Advanced Topics in Python

House Rules

Stop me whenever you want

No such thing as a silly question, and there is no such thing as a badly timed question

Stop underestimating your knowledge

Coding requires you to shift your thinking, not your understanding.

Stop worrying

This course is a whistle-stop tour of the topic, not a assessed lecture series.

Day 1 - Crash Course in Advanced Features

1. Patterns in design
2. Functions
3. Classes
4. Data Structures

You'll need:

IDE

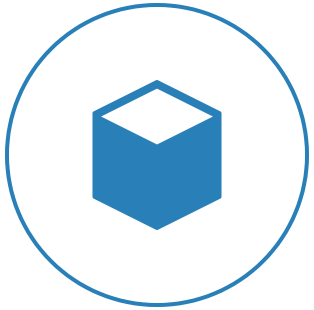
Spyder

PyCharm

REPL

Terminal

In IDE



Environment

Conda

PIP

Source Code Management

GitHub

Section 1

Code Patterns

Better, cleaner code

Some components adapted from Dan Bader: Python tricks

There are only 8 things you can do with code

Borrowed heavily from Andy Harris

New Variable

- **Name**

  What do we call this thing?

- **Type**

  What type of data does it contain

- **Initial Value**

  What is its starting value?

New Variable

- **Name**

  What do we call this thing?

- **Type**

  What type of data does it contain

- **Initial Value**

  What is its starting value?

"Create a variable called **name** of type **type** that starts with the value **initial value**"

New Variable

- **Name**

  What do we call this thing?

- **Type**

  What type of data does it contain

- **Initial Value**

  What is its starting value?

```
name = initial_value
```

"Create a variable called **name** of type **type** that starts with the value **initial value**"

Output

- **Message**

  Text to write to the user

print (“message”)

“Output the text **message**”

Input

- **Variable**

  Where the answer from the user will be stored

- **Message**

  Question being asked of the user

```
variable = input(“message”)
```

“Ask the user **message** and store the answer in **variable**”

Write a program in English only

Ask the user for two numbers and add them

Something like this?

Create a variable called **first_number** with the type integer with an initial value of None

Create a variable called **second_number** with the type integer with an initial value of None

Create a variable called **sum** with the type integer with an initial value of None

Ask the user “first number:” and put the answer in **first_number**

Ask the user “second number:” and put the answer in **second_number**

Put **first_number** + **second_number** in **sum**

Tell user “answer is **sum**”

Something like this?

Create a variable called **first_number** with the type integer with an initial value of None

Create a variable called **second_number** with the type integer with an initial value of None

Create a variable called **sum** with the type integer with an initial value of None

Ask the user “first number:” and put the answer in **first_number**

Ask the user “second number:” and put the answer in **second_number**

Put **first_number** + **second_number** in **sum**

Tell user “answer is **sum**”

You can google and check every step if you don't know how to do it

To python

```
first_number = int()
second_number = int()
sum = int()
first_number = input(“First number:”)
second_number = input(“Second number:”)
sum = first_number + second_number
print(“answer is {}”.format(sum))
```

Run it?

What went wrong?

How do we find out?

How do we test our theory?

Convert

- **Variable**

  Where the answer from the user will be stored

- **Message**

  Question being asked of the user

```
variable = input(“message”)
```

“Ask the user **message** and store the answer in **variable**”

For, While (or any control loop)

- **Sentry**

  Integer variable that will control loop

- **Start**

  Integer value of sentry at beginning

- **Finish**

  Integer value of sentry at end

- **Change code**

  code to change sentry so condition can be triggered

“Ask the user **message** and store the answer in **variable**”

For

- **Sentry**

  Integer variable that will control loop

- **Start**

  Integer value of sentry at beginning

- **Finish**

  Integer value of sentry at end

- **Change code**

  code to change sentry so condition can be triggered

```
for i in range(start, finish, step):
```

"begin with **sentry** at start. Change it with **change code** on each pass until **sentry** is larger or equal to **finish**"

While

- **Sentry**
  variable that will control the loop
- **Start code**
  starts the sentry
- **Condition**
  finish “test”
- **Change code**
  code to change sentry so condition can be triggered

```
<start code>
while(condition):
        change code
```

“Initialise **sentry** with **start code** then continue loop as long as **condition** is true. Change **sentry** with **change code**”

You spend 10 times as long reading code as you do writing it

Readability is as important as function

import this

BEAUTIFUL IS BETTER THAN UGLY.

EXPLICIT IS BETTER THAN IMPLICIT.

SIMPLE IS BETTER THAN COMPLEX.

COMPLEX IS BETTER THAN COMPLICATED.

FLAT IS BETTER THAN NESTED.

SPARSE IS BETTER THAN DENSE.

READABILITY COUNTS.

SPECIAL CASES AREN'T SPECIAL ENOUGH TO BREAK THE RULES. ALTHOUGH PRACTICALITY BEATS PURITY.

ERRORS SHOULD NEVER PASS SILENTLY. UNLESS EXPLICITLY SILENCED.

IN THE FACE OF AMBIGUITY, REFUSE THE TEMPTATION TO GUESS.

THERE SHOULD BE ONE-- AND PREFERABLY ONLY ONE --OBVIOUS WAY TO DO IT. ALTHOUGH THAT WAY MAY NOT BE OBVIOUS AT FIRST UNLESS YOU'RE DUTCH.

NOW IS BETTER THAN NEVER. ALTHOUGH NEVER IS OFTEN BETTER THAN *RIGHT* NOW.

IF THE IMPLEMENTATION IS HARD TO EXPLAIN, IT'S A BAD IDEA. IF THE IMPLEMENTATION IS EASY TO EXPLAIN, IT MAY BE A GOOD IDEA.

NAMESPACES ARE ONE HONKING GREAT IDEA -- LET'S DO MORE OF THOSE!

Use the REPL

help()

dir()

type()

([{

- (Parenthesis)
- [Bracket]
- {Brace}

When do we use parentheses?

- **instantiating a class**

  instance = ExampleClass()

- **calling the output of a function**

  output = example_function(input)

- **not using brackets allows us to interact with the class or function directly**

- **creating tuples**

  example_tuple = (“this”, “is a”, “tuple”)

- **creating generators**

  a = (x + 3 for x in example_list) #example_list = [2, 4, 5, 1]

- **Mathematical precedence**

  (a + b) * c

Brackets

- **to create lists**

  example_list = [1, 2, 6, “v”, 2]

- **to create list comprehensions**

  a = [x + 3 for x in example_list] #example_list = [2, 4, 5, 1]

- **selection and slicing from a collection**

  value = dict['key'] OR list[2] OR array[1:4]

Braces

- **to create a dict**

  example_dict = {“key”: value, “key: value”}

- **dict comprehension**

  example_dict = {x: x + 3 for x in example_list} #example_list = [2, 4, 5, 1]

- **to create sets**

  {a, b, d, a, d, c, a} -> {a, b, d, c}

- **string formatting**

  “example {}”.format(4) -> “example 4”

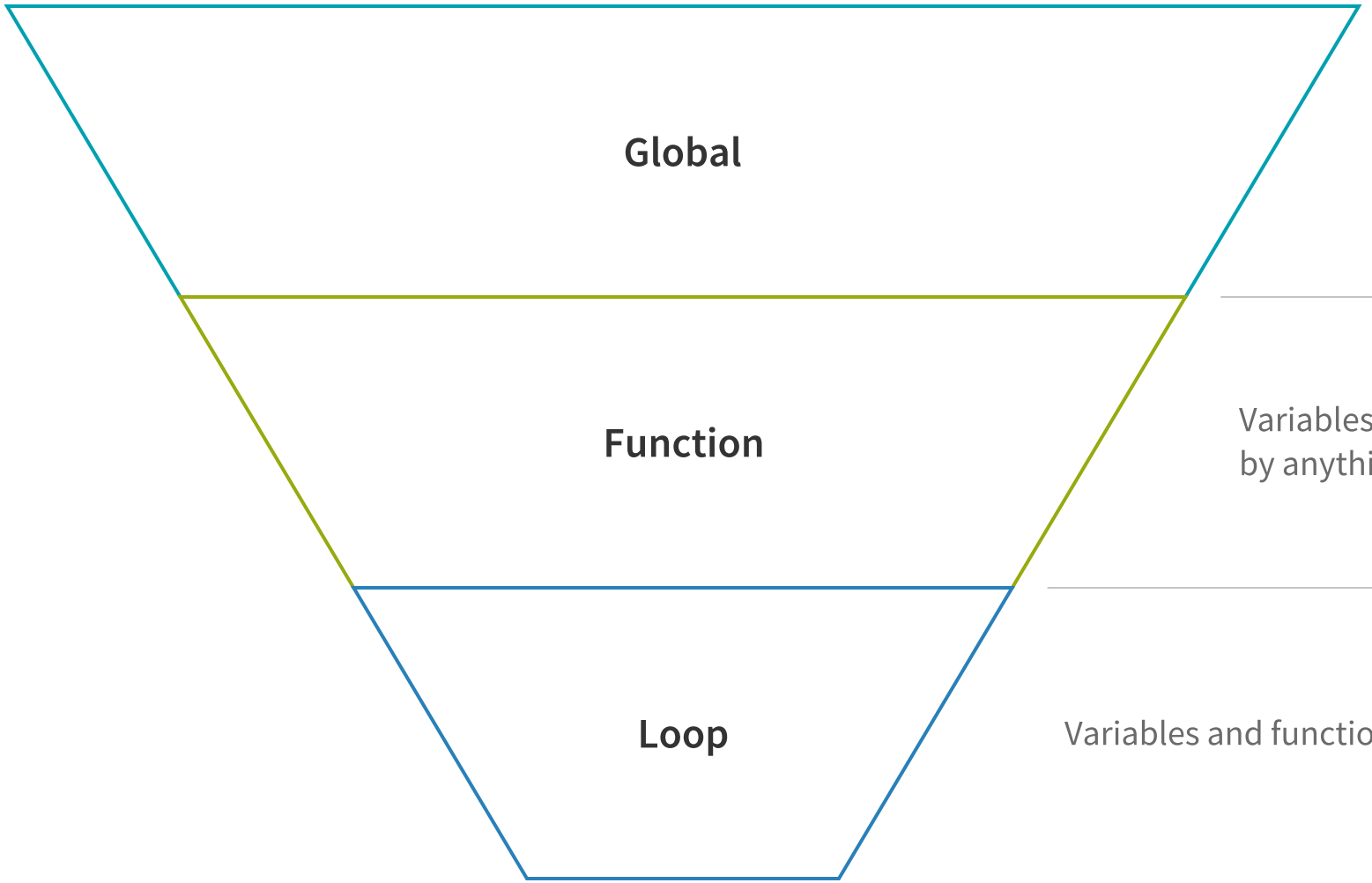
Cases

- variables_and_functions
- ClassesAndSubclasses
- test_cases

Avoid numbers in your names

Watch out for namespaces

local vs global

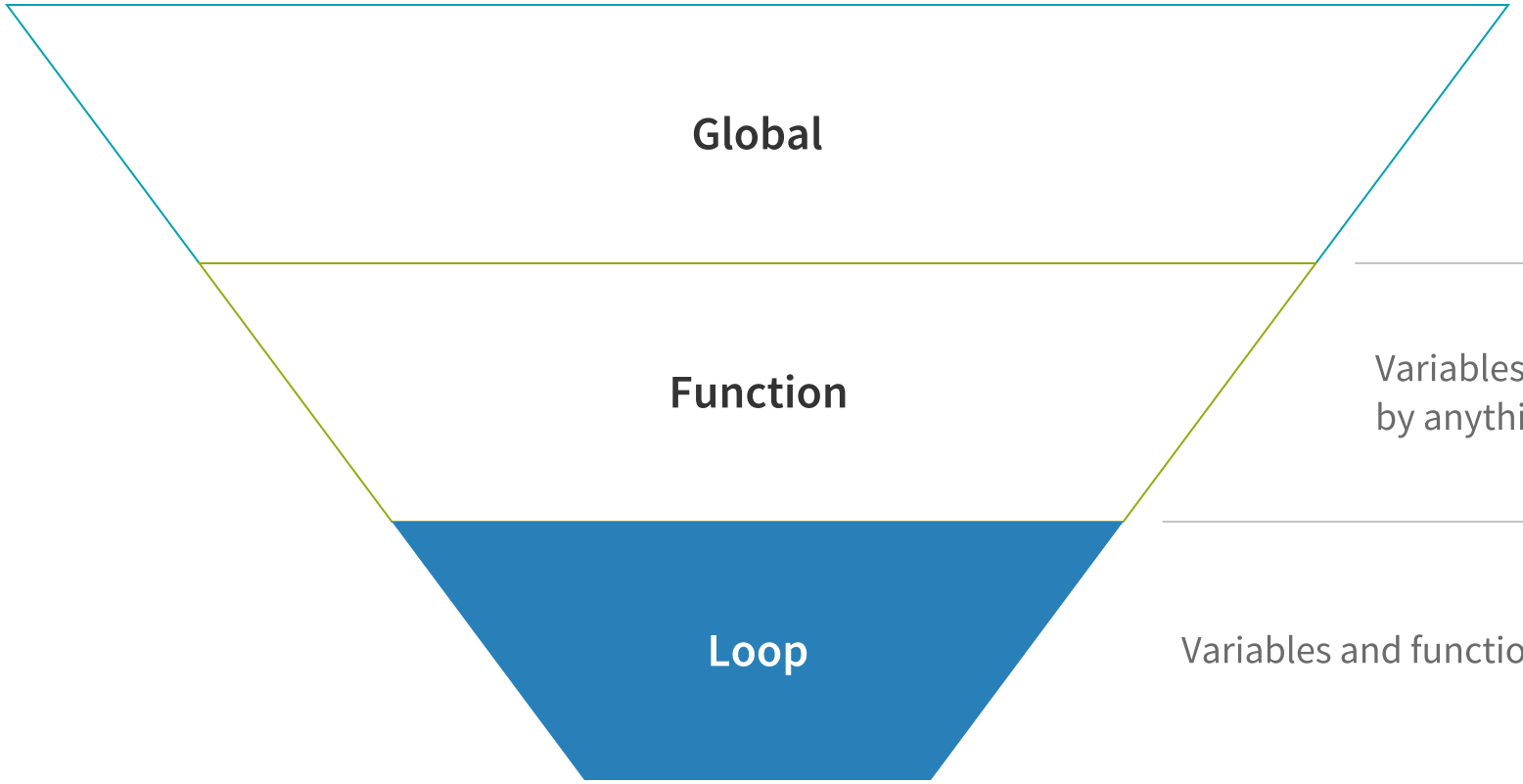


Variables and functions defined here can be accessed by anything in the file

Variables and functions defined here can be accessed by anything in the function only

Variables and functions defined here only exist in the loop

local vs global



Variables and functions defined here can be accessed by anything in the file

Variables and functions defined here can be accessed by anything in the function only

Variables and functions defined here only exist in the loop

Loop objects are local to a function level in Python


STOP

Friends don't let friends declare global variables

Underscores

For functions and classes
(“double underscores” == “dunders”)

- **_please_dont_touch_me**

  Unenforced hint that a variable is internal and not to be used. Not imported with * imports

- **my_name_is_already_taken_**

  Used if the variable name is already used else where: e.g. len_ ... not always needed

- **__im_private**

  “Mangles” the variable name so that it is hard to call externally and makes it private in practice

- **__im_special__**

  wrapped double underscores are for core reserved functions. Don't name your variables or functions like this

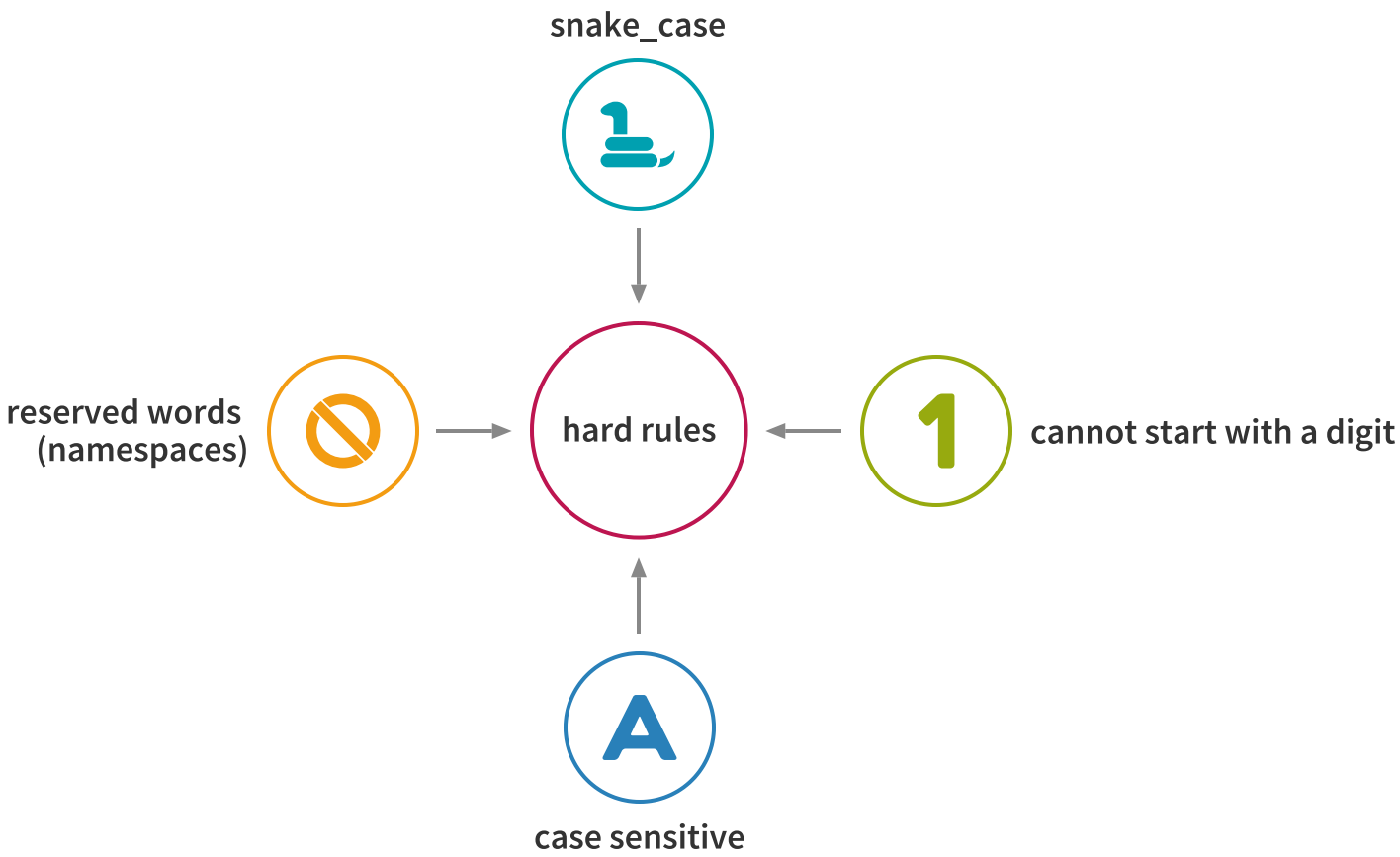
- **_**

  idiosyncratic. Means “unimportant variable” or “idc”.

variable naming matters

How do you name a variable?

Naming rules



reserved keywords

False None True and as assert break
class continue def del elif else except
finally for from global if import in
is lambda nonlocal not or pass
raise return try while with yield

Python won't stop you overwriting anything else

```
>>> example_int = 50
>>> example_int
50
>>> str(example_int)
'50'
>>> str = 80
>>> str(example_int)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'int' object is not callable
>>>
```

clean variable names

- **reveal intention**

  x = 4 OR number_of_days = 4

- **beware similar names**

  xyz_controller_for_efficient_handling_of_strings and now
  xyz_controller_for_efficient_storing_of_strings

- **Use pronounceable names**

  dt_rcrd_sytm_chk OR data_record_system_check

- **Classes should be nouns and singular**

  Measurement, Bear, Flower, Machine

- **Function names should have verbs**

  adder, cleaner, delete_older

clean variable names

- **one word per concept**

  compute a model or calculate a model?

- **names shouldn't rely on comments**

  If you need a comment to explain the variable's purpose, rethink it

- **long and clear is better than concise and unclear**

  persons vs expired_persons_list

- **avoid misleading names**

  e.g. expired_persons_list when it is actually an array

- **In some cases, avoid datatypes altogether**

  Good code will explicitly state the datatype. expired_persons > expired_persons_list

rules

- Don't hard code variables - use parameters
- Assume you'll need to reuse the code
- Assume you'll forget everything about your code by the next time you visit it
- Document your code
- shorter code isn't cleverer code

Comparisons

Is and ==

- **a == b**

  True

- **a is b**

  True

- **a == c**

  True

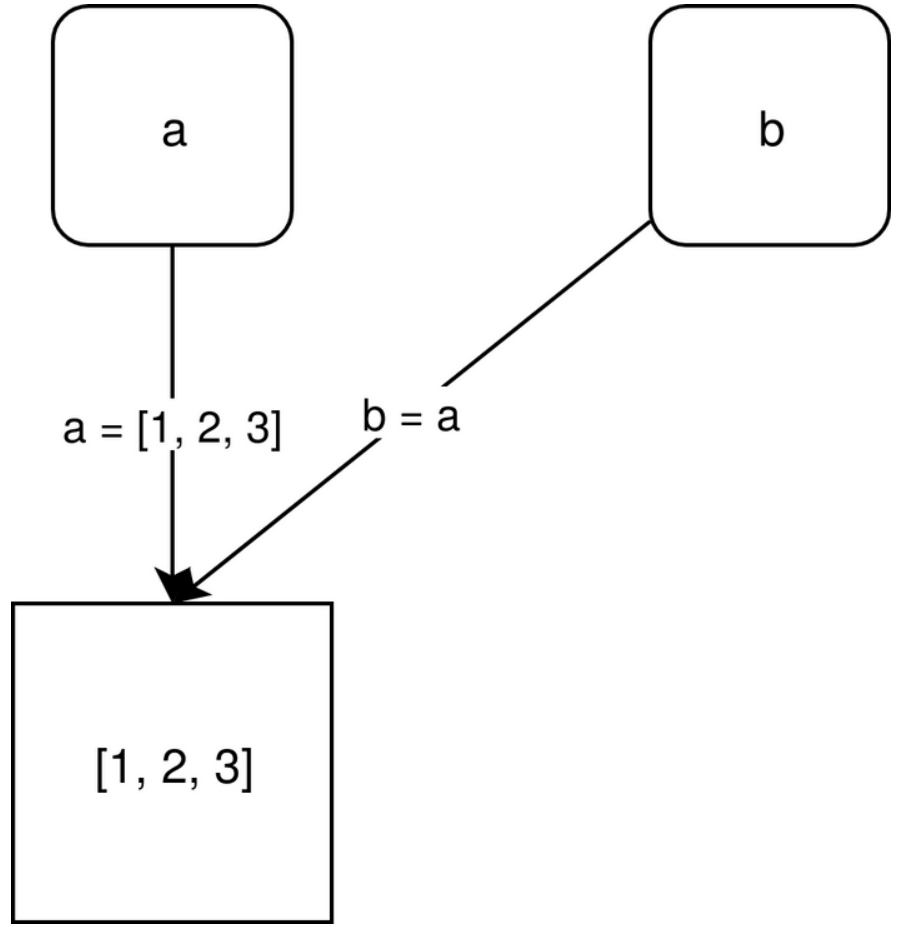
- **a is c**

  False

```
a = [1, 2, 3]
b = a
print(b) -> [1, 2, 3]
c = list(a)
print(c) -> [1, 2, 3]
```

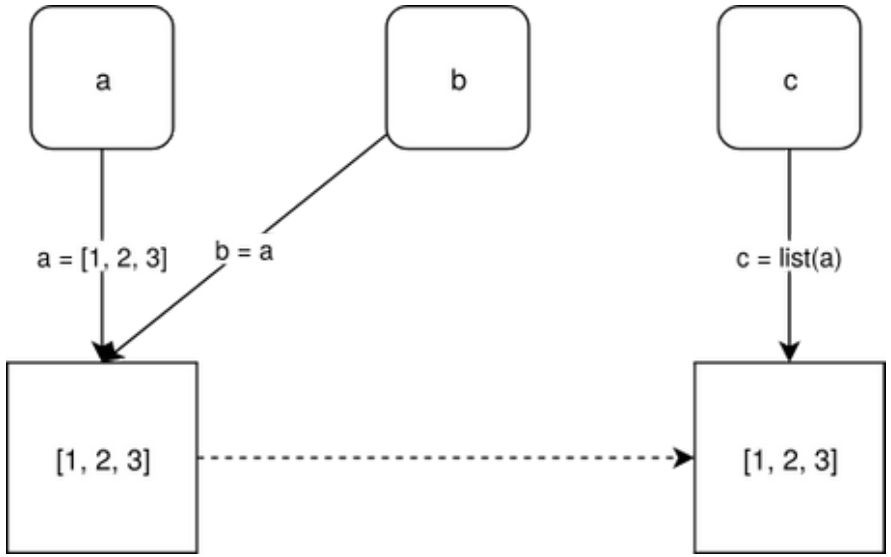
Pointers and memory

- a = [1, 2, 3]
- b = a



Pointers and memory

- a = [1, 2, 3]
- b = a
- c = list(a)



is and ==

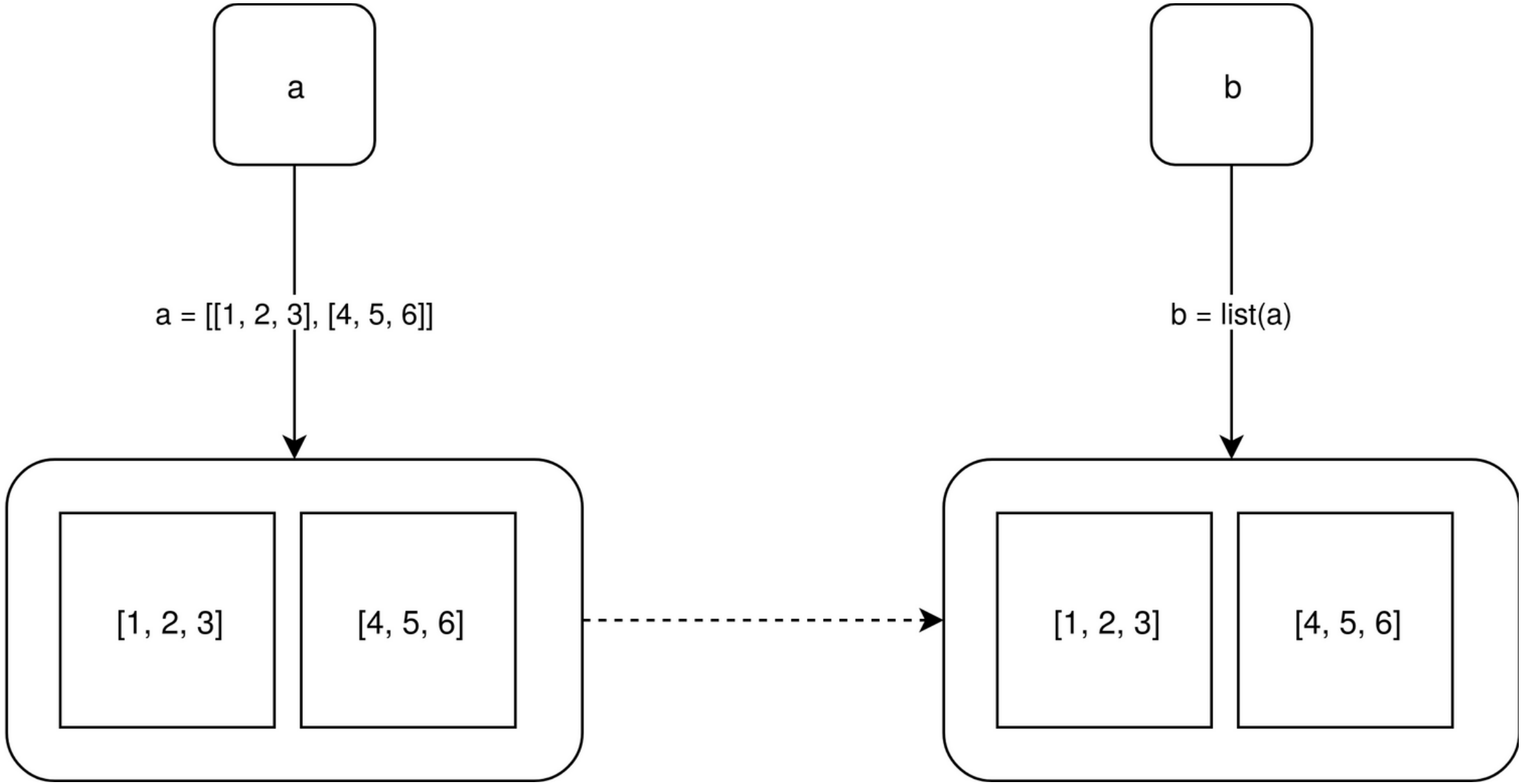
==

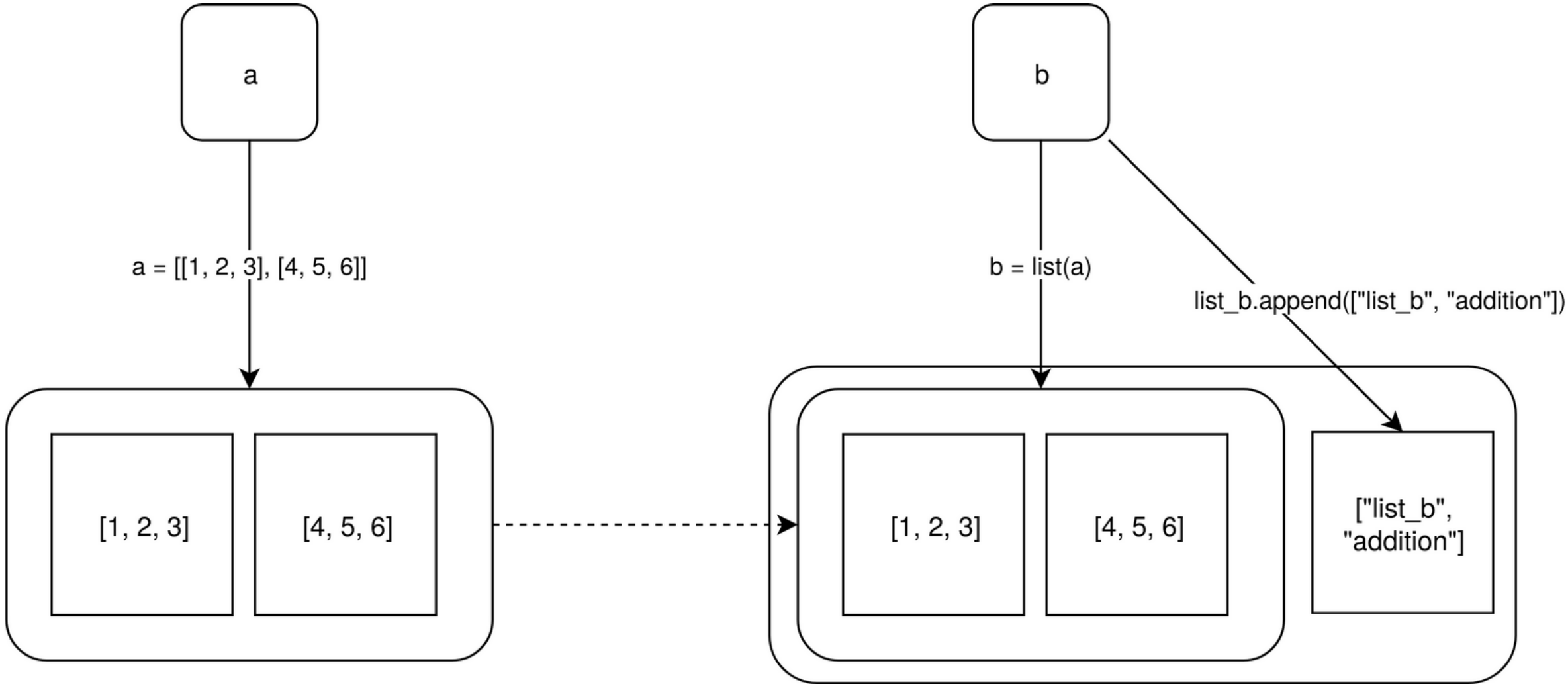
This checks that the values of the compared objects are the same

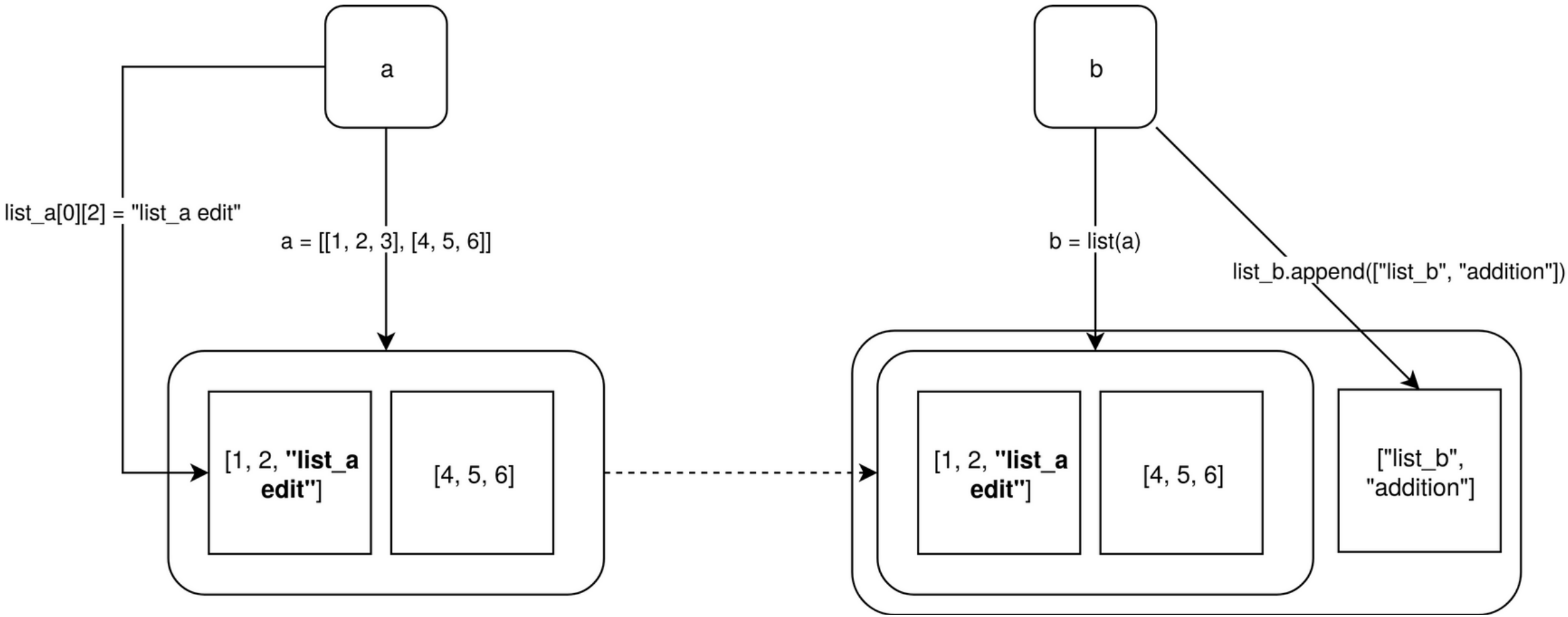
is

This checks that the compared objects are the same (reference the same memory)

```
>>> list_a = [[1, 2, 3], [4, 5, 6]]
>>> list_b = list(list_a)
>>> list_b.append(["list_b", "addition"])
>>> list_b
[[1, 2, 3], [4, 5, 6], ['list_b', 'addition']]
>>> list_a
[[1, 2, 3], [4, 5, 6]]
>>> list_a[0][2] = "list_a edit"
>>> list_a
[[1, 2, 'list_a edit'], [4, 5, 6]]
>>> list_b
[[1, 2, 'list_a edit'], [4, 5, 6], ['list_b', 'addition']]
```


a
a = [[1, 2, 3], [4, 5, 6]]
[1, 2, 3]
[4, 5, 6]
b
b = list(a)
[1, 2, 3]
[4, 5, 6]


a
a = [[1, 2, 3], [4, 5, 6]]
[1, 2, 3]
[4, 5, 6]
b
b = list(a)
list_b.append(["list_b", "addition"])
[1, 2, 3]
[4, 5, 6]
["list_b",
"addition"]


a
list_a[0][2] = "list_a edit"
a = [[1, 2, 3], [4, 5, 6]]
[1, 2, "list_a edit"]
[4, 5, 6]
b
b = list(a)
list_b.append(["list_b", "addition"])
[1, 2, "list_a edit"]
[4, 5, 6]
["list_b", "addition"]

shallow and deep copies

- Sometimes, the shallow pointer copies can cause problems
- You might change two objects by editing one
- For creating full copies of data, use “deep” copies

```
from copy import deepcopy

a = [1, 2, 3]

b = deepcopy(a)

b is a -> False
```

Assertions

- Checks values and statements
- Might be bypassed at run-time!
- raises an *AssertionError* on failure

```
def decrease_parameter(parameter, decrease_percentage):
    new_parameter_value = parameter["value"] * (1 - decrease_percentage)

    assert 0 <= new_parameter_value <= 1

    return new_parameter_value
```

Assertions

- Assertions should not be used in place of an if-else block
- Assertions inform users that an unrecoverable error has occurred
- Recoverable errors should be handled with try-except blocks

Context Managers

- Good for resource management - automatically closes files and objects when finished
- Error prone - even with an error, the file is closed
- Important for threading code

```
with open("example_file.txt", "w") as file:
    file.write("this is going into a file")
```

Common design patterns

design acronyms

- **DRY - Don't repeat yourself**

  If code is repeating, make it a function and call that instead. Try to only have one point in the code controlling that logic

- **YAGNI - You Ain't Gonna Need It**

  Write code to do what you need it to do now, instead of imagined requirements 6 months down the line. Code should be open to extension but closed to modification

- **KISS - Keep It Simple Stupid**

  Simpler solutions are often the better ones

- **SOC - Separation of Concerns**

  Every function, class and module should do ONE thing and one thing only.

Section 2

Functions

Standard functions

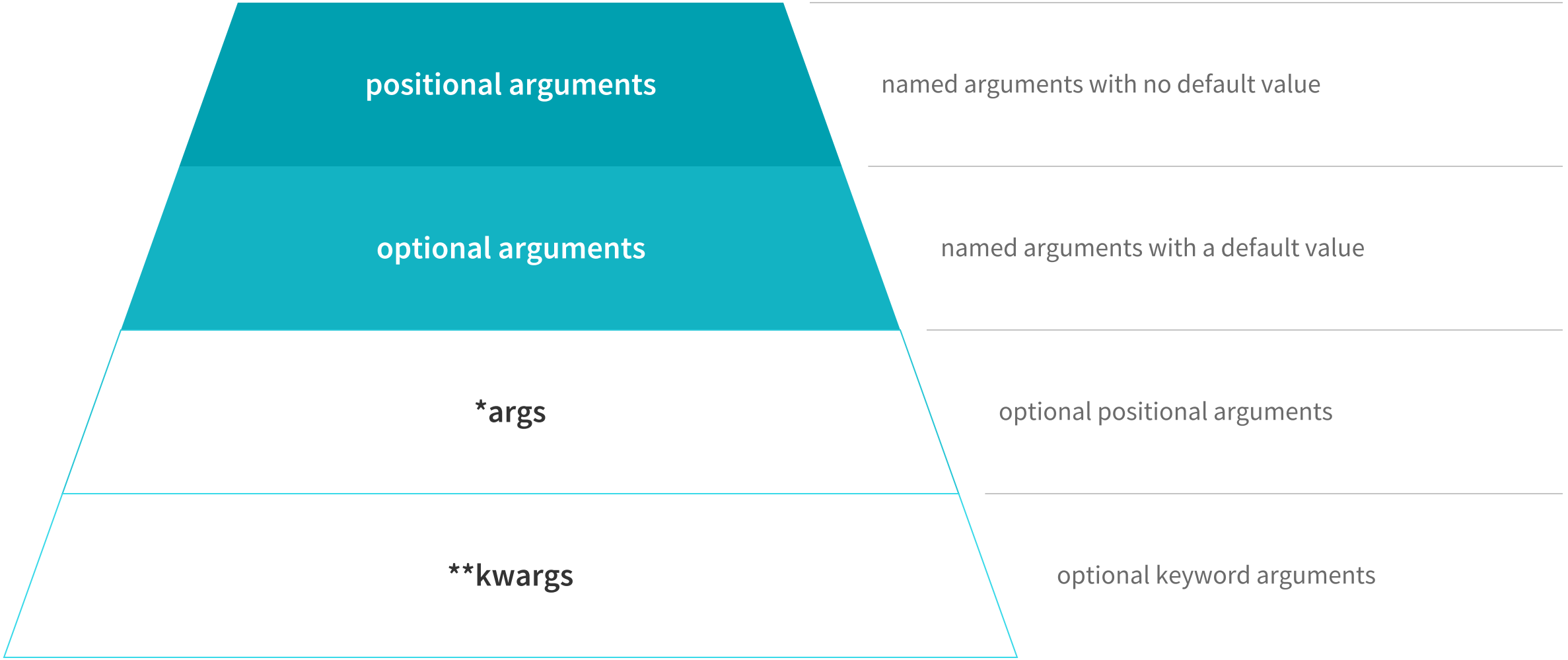
```
def functionname (arguments):

    do something with arguments

    return output
```

Standard functions

```
def adder (first_number, second_number=0):

    do something with arguments

    return output
```

```
>>> def adder(first_number, second_number=0):
...     return first_number + second_number
...
>>> adder(3)
3
>>> adder(3, 5)
8
>>> adder(second_number = 3, 2)
  File "<stdin>", line 1
SyntaxError: positional argument follows keyword argument
>>> adder(first_number=2)
2
>>>
```

Argument order



What are functions?

Practically?

First-Class Objects

- can be assigned to variables
- can be stored in data structures
- can be passed as arguments
- can be returned as values from a function

Python handles data this way

Python also handles functions this way

Functions as first-class objects

- can be assigned to variables
- can be stored in data structures
- can be passed as arguments
- can be returned as values from a function

```
def yell(text):
    return text.upper() + "!"
```

Functions as first-class objects

- can be assigned to variables
- can be stored in data structures
- can be passed as arguments
- can be returned as values from a function

```
def yell(text):
    return text.upper() + "!"


bark = yell

bark("hello")
```

you can delete yell and bark will still work

Functions as first-class objects

- can be assigned to variables
- can be stored in data structures
- can be passed as arguments
- can be returned as values from a function

```
>>> functions = [bark, str.lower, str.capitalize]
>>> for f in functions:
...     print(f, f("hey there"))
...
<function yell at 0x7f81cb903c80> HEY THERE!
<method 'lower' of 'str' objects> hey there
<method 'capitalize' of 'str' objects> Hey there
```

Functions as first-class objects

- can be assigned to variables
- can be stored in data structures
- can be passed as arguments
- can be returned as values from a function

```
>>> def make_adder(n):
...     def add(x):
...         return x + n
...     return add
...
>>> plus_3 = make_adder(3)
>>> plus_5 = make_adder(5)
>>> plus_3(4)
7
>>> plus_5(4)
9
```

Everything in Python is an object

OBJECTS CAN BE ASSIGNED TO A VARIABLE

OBJECTS CAN BE STORED IN A DATA STRUCTURE

OBJECTS CAN BE PASSED AS ARGUMENTS

OBJECTS CAN BE RETURNED FROM FUNCTIONS

Lambdas

- **lambdas are small anonymous functions**

  can be assigned a name, and restricted to one line

- **cannot use statements or annotations**

- **health warning!**

  try to only use unnamed lambdas, and use them briefly

```
>>> add = lambda x, y: x + y
>>> add(5, 3)
8
```

Why use lambdas at all?

- They seem to fly in the face of Pythonic code - concise, hard to read and non-intuitive
- Vital for functional programming
- Powerful for mapping
- Opt for functions if you can

Exercises

Challenge

“Richard”, “Strange”, 1995 ----> [(“Richard Strange”, 1995), “Richard Strange, born 1995”, “R.S. 1995”, “R. Strange 1995”]

- **Inputs:**

  first and last name, birth year

- **Outputs:**

  tuple of merged first and last name, and year; string of names and year; string of initials and year

- **Requirements:**

  Use assertions to check types

  remember SOC

  use a lambda for the fourth output

Section 3

Classes

Create three dogs, that have fur, four legs and can bark

```python
def bark(dog_name: str) -> None:
    print("{} says 'WOOF!'".format(str(dog_name)))

fido_legs = 4
spot_legs = 4
lassie_legs = 4


fido_has_fur = True
spot_has_fur = True
lassie_has_fur = True

bark("fido")
bark("spot")
bark("lassie")
```

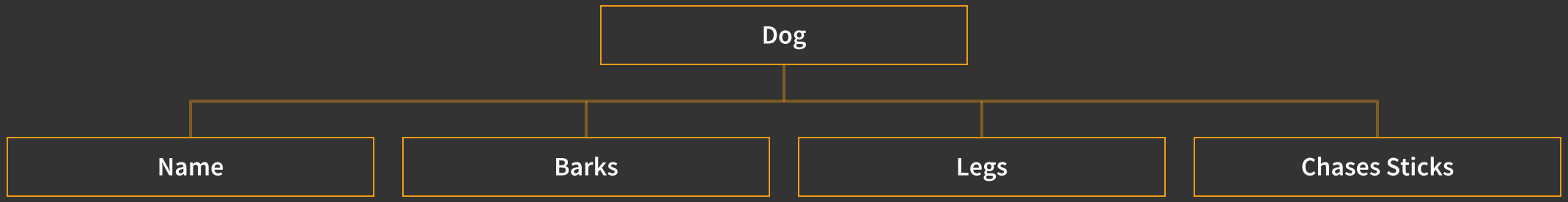
```
fido says 'WOOF!'
spot says 'WOOF!'
lassie says 'WOOF!'

Process finished with exit code 0
```

add the attributes “chases sticks”, and add 3 more dogs

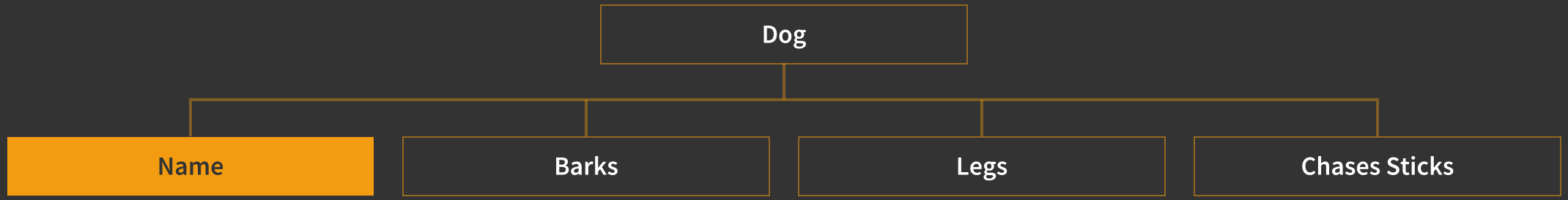
Repetition in your code

Is there better way?

Dogs



Dogs

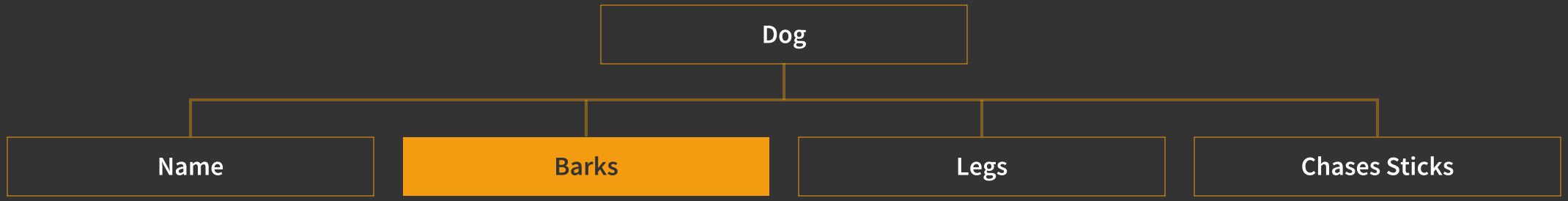


We only expect the name to vary every time

```
1   class Dog:
2
3       def __init__(self):
4           self.number_of_legs = 4
5           self.chases_sticks = True
6
```

```python
class Dog:

    def __init__(self, name):
        self.number_of_legs = 4
        self.chases_sticks = True
        self.name = name

```

Dogs



This has function-like behaviour

```
1   class Dog:
2
3       def __init__(self, name):
4           self.number_of_legs = 4
5           self.chases_sticks = True
6           self.name = name
7
8       def barks(self):
9           print("{} says 'WOOF!".format(self.name))
10
```

```
>>> from examples.oop_dogs import Dog
>>> fido = Dog("fido")
>>> spot = Dog("spot")
>>> lassie = Dog("lassie")
>>> fido.barks()
fido says 'WOOF!
>>> spot.number_of_legs
4
```

Classes

- A class is a pattern of logic for describing an object
- *instances* of a class can be created, which are independent
- variables (attributes) and functions (methods) that belong to an instance can be accessed with dot-notation

__init__

- Reserved function/method for classes. This function runs when an instance is first created - or “initialised”.
- Used to define all of the input parameters
- It is good practice to define all your variables here, even if they are null to begin with

self

- **self is used in functions and variables inside a class**
- **It refers to the class itself**

  self.[function] points to an internal method

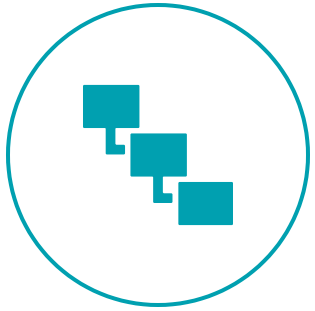
  self.[variable] points to an internal attribute

- **These structures are passed on to an instance you create**

  This allows us to do fido.barks() and spot.barks() even though the method name “barks” is the same.

OOP (Object Oriented Programming) is a different way to approach code

Coding Paradigms



Procedural

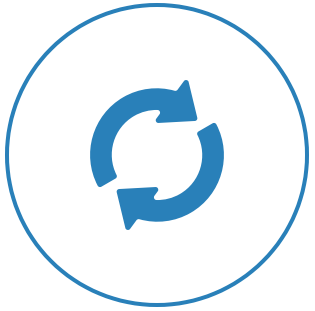
Run from point A to point B

Use defined functions to better handle it's code

Object Oriented

Recreate the logic of a real-world object in code

Mimicks its behaviour to achieve an output



Functional

Functions can change other functions, and we can use these to change how we act on an event or data

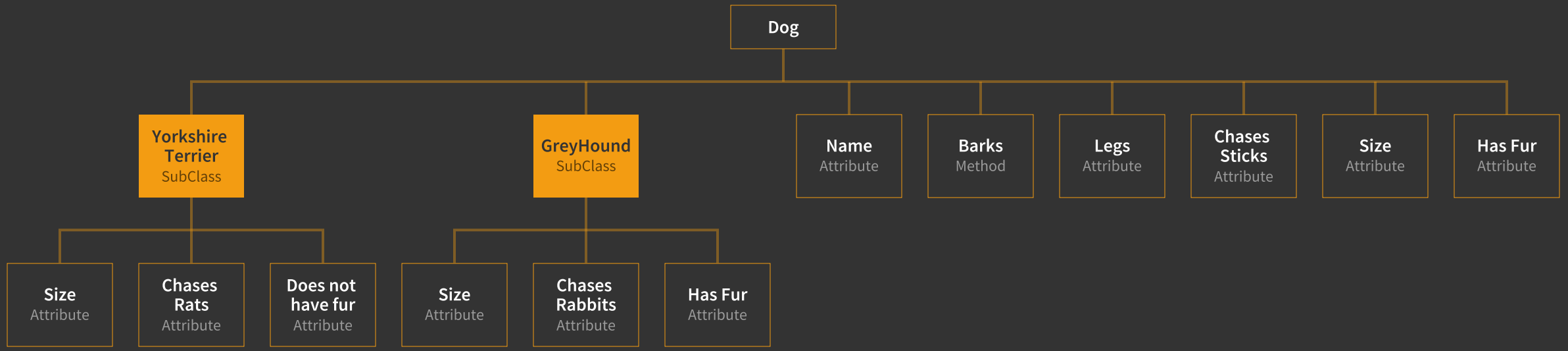
Subclasses

- Classes can exist inside other classes
- These are “subclasses” to their “superclasses”
- Also referred to as child and parent classes

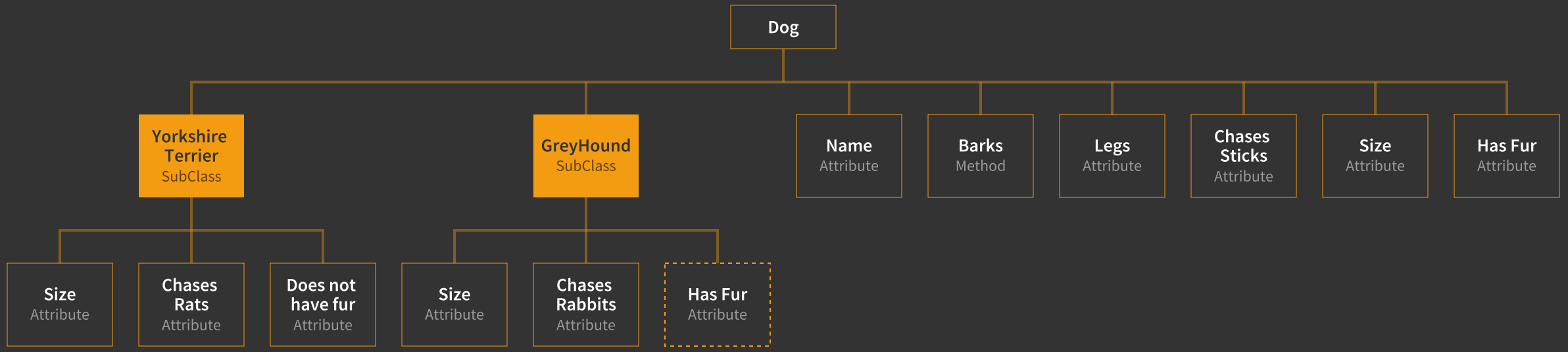
Greyhound - larger, fast, chases rabbits

Yorkshire Terrier - smaller, yaps, chases rats

Dogs


Dog
Yorkshire Terrier
SubClass
GreyHound
SubClass
Name
Attribute
Barks
Method
Legs
Attribute
Chases Sticks
Attribute
Size
Attribute
Has Fur
Attribute
Size
Attribute
Chases Rats
Attribute
Does not have fur
Attribute
Size
Attribute
Chases Rabbits
Attribute
Has Fur
Attribute

Dogs



“Greyhound” has fur like “Dog” - redundant.

```
class Dog:

    def __init__(self, name, breed="Unknown"):
        self.number_of_legs = 4
        self.chases_sticks = True
        self.breed = breed
        self.name = name
        self.has_fur = True
        self.size = "Unknown"

    def barks(self):
        print("{} says 'WOOF!".format(self.name))


class YorkshireTerrier(Dog):

    def __init__(self, name):
        super().__init__(name=name, breed="Yorkshire Terrier")
        self.chases_rats = True
        self.has_fur = False
        self.size = "Small"
```

```
>>> from examples.oop_dogs import *
>>> teacup = YorkshireTerrier("teacup")
>>> teacup.size
'Small'
>>> teacup.barks()
teacup says 'WOOF!
>>> teacup.chases_sticks
True
>>> teacup.chases_rats
True
>>> teacup.chases_rabbits
Traceback (most recent call last):
  File "<input>", line 1, in <module>
AttributeError: 'YorkshireTerrier' object has no attribute 'chases_rabbits'
```

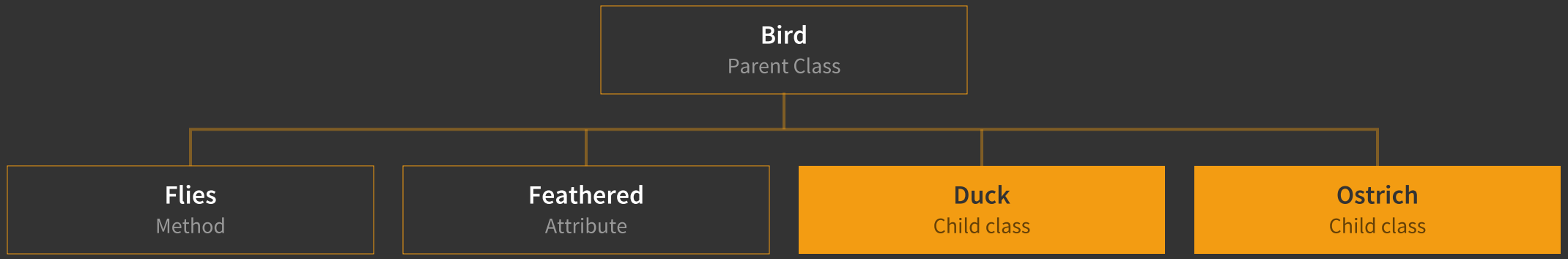
Add the greyhound child class

Has fur, is medium sized, and chases rabbits

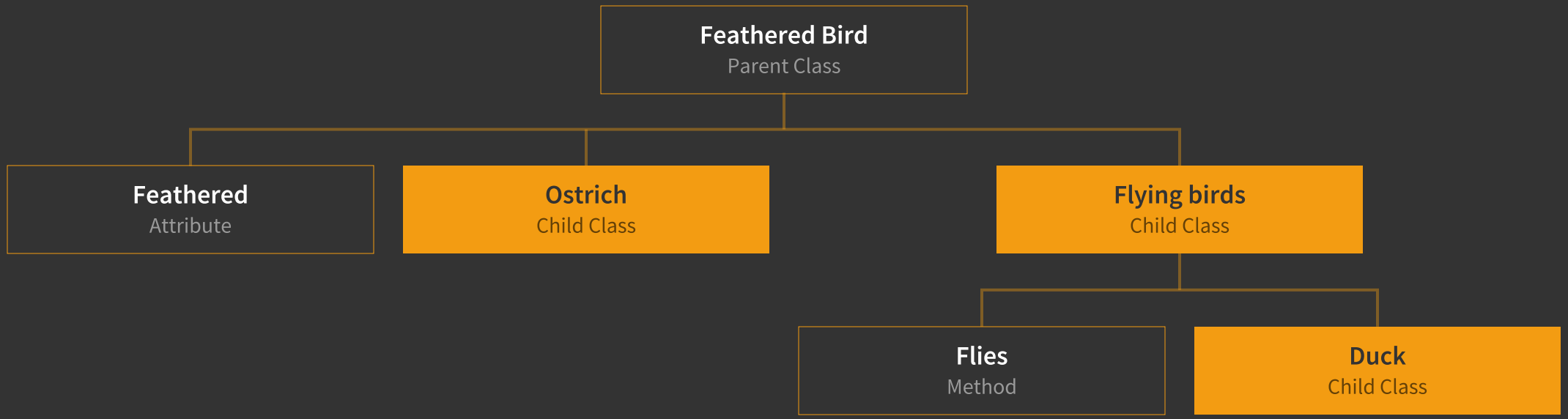
Liskov Substitution Principle

- A child class should be able to be used anywhere you have a parent class
- consider ostriches and ducks...

LSP


Bird
Parent Class
Flies
Method
Feathered
Attribute
Duck
Child class
Ostrich
Child class

LSP


Feathered Bird
Parent Class
Feathered
Attribute
Ostrich
Child Class
Flying birds
Child Class
Flies
Method
Duck
Child Class

__str__ and __repr__

- objects have representations and may have string functions
- By default, the representation is the object memory address
- Many base python objects have string methods

```
>>> class ExampleClass:
...     def __init__(self):
...         pass
...
>>> example_instance = ExampleClass()
>>> example_instance
<__main__.ExampleClass object at 0x7fe7f47cb6d8>
```

```python
class MyCar:
    def __init__(self, mileage, colour):
        self.mileage = mileage
        self.colour = colour

    def __str__(self):
        return "a {} car".format(self.colour)

    def __repr__(self):
        return "car: Mileage {}, Colour {}".format(self.mileage, self.colour)
```

```
>>> from examples.repr_and_str import MyCar
>>> a_car = MyCar(mileage=3000, colour="red")
>>> a_car
car: Mileage 3000, Colour red
>>> str(a_car)
'a red car'
>>> print(a_car)
a red car
>>> [a_car]
[car: Mileage 3000, Colour red]
```

Str and Repr usage

String

__str__

returned when printed or called with str()

Output should be **readable**

Representation

__repr__

returned when an object is inspected

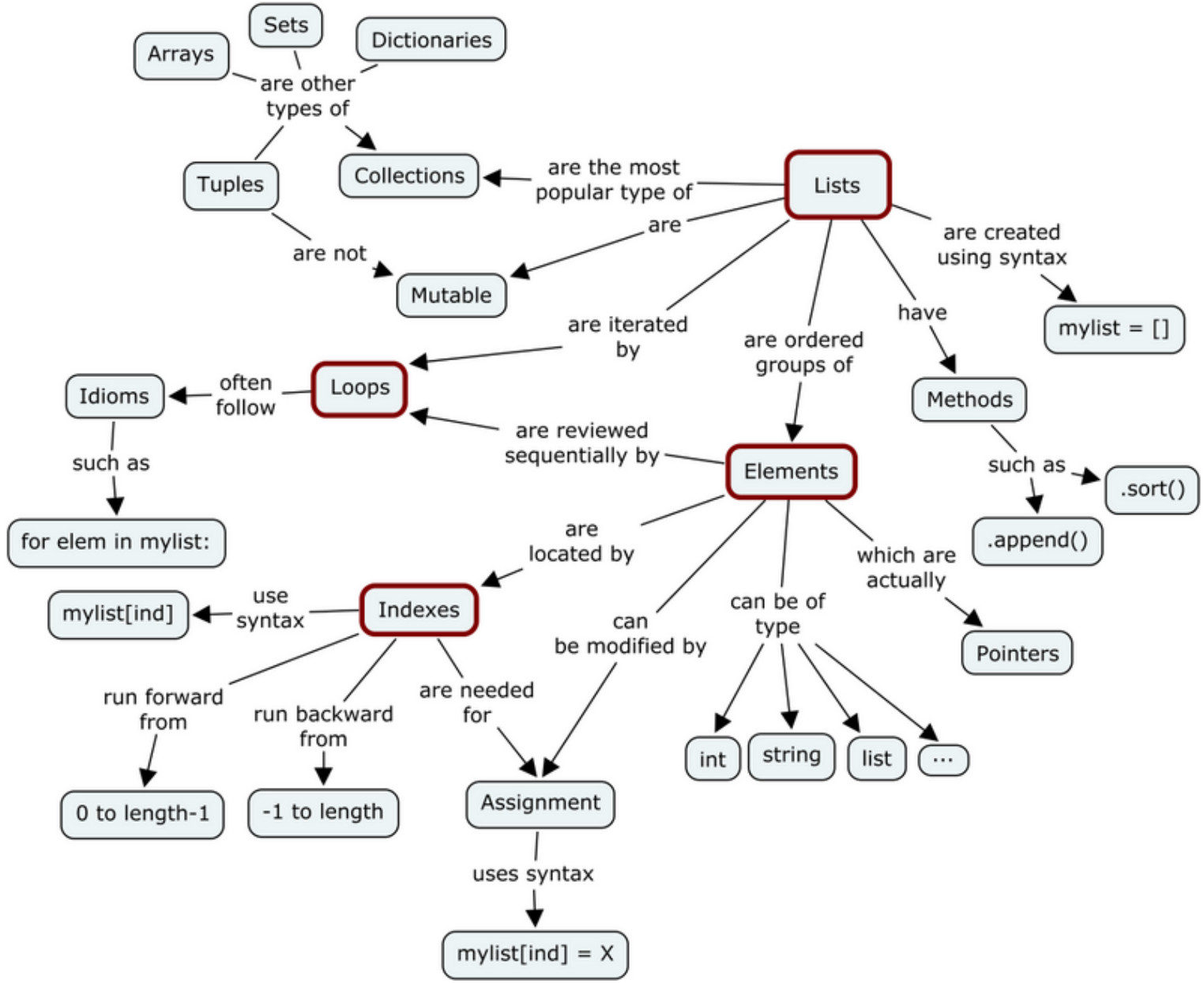
returns when called with repr()

Output should be **unambiguous**

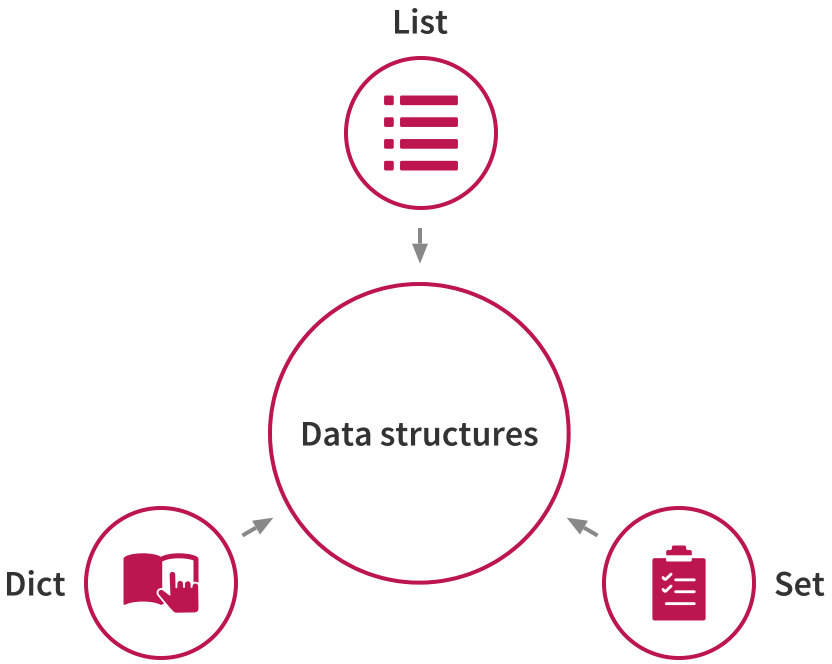
An object without a __str__ method will use the __repr__ output

Section 5

Data Structures


Sets
Arrays
Dictionaries
are other types of
Tuples
Collections
are the most popular type of
Lists
are
are not
Mutable
are created using syntax
mylist = []
are iterated by
are ordered groups of
have
Loops
Idioms
often follow
Methods
such as
.sort()
.append()
are reviewed sequentially by
Elements
such as
for elem in mylist:
are located by
which are actually
Pointers
mylist[ind]
use syntax
Indexes
can be modified by
can be of type
run forward from
run backward from
are needed for
int
string
list
...
0 to length-1
-1 to length
Assignment
uses syntax
mylist[ind] = X

Standard data structures



Dicts

aka Dictionaries

- key:value pairing
- keys must be unique
- can be nested: {k:([k:v, k:v, k:v]), k:v}
- Performant, safe, well supported
- Keys MUST be immutable

```
>>> my_dict = {"alex": 45, "jane": 30, "saoirse": 23}
>>> my_dict["alex"]
45
```

Alternatives

using “import collections”

- **Ordered Dicts**

  Remember the insertion order of keys

  collections.OrderedDict

- **DefaultDict**

  Creates a default value for missing values by specifying a callable

  collections.defaultdict

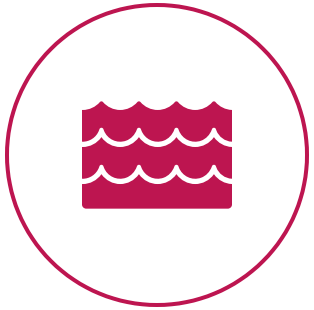
Arrays

aka Lists

- series of data stored in continguous memory blocks
- Allow elements to be efficiently located based on their position (index) in the array
- Not all arrays are the same...

Mutability

from “mutation”



Mutable

After being created with an initial value, can change it's value

Immutable

Is frozen to the initial value it is created with

Lists

[list], list(iterable)

- **Mutable**

  Its contents can change after creation

- **Arbitrary array**

  Can hold any kind of element or object

- **Not memory optimised**

  As it can hold anything, its memory blocks are less tightly packed

Tuples

(element, element), tuple()

- **Immutable**

  Frozen to its initial values

- **Arbitrary array**

  Can hold any kind of element or object

- **Not memory optimised**

  As it can hold anything, its memory blocks are less tightly packed

Array

array.array('datatype', (element, element))

- **Immutable**

  Frozen to its initial values

- **Typed**

  Can only hold one type of data at a time

- **Memory optimised**

  As each element holds a fixed size in memory, it can be packed efficiently

Type code	C Type	Python Type	Minimum size in bytes
`'b'`	signed char	int	1
`'B'`	unsigned char	int	1
`'u'`	Py_UNICODE	Unicode character	2
`'h'`	signed short	int	2
`'H'`	unsigned short	int	2
`'i'`	signed int	int	2
`'I'`	unsigned int	int	2
`'l'`	signed long	int	4
`'L'`	unsigned long	int	4
`'q'`	signed long long	int	8
`'Q'`	unsigned long long	int	8
`'f'`	float	float	4
`'d'`	double	float	8

String

“elements”, str(element)

- **Immutable**

  Frozen to its initial values

- **Typed - unicode**

  Can only hold unicode characters

- **Memory optimised**

  As each element holds a fixed size in memory, it can be packed efficiently

```
>>> my_string[3] = "e"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

Bytes

bytes(element, element)

- **Immutable**

  Frozen to its initial values

- **Typed - bytea**

  Can only hold byte elements

- **Extremely memory optimised**

  Byte-level data allows very efficient stacking

Bytearray

bytearray(element, element)

- **Mutable**

  Elements can be edited, and the size of the array will grow and shrink

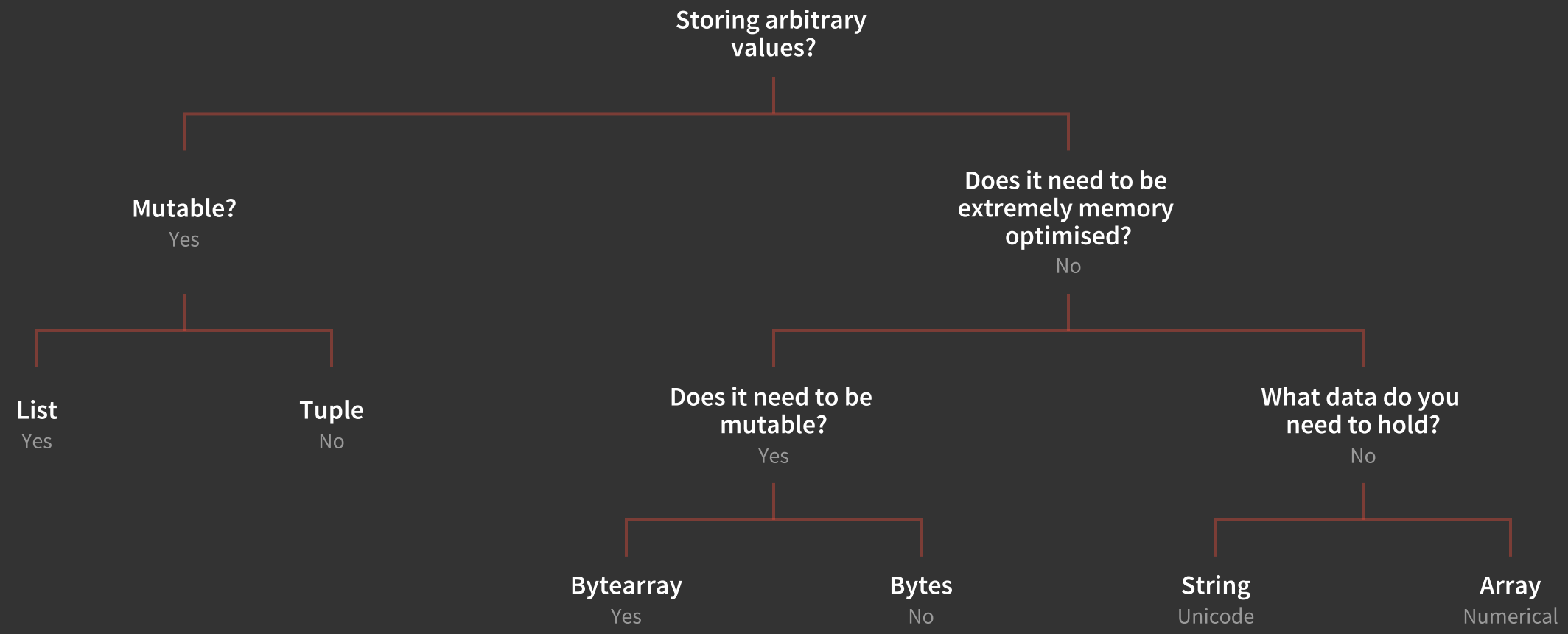
- **Typed - bytea**

  Can only hold byte elements

- **Extremely memory optimised**

  Byte-level data allows very efficient stacking

Which array?


Storing arbitrary values?
Mutable?
Yes
Does it need to be extremely memory optimised?
No
List
Yes
Tuple
No
Does it need to be mutable?
Yes
What data do you need to hold?
No
Bytearray
Yes
Bytes
No
String
Unicode
Array
Numerical

Data Objects and structures

- data structures provide fixed fields for information
- May be named
- May be typed, or internally typed

Sets

set()

- An unordered collection of objects
- Does not allow duplicate elements
- Ideal for deduplication of data
- Performant for merging datasets

Set

set()

- **Mutable**

  dynamic insertion and deletion of elements

- **Only ingests new elements**

  already existing elements are quietly skipped

- **Behaves like a dict behind the scenes**

  Similar performance. However, does not use key value pairs

Frozen set

frozenset()

- **Immutable**

  Elements can only be added at creation

- **Powerful for filtering**

  Hard data rules and filters can be used safely with frozen sets e.g. **is a in set**

- **Ideal for storing dict keys**

  dict keys have to be immutable - sets won't work

Counter

from collections import Counter,
a = Counter(), a.update(elements)

- **Mutable**

  Elements can be added and removed at any point

- **Counts how many times a value appears**

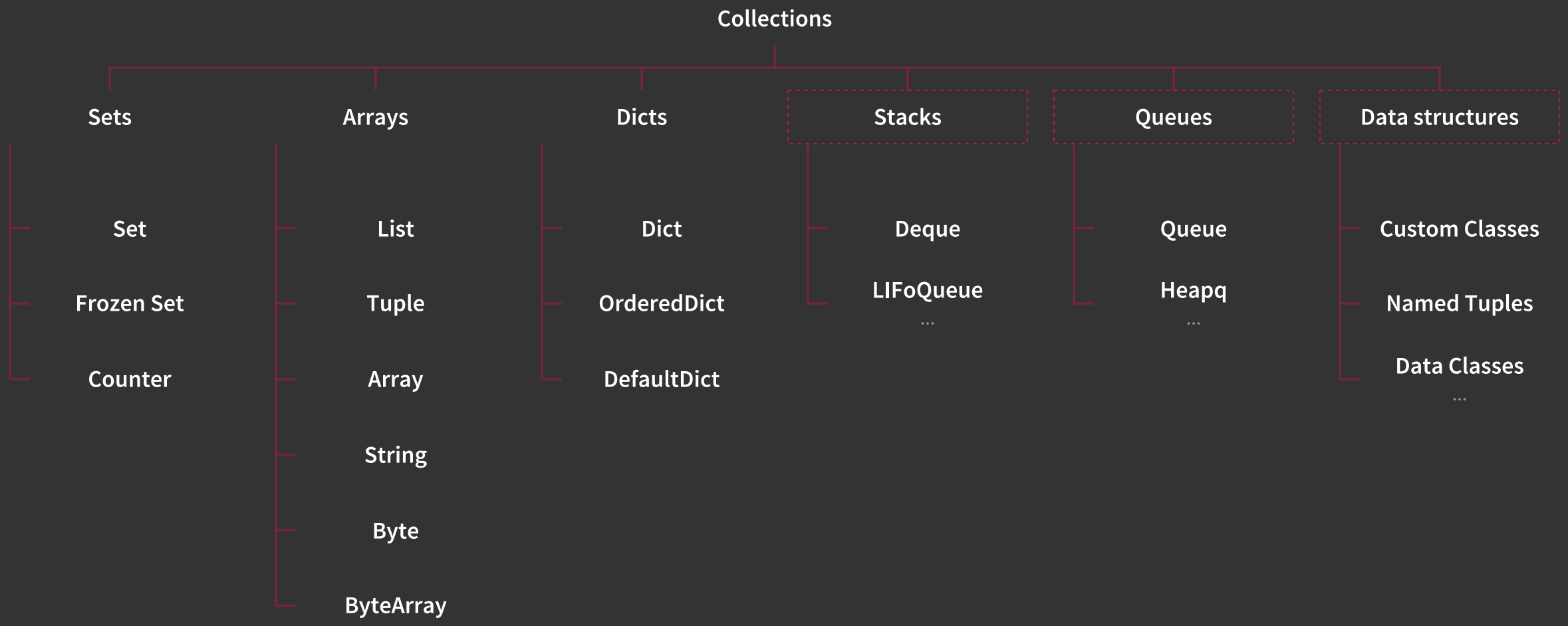
  provides a unique list of values, and their frequency

- **len(a) gives the number of unique elements.**

  sum(a) gives the total number of counted values

Collections

Collections



What we covered

- import this
- ()[]{}
- Cases
- local & global
- variable naming
- is & ==
- copying
- assertions
- context managers
- DRY, YAGNI, KISS, SOC
- Function arguments
- first class objects
- lambdas
- classes
- attributes
- methods
- self
- __init__
- __str__ & __repr__
- Liskov Substitution Principle
- Dicts
- Sets
- Arrays

Advanced Topics in Python

Day 2

Day 2 content

1. TDD
2. Git
3. Project

Section 1

Comments

OR HOW NOT TO BE AN ARSEHOLE TO FUTURE YOU

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!

Comment syntax

- **one liners are preceded with a #**
- **Multiline comments are wrapped with three quotes**

  “”“ or ''' comments ''' or ”“”

Good code

- Bad code is undocumented code
- Better code has comments
- The best code has documentation

```python
class CommentedDog:

    def __init__(self, name, breed="Unknown"):  # take the name from user and set unknown breed
        self.number_of_legs = 4  # a dog has 4 legs
        self.chases_sticks = True  # set the dog to like chasing sticks
        self.breed = breed  # set the breed
        self.name = name  # set the user-supplied name
        self.has_fur = True  # set the dog to have fur
        self.size = "Unknown"  # set the size to be unknown
```

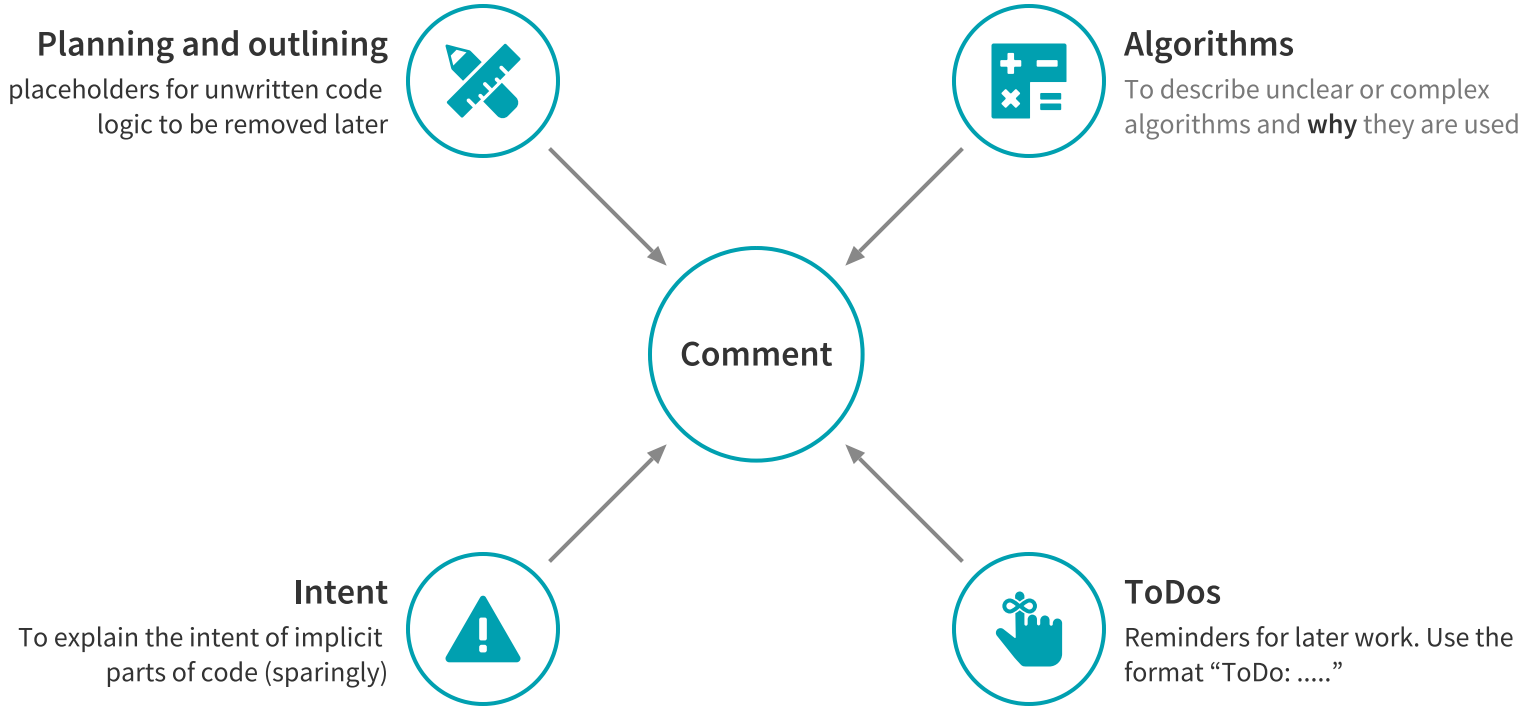
```python
class CommentedDog:

    def __init__(self, name, breed="Unknown"):  # take the name from user and set unknown breed
        self.number_of_legs = 4  # a dog has 4 legs
        self.chases_sticks = True  # set the dog to like chasing sticks
        self.breed = breed  # set the breed
        self.name = name  # set the user-supplied name
        self.has_fur = True  # set the dog to have fur
        self.size = "Unknown"  # set the size to be unknown
```

Clean simple code will already tell us how we're handling our logic
These comments are redundant

```python
class CommentedYorkshireTerrier(CommentedDog):

    def __init__(self, name):
        super().__init__(name=name, breed="Yorkshire Terrier")  # need to provide a name to dog superclass
        self.chases_rats = True
        self.has_fur = False  # overwrites superclass attribute
        self.size = "Small"  # overwrites superclass attribute
```

These comments are better, but not always neccessary. Save # comments for unavoidably unreadable lines of code

General comment usage



DocStrings

- Multiline comments just under a class or function definition are interpreted differently by Python
- These Documentation Strings or DocStrings are read by IDEs, and when you call help(object)
- Can be automatically read to generate documentations

  See Sphinx and reStructuredText

- Can be accessed even when someone doesn't have your code open

Settings | Tools | Python Integrated Tools | Docstring format > Google

```python
class DocumentedDog:
    """
    the DocumentedDog object holds various breeds of dog.

    Args:
        name (str): The name of the dog
        breed (str): The breed of the dog. Default to "Unknown".

    Attributes:
        number_of_legs (int): The number of legs a dog has.
        chases_sticks (bool): If a dog likes to chase after sticks.
        breed (str): The name of the breed of dog.
        name (str): The name of the breed of dog.
        has_fur (bool): If a dog has fur or not.
        size (str): The qualitative size of the dog.
    """
    def __init__(self, name, breed="Unknown"):
        self.number_of_legs = 4
        self.chases_sticks = True
        self.breed = breed
        self.name = name
        self.has_fur = True
        self.size = "Unknown"

    def barks(self):
        """
        We are told that a dog is barking
        Returns:
            None
        """
        print("{} says 'WOOF!'".format(self.name))
```

DocString Template

```
def my_function(args):
'''
one line summary description

longer, extended description of
the function purpose and intent if required

args:
    argument_name (dtype): what and why

returns:
    describe what is returned and why. Be verbose.

Raises:
    list all exceptions the function explicitly raises

'''
```

docstring usage

- Classes will also need an “attribute” section
- Simple short functions can be summarised with a one line summary with ''' ... '''
- If a function raises nothing, omit the section
- If a function returns nothing, or the return is simple and described in the summary, omit the section

Your code should clearly say how

Your comments should clearly say why

One additional way to hint to your code function...

Type Annotations

- New with Python 3.5
- not strict
- Allows tools like IDEs and PyLint to understand your code better

  e.g. the type error highlighted for “breed”

```
class DocumentedDog:
    """
    the DocumentedDog object holds various breeds of dog.

    Args:
        name (str): The name of the dog
        breed (str): The breed of the dog. Default to "Unknown".

    Attributes:
        number_of_legs (int): The number of legs a dog has.
        chases_sticks (bool): If a dog likes to chase after sticks.
        breed (str): The name of the breed of dog.
        name (str): The name of the breed of dog.
        has_fur (bool): If a dog has fur or not.
        size (str): The qualitative size of the dog.
    """
    def __init__(self, name: str, breed: str = "Unknown"):
        self.number_of_legs: int = 4
        self.chases_sticks: bool = True
        self.breed: bool = breed
        self.name: str = name
        self.has_fur: bool = True
        self.size: str = "Unknown"

    def barks(self) -> None:
        """
        We are told that a dog is barking
        Returns:
            None
        """
        print("{} says 'WOOF!'".format(self.name))
```

Exercise

revisit your function code - comment it

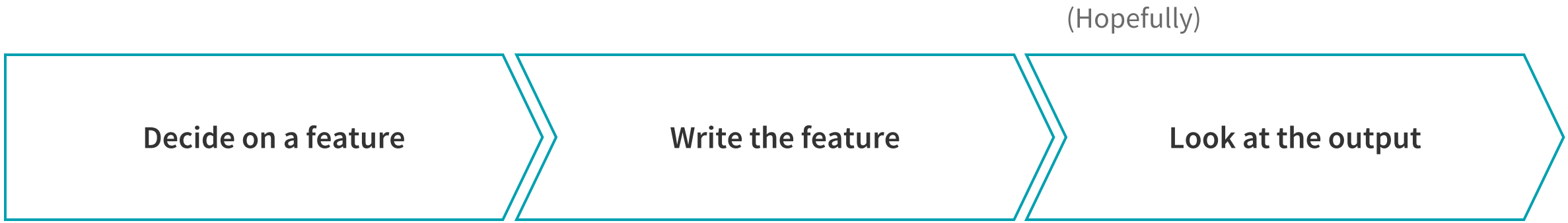
Section 2

Test Driven Development

Test Driven Development

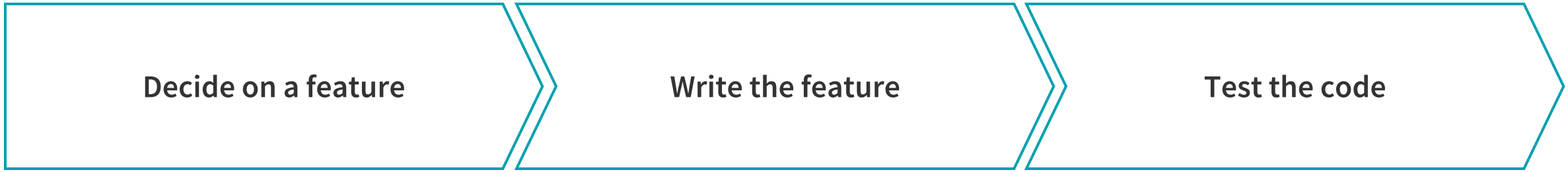
- A design practice for writing better code
- You write tests for your code as you go along
- De-clutters your code
- Quantifiable code quality

Writing code



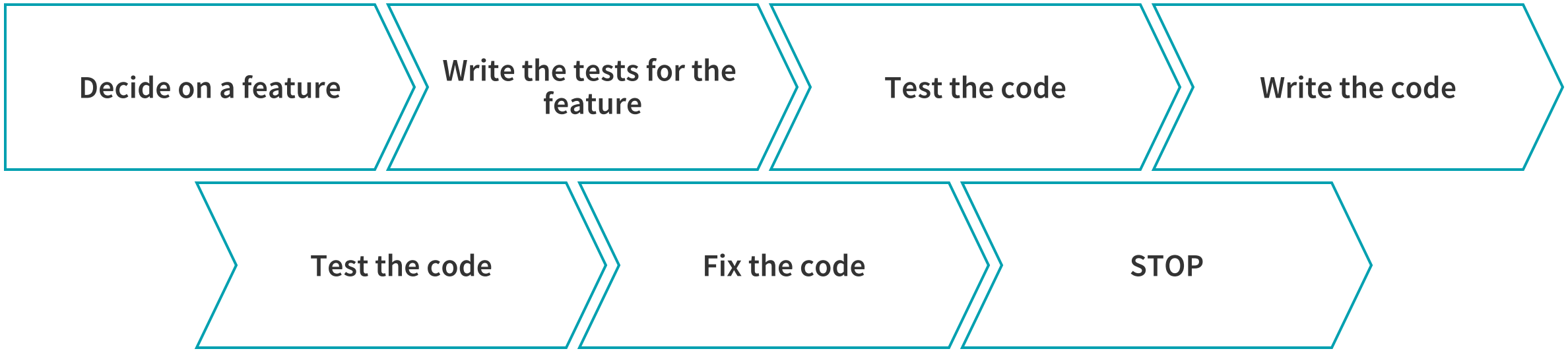
You are often taught this pattern

Writing code



Better

Test Driven Development



Ideal

TDD

- Your tests should all fail before you write any code
- Stop writing when all your tests pass
- Every function and method should have at least one test
- If you don't know how to test a function, your abstraction is wrong

POSIX

- Any element of a system, should take an input, produce an output and flag an error when it fails.
- “[Every element should] do one thing and do it well”

Testing approaches

Unit Testing

low level granularity. Tests a few lines of code at a time

Behavioural Testing

higher granularity. Tests groups of functions that provide a feature at a time.

Acceptance Testing

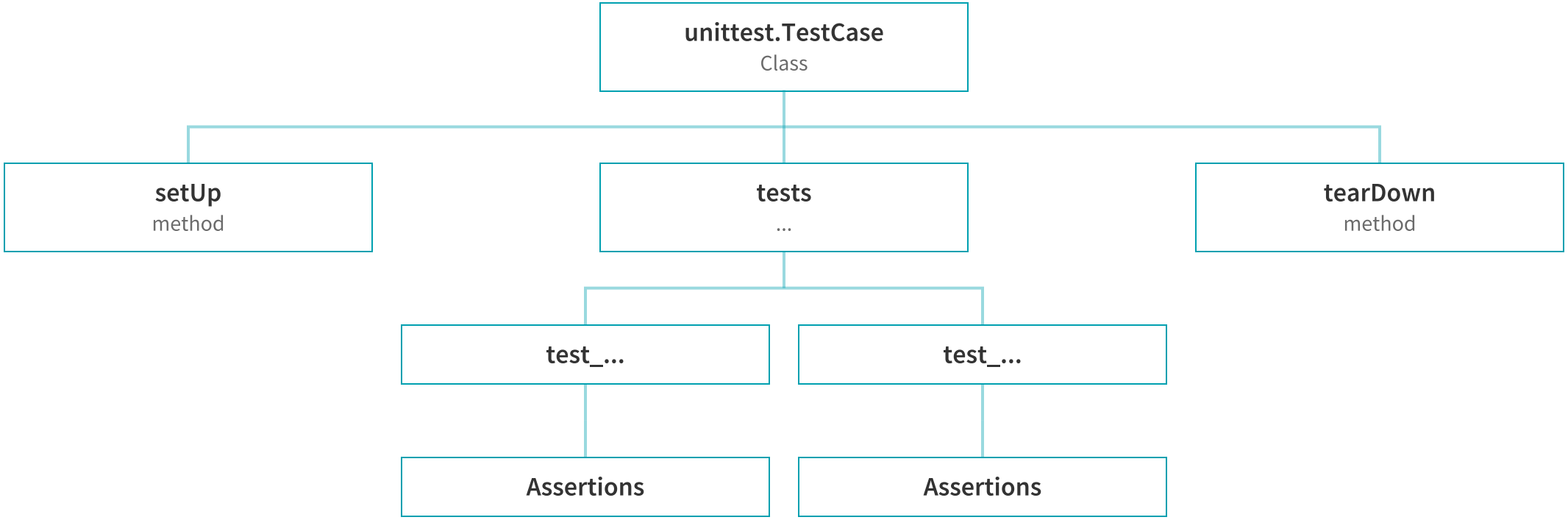
End-user viewpoint testing - more enterprise focused

Unit testing

Today's focus

- No need to reinvent the wheel - use python's inbuilt library
- each function/method should have at least one test
- you should test that a function act's correctly, but also that it fails properly when given the wrong information

Unittest class



setUp runs before every test, to set up any conditions for a test. tearDown runs after every test, usually to reset the code

```python
import unittest
from examples.oop_dogs import YorkshireTerrier


class TestYorkshireTerrier(unittest.TestCase):

    def setUp(self):
        self.yorkie_instance = YorkshireTerrier("test_yorkie")

    def test_has_fur(self):
        self.assertFalse(self.yorkie_instance.has_fur)


if __name__ == '__main__':
    unittest.main()
```

```python
import unittest
import contextlib
import io
from examples.oop_dogs import YorkshireTerrier


class TestYorkshireTerrier(unittest.TestCase):

    def setUp(self):
        self.yorkie_instance = YorkshireTerrier("test_yorkie")

    def test_has_fur(self):
        self.assertFalse(self.yorkie_instance.has_fur)

    def test_barks(self):
        f = io.StringIO()
        with contextlib.redirect_stdout(f):
            self.yorkie_instance.barks()

        bark_output = f.getvalue()

        self.assertEqual(bark_output, "test_yorkie says 'WOOF!'\n")


if __name__ == '__main__':
    unittest.main()

```

Section 2

Source Version Control

Git and GitHub

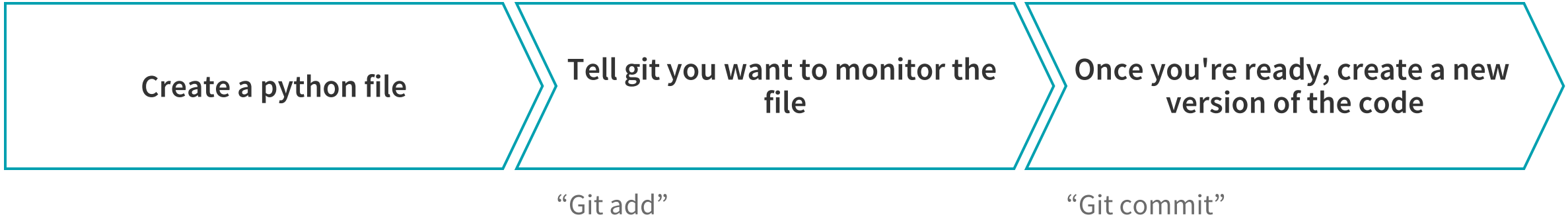
installing git

- www.gitkraken.com

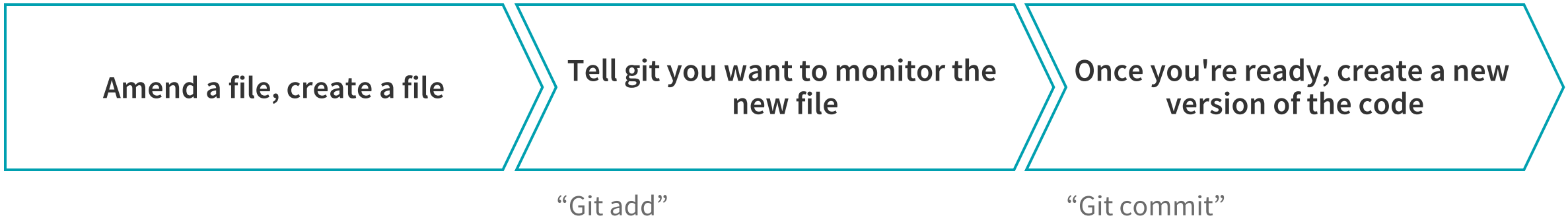
What is git?

- Developed by Linus Torvalds
- Manages versions of your codebase
- Looks for individual changes inside each code file
- Can roll your code back and forth in code versions
- Stores all the versions of your code locally

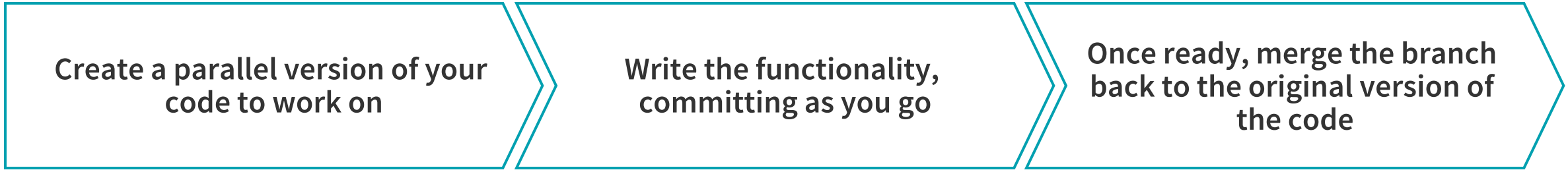
Workflow


Create a python file
Tell git you want to monitor the file
Once you're ready, create a new version of the code
“Git add”
“Git commit”

Workflow


Amend a file, create a file
Tell git you want to monitor the new file
Once you're ready, create a new version of the code
“Git add”
“Git commit”

Workflow - branching



"Create a branch"

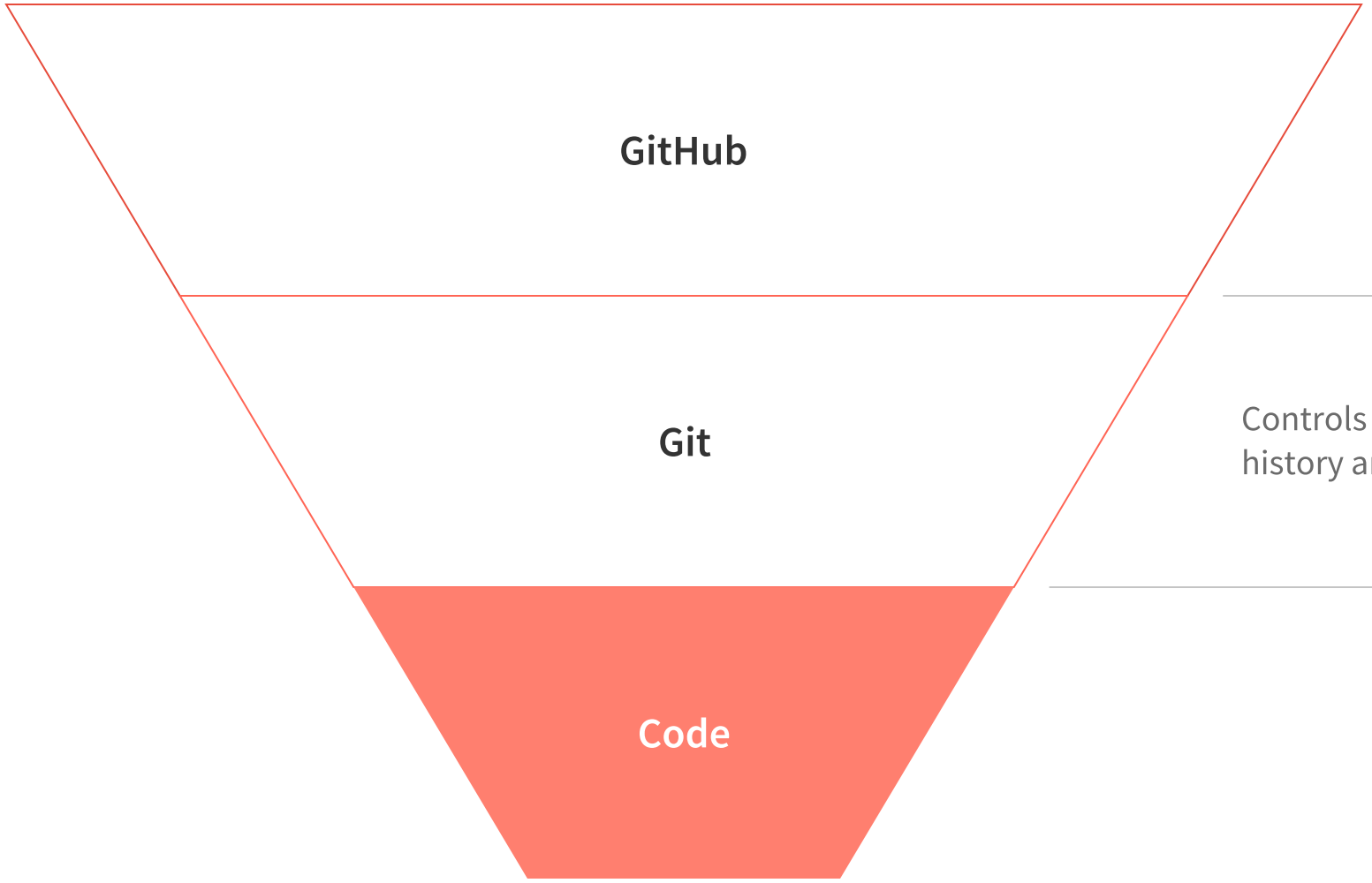
"git merge"

Your Master branch should hold a working version of your code

---

Each branch should focus on one feature

Git vs GitHub



Holds a remote copy of your git repository. Git interacts by pulling down remote commits and pushing up local commits

Controls local changes to your code, holding a commit history and versioning, for branches

Arrange yourselves into pairs

WITH ONE GROUP THREE IF WE ARE ODD-NUMBERED

Create a project repository

1. **One person should use github to create a new project called “STSimulator”**

   Initialise it with a ReadMe when prompted

2. **Everyone should log into GitKraken, and clone the GitHub repository**

   Look at the github project homepage for the HTTP git clone address


U.S.S. ENTERPRISE
NCC-1701

Design pattern

GitHub

One branch per feature

Commit when tests pass

Don't merge with failing tests

Tests

Write tests first

Write until the tests pass

refactor

Classes

representing logic as objects, not routines, may make the task easier

Comments

Every function and class needs a docstring

User flow

1. On start, the program lets the user place 30 new crew members
2. Each Crew member is named, and is given a random distribution of skills
3. The user picks which one of five stations they are best suited to
4. Bonus scores for each station are calculated, and the ship is ready

Task 1

Create an object for the Enterprise, which monitors its power levels, shield levels and crew count

Requirements

1. The ship has a maximum power level of 1000. It begins at maximum
2. The ship has a population cap of 30. It begins at 0
3. The ship has a name
4. The ship has a shield level of 100. It begins at maximum
5. The ship has a repr and a str method

Remember - create a branch and merge when done. Work together

Task 2

The Enterprise has 5 stations: Command, Science, Tactical, Engineering and Medical

They have their own bonus scores, crew levels and caps

Requirements

1. Each station begins with a score of 30
2. Each station has a crew cap of 6. It begins at 0
3. Each station must hold a list of its crew
4. Each station should have a str and repr method

Requirements

1. Each station begins with a score of 30
2. Each station has a crew cap of 6. It begins at 0
3. Each station must hold a list of its crew

Divide the stations between two and work concurrently. You may want to create a “station” base class and five child classes

Task 3

The Enterprise has new crew coming on board

They have their own abilities, related bonus scores, and names

Requirements

1. **Each crew member is named**

2. **Each crew member has a random series of scores, one for Eng., Com., Med., Tac., Sci.,**

   Calculate this by randomly rolling five 6-sided dice and summing the top three.

3. **Calculate the bonuses for each score**

Requirements

1. **Each crew member is named**

2. **Each crew member has a random series of scores, one for Eng., Com., Med., Tac., Sci.,**

   Calculate this by randomly rolling 5 6-sided dice and summing the top three.

3. **Calculate the bonuses for each score**

Split tasks between two in a new branch

Scores to Bonuses

Score	Bonus
3	-4
4-5	-3
6-7	-2
8-9	-1
10-11	0
12-13	1
14-15	2
16-17	3
18	4

Task 4

Stations can calculate their scores

Scores are their base score and the bonuses of the station crew

Requirements

1. The station can retrieve the bonuses of their crew members
2. The station can calculate their sum score
3. Calculate the bonuses for each score

Task 5

- The player can interact with the code to create their ship and staff it

Requirements

1. Create a “main.py” that runs the logic for all the classes, and takes a player through the program
2. You'll need to take user inputs
3. A player can only add staff to a station that has space
4. A player can only create 30 staff
5. A player should see the station score summaries after every staff assignment
6. A player can see a new staff member's bonuses and see the options for stations

What we covered

- import this
- ()[]{}
- Cases
- local & global
- variable naming
- is & ==
- copying
- assertions
- context managers
- DRY, YAGNI, KISS, SOC
- Function arguments
- first class objects
- lambdas
- classes
- attributes
- methods
- self
- __init__
- __str__ & __repr__
- Liskov Substitution Principle
- Dicts
- Sets
- Arrays

What we covered

- Single line comments
- Multiline comments
- DocStrings
- help()
- Google format DocStings
- Code Annotations
- Test Driven Development
- Unit Tests
- setUp, TestCase, tearDown
- TDD development cycle
- Git add, commit
- GitKraken
- GitHub
- Git pull, push

PYTHON TRICKS THE BOOK

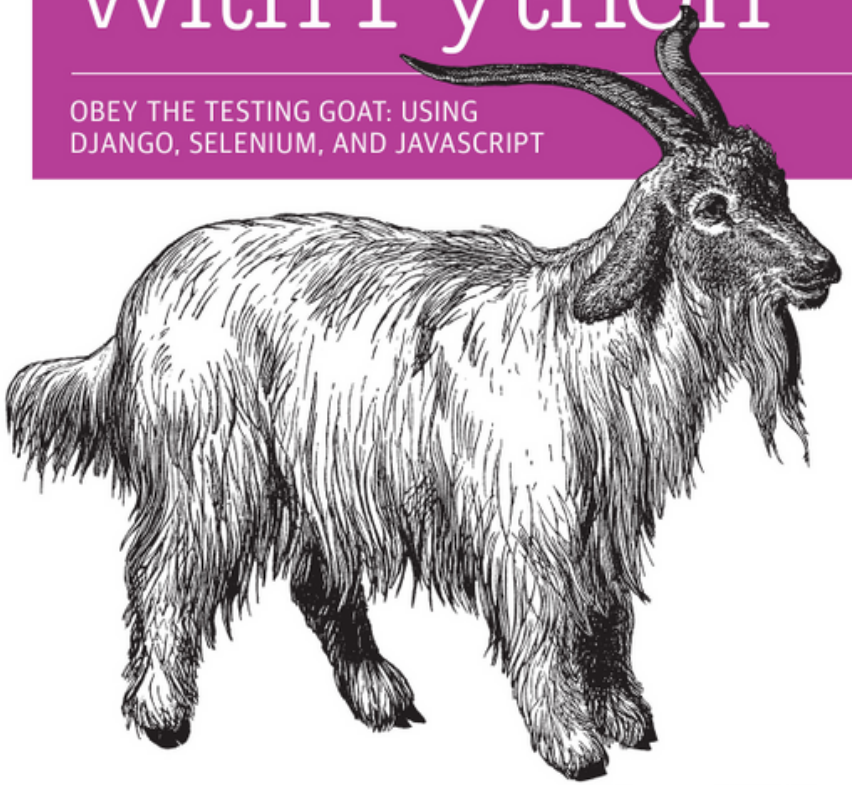
A Buffet of Awesome Python Features

Dan Bader

O'REILLY®

Test-Driven Development with Python

OBEY THE TESTING GOAT: USING DJANGO, SELENIUM, AND JAVASCRIPT

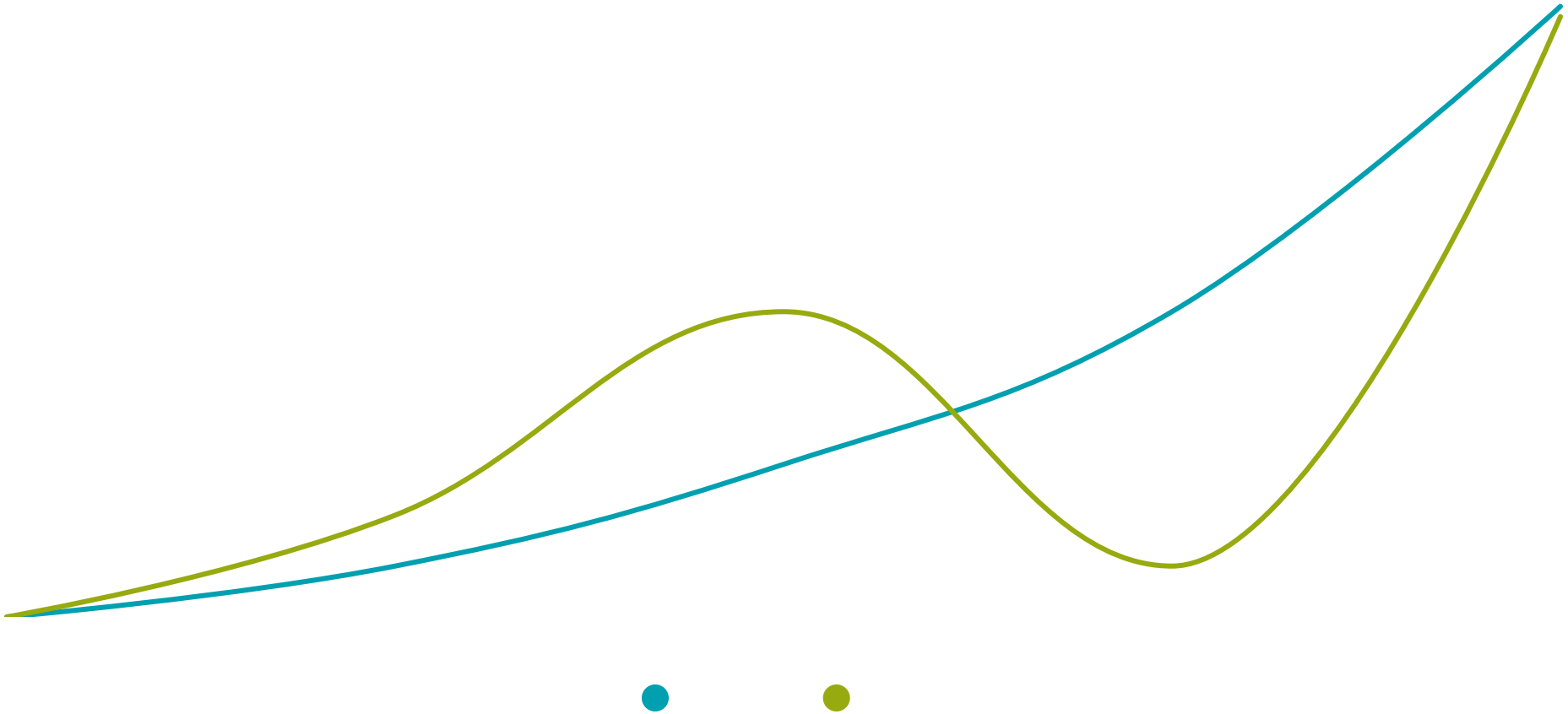


Harry J.W. Percival

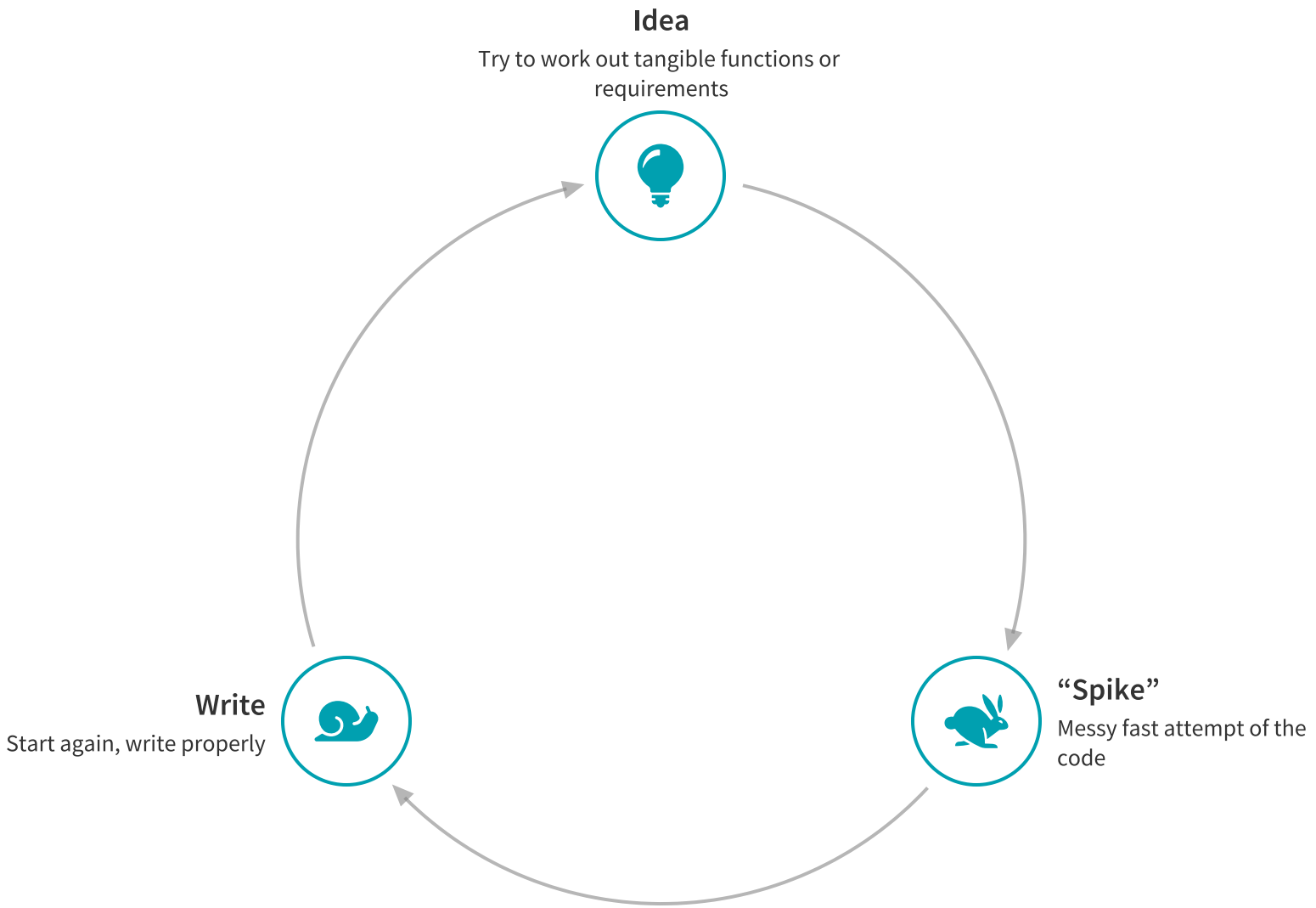
For the python masochists...

- decorators
- generators
- functional programming
- multi-threading
- packaging
- pyNose
- cPython extensions
- Continuous Integration
- custom Error Handling
- logging

Slow and steady



Starting a project



Good luck!