True)
assert_equal(is_sweet(Fruit("Lemon", 5)), False)
```

What type should the `fruit` parameter be?

- ❑ Str
- ❑ Bool
- ❑ Fruit
- ❑ Int

Check Your Understanding



Question 2

Given the following code:

```
from dataclasses import dataclass
from bakery import assert_equal
@dataclass
class Fruit:
  kind: str
  sweetness: int
def is_sweet(fruit: Fruit) -> bool:
  return fruit.sweetness > 10
assert_equal(is_sweet(Fruit("Orange", 15)), True)
assert_equal(is_sweet(Fruit("Lemon", 5)), False)
```

What type should the `fruit` parameter be?

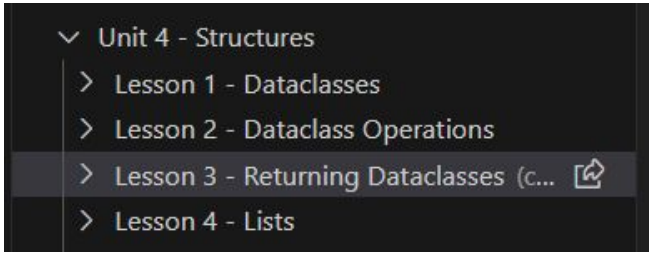
- ❑ Str
- ❑ Bool
- ✓ Fruit
- ❑ Int

Lab

Lab - GitHub Module

For the remaining lab time, break into your pod groups and complete the **VSCode Unit 4: Lesson 1 - Lesson 3**

When you complete this modules, work on your TLAB with your pod.



Wrap-Up

Lab (Due 03/28)

Taipei City, Taiwan

The company you work for, Seng-Links, aims to identify periods when a user sleeps or exercises using their varying recorded heart rates.

Your company has provided you a data folder (*data/*) of **4 files** that contain heart-rate samples from a participant. The participants device records heart rate data every 5 minutes (aka *sampling rate*).

You are tasked with writing code that **processes each data file**. You will utilize test-driven development in order to complete this project.

Announcements

A couple of assignment-related announcements:

- **Week 2 Pre-Class Quiz**
  - Cohort B: due 3/22
  - Cohort A: due 3/19
- **OOP Quiz (both cohorts due 3/22)**
- Fellows waiting on loaner laptops: please monitor Slack
- Office Hours are Open after class: Please bring questions!

Tuesday

Tomorrow will entail:

- Further exploration of OOP.

Jupyter: scratchpad of the data scientist

If you understand what you're doing, you're not learning anything. - Anonymous