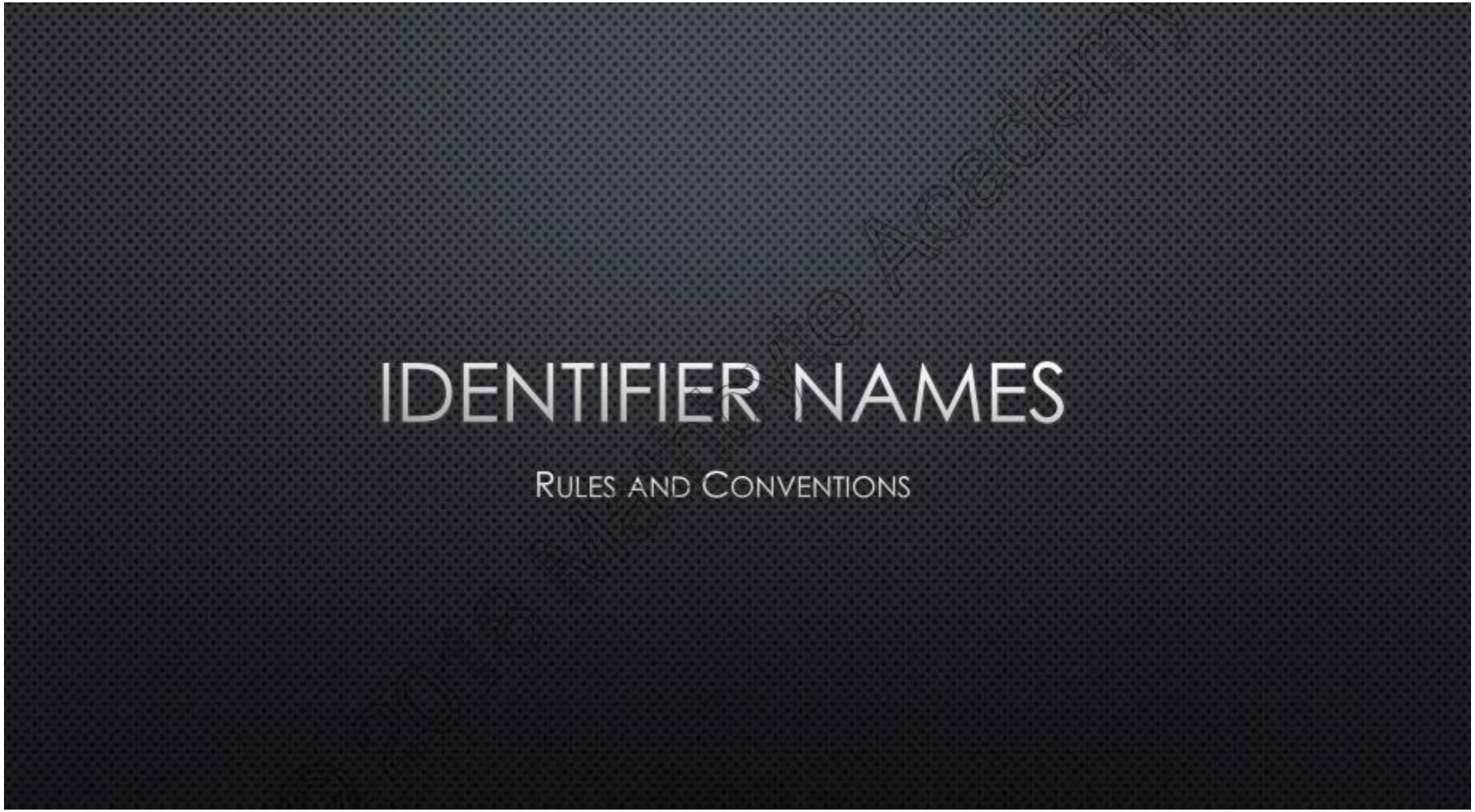
CAP 379
Artificial Intelligence Lab

Tanzeela Javid Kaloo (32638)
Assistant Professor
System And Architecture
Lovely Professional University

Python Fundamentals

Tanzeela Javid Kaloo (32638)
Assistant Professor
System And Architecture
Lovely Professional University


IDENTIFIER NAMES
RULES AND CONVENTIONS


Identifier names
are case-sensitive
my_var
my_Var
ham
Ham
are different identifiers
must follow certain rules
should follow certain conventions

Must

start with underscore (_) or letter (a-z A-Z)

followed by any number of underscores (_), letters (a-z A-Z), or digits (0-9)

`var` `my_var` `index1` `index_1` `_var` `__var` `__lt__` are all legal names

cannot be reserved words:

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

Conventions

`_my_var`

single underscore

This is a convention to indicate "internal use" or "private" objects

Objects named this way will not get imported by a statement such as :

```
from module import *
```

`__my_var`

double underscore

Used to "mangle" class attributes – useful in inheritance chains

`__my_var__`

double underscore

Used for system-defined names that have a special meaning to the interpreter.

Don't invent them, stick to the ones pre-defined by Python!

```
__init__
```

```
x < y  ⟶  x.__lt__(y)
```

Other Naming Conventions

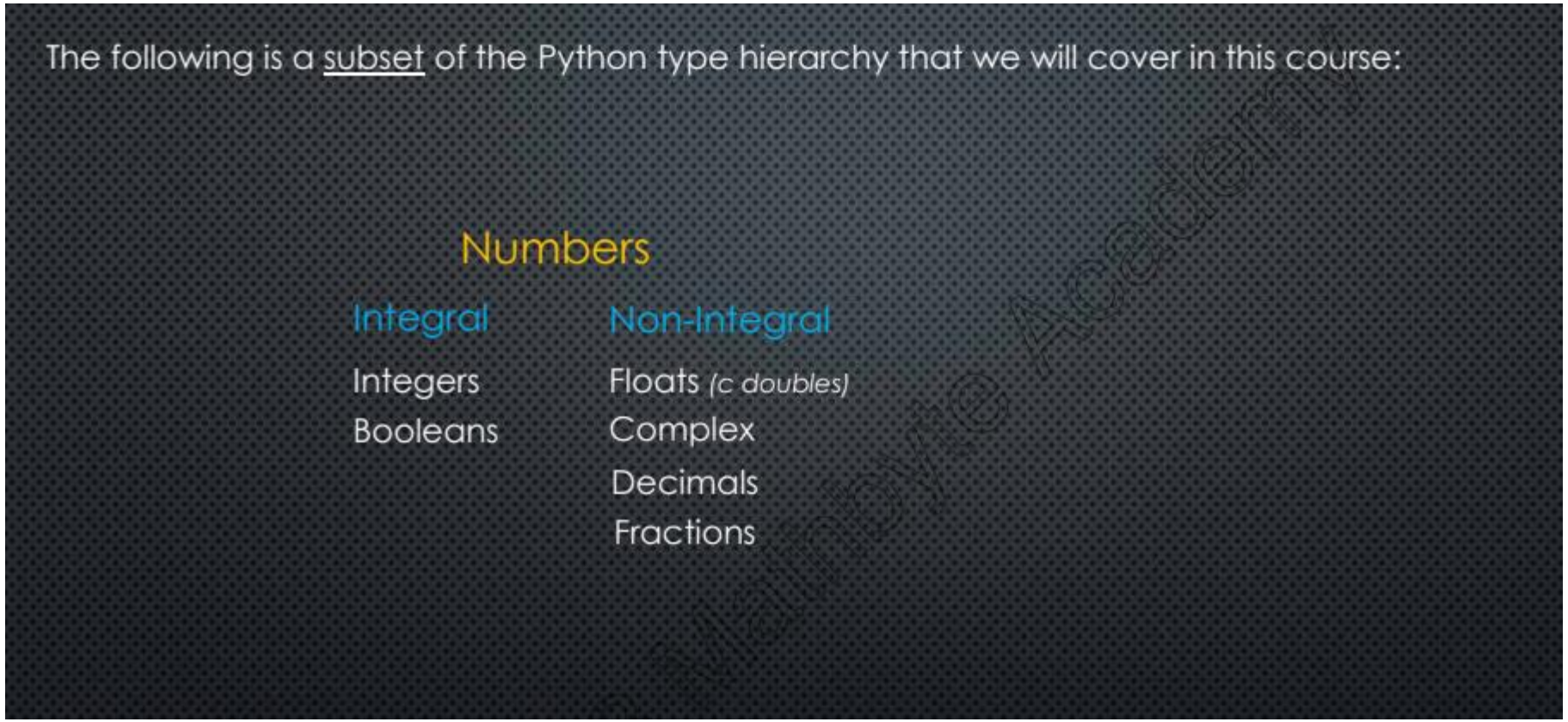
from the PEP 8 Style Guide

Packages	short, all-lowercase names. Preferably no underscores.	`utilities`
Modules	short, all-lowercase names. Can have underscores.	`db_utils` `dbutils`
Classes	CapWords (upper camel case) convention	`BankAccount`
Functions	lowercase, words separated by underscores (snake_case)	`open_account`
Variables	lowercase, words separated by underscores (snake_case)	`account_id`
Constants	all-uppercase, words separated by underscores	`MIN_APR`

https://www.python.org/dev/peps/pep-0008/ ← This is a should-read!

A foolish consistency is the hobgoblin of little minds
(Emerson)


PYTHON TYPE HIERARCHY
(SUBSET)


The following is a subset of the Python type hierarchy that we will cover in this course:
Numbers
Integral
Integers
Booleans
Non-Integral
Floats (c doubles)
Complex
Decimals
Fractions


Collections
Sequences
Mutable
Lists
Immutable
Tuples
Strings
Sets
Mutable
Sets
Immutable
Frozen Sets
Mappings
Dictionaries

Callables

User-Defined Functions

Generators

Classes

Instance Methods

Class Instances (`__call__()`)

Built-in Functions (e.g. `len()`, `open()`)

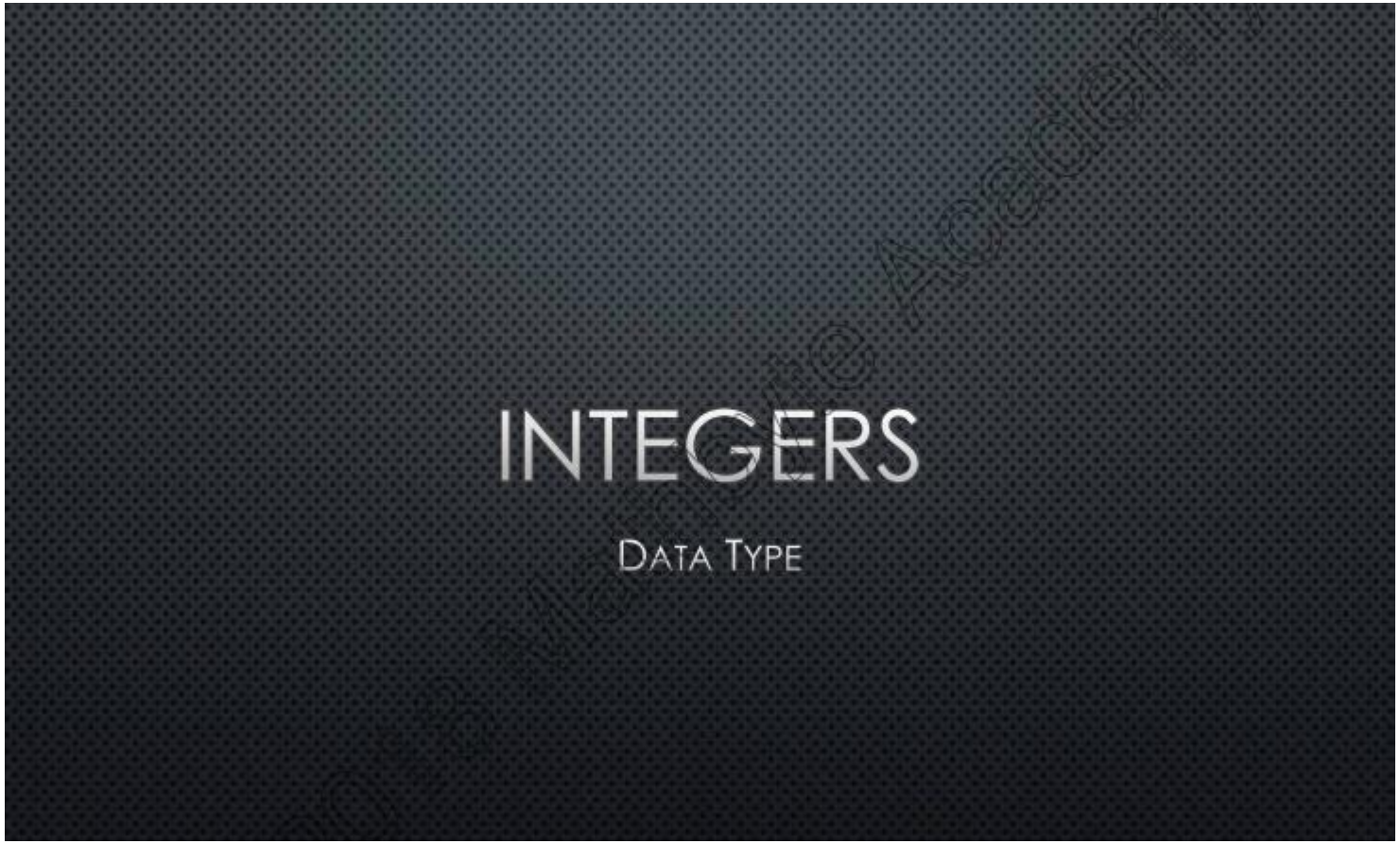
Built-in Methods (e.g. `my_list.append(x)`)

Singletons

None

NotImplemented

Ellipsis (...)

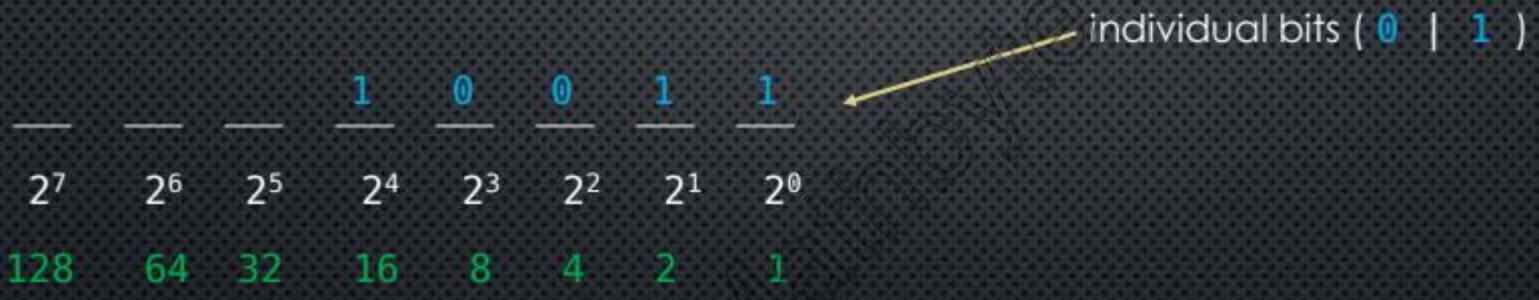

INTEGERS
DATA TYPE

The `int` data type

Ex: `0, 10, -100, 100000000, ...`

How large can a Python `int` become (positive or negative)?

Integers are represented internally using base-2 (binary) digits, not decimal.



$1 \times 16 + 0 \times 8 + 0 \times 4 + 1 \times 2 + 1 \times 1 = 16 + 2 + 1 = 19$

$(10011)_2 = (19)_{10}$

Representing the decimal number 19 requires **5 bits**

What's the largest (base 10) integer number that can be represented using 8 bits?

Let's assume first that we only care about non-negative integers

1	1	1	1	1	1	1	1
2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
128	64	32	16	8	4	2	1

$$128 + 64 + 32 + 16 + 8 + 4 + 2 + 1$$
$$= 255$$
$$= 2^8 - 1$$

If we care about handling negative integers as well, then 1 bit is reserved to represent the sign of the number, leaving us with only 7 bits for the number itself out of the original 8 bits

The largest number we can represent using 7 bits is $2^7 - 1 = 127$

So, using 8 bits we are able to represent all the integers in the range $[-127, 127]$

Since 0 does not require a sign, we can squeeze out an extra number,
and we end up with the range $[-128, 127]$

$$[-2^7, 2^7 - 1]$$

If we want to use 16 bits to store (signed) integers, our range would be:

$2^{(16-1)} = 2^{15} = 32,768$ Range: [-32,768 … 32,767]

Similarly, if we want to use 32 bits to store (signed) integers, our range would be:

$2^{(32-1)} = 2^{31} = 2,147,483,648$ Range: [-2,147,483,648 … 2,147,483,647]

If we had an unsigned integer type, using 32 bits our range would be:

$[0, 2^{32}] = [0 … 4,294,967,296]$

In a 32-bit OS:

memory spaces (bytes) are limited by their address number → 32 bits

4,294,967,296 bytes of addressable memory

= 4,294,967,296 / 1024 kB = 4,194,304 kB

= 4,194,304 / 1024 MB = 4,096 MB

= 4,096 / 1024 GB = 4 GB

So, how large an integer can be depends on how many bits are used to store the number.

Some languages (such as Java, C, …) provide multiple distinct integer data types that use a fixed number of bits:

Java

`byte`	signed 8-bit numbers	`-128, …, 127`
`short`	signed 16-bit numbers	`-32,768, …, 32,767`
`int`	signed 32-bit numbers	-2^{31} …, $2^{31} - 1$
`long`	signed 64-bit numbers	-2^{63} …, $2^{63} - 1$

and more…

Python does not work this way

The `int` object uses a ***variable*** number of bits

Can use 4 bytes (32 bits), 8 bytes (64 bits), 12 bytes (96 bits), etc. Seamless to us

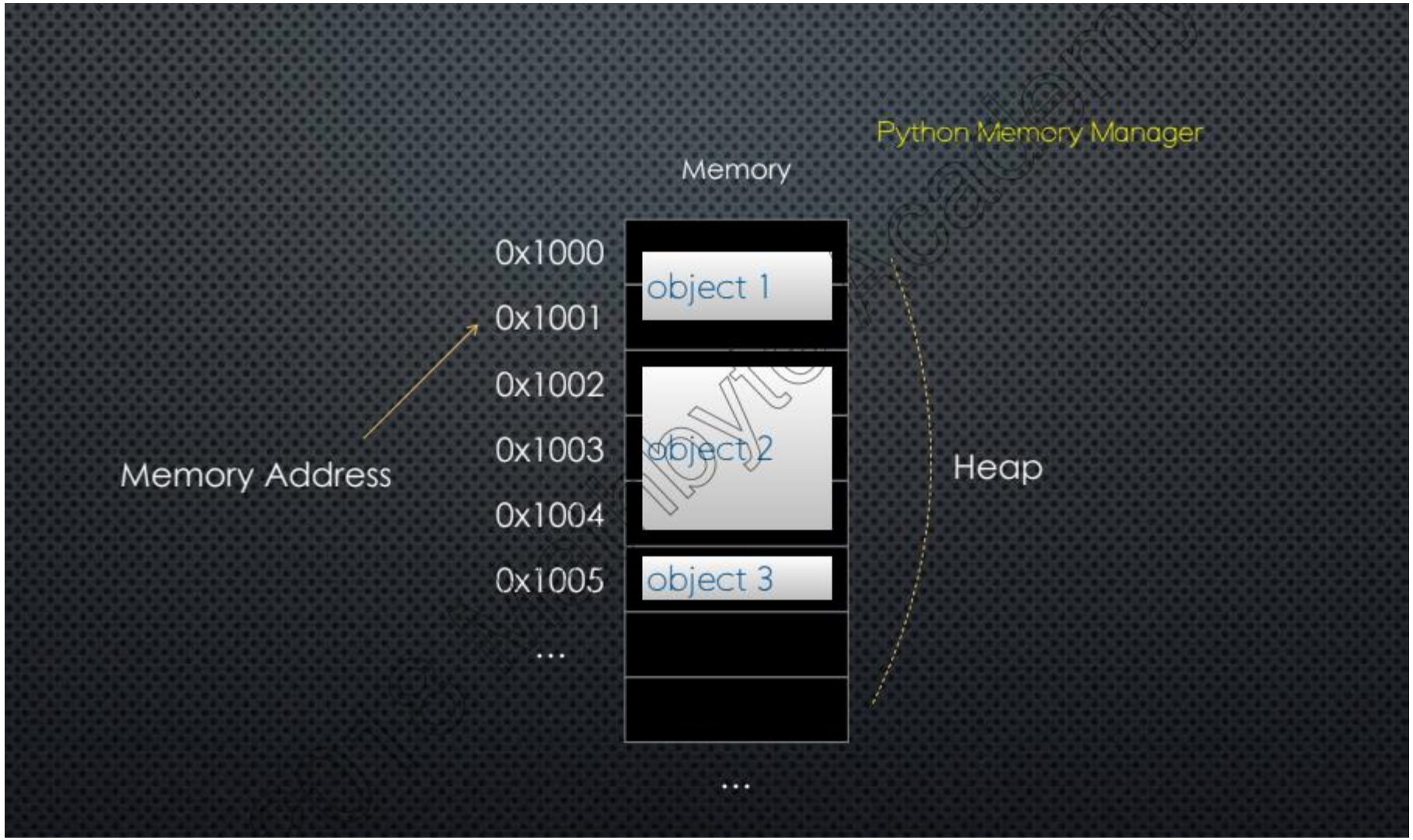
[since `int`s are actually objects, there is a further fixed overhead per integer]

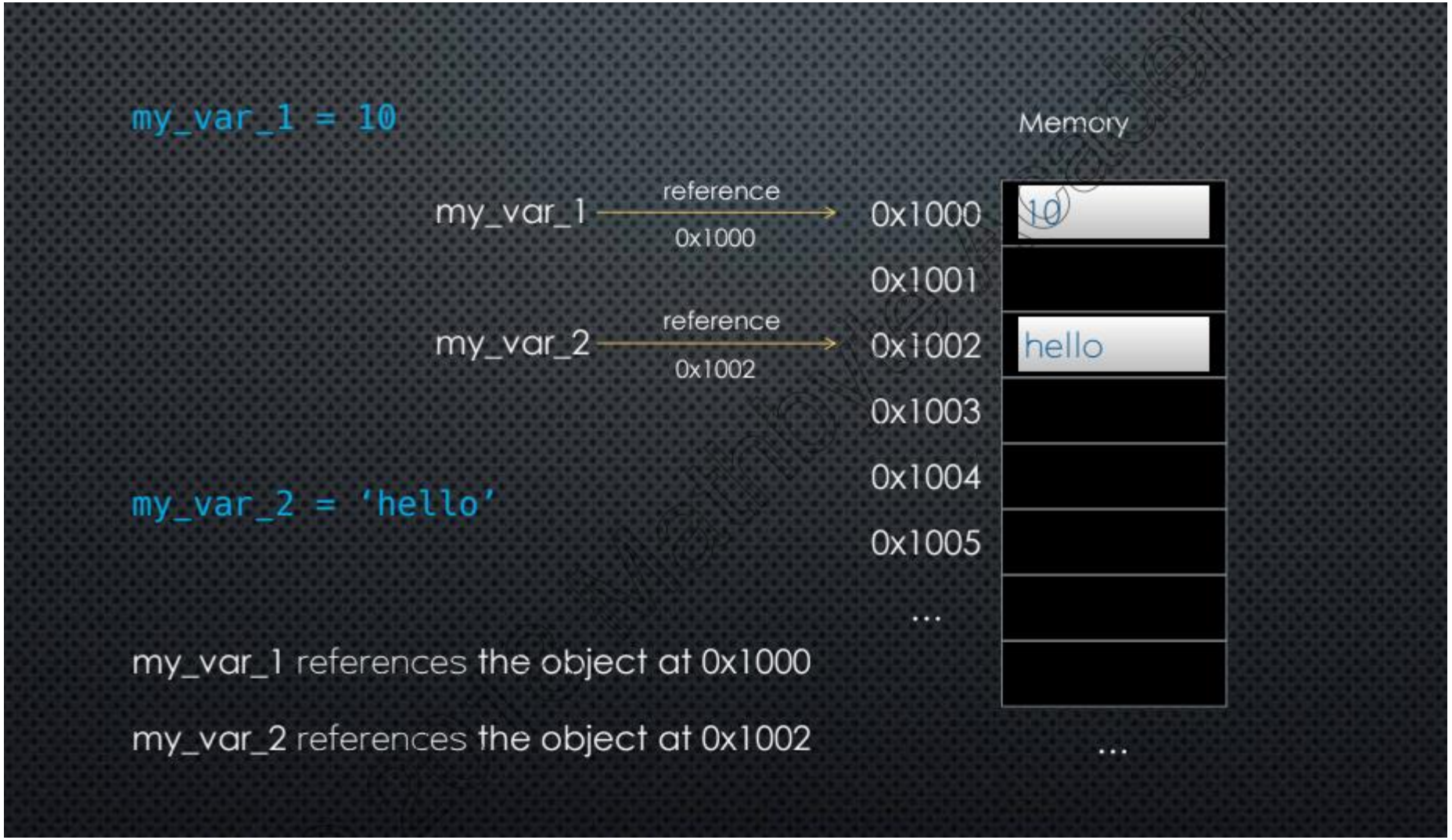
Theoretically limited only by the amount of memory available

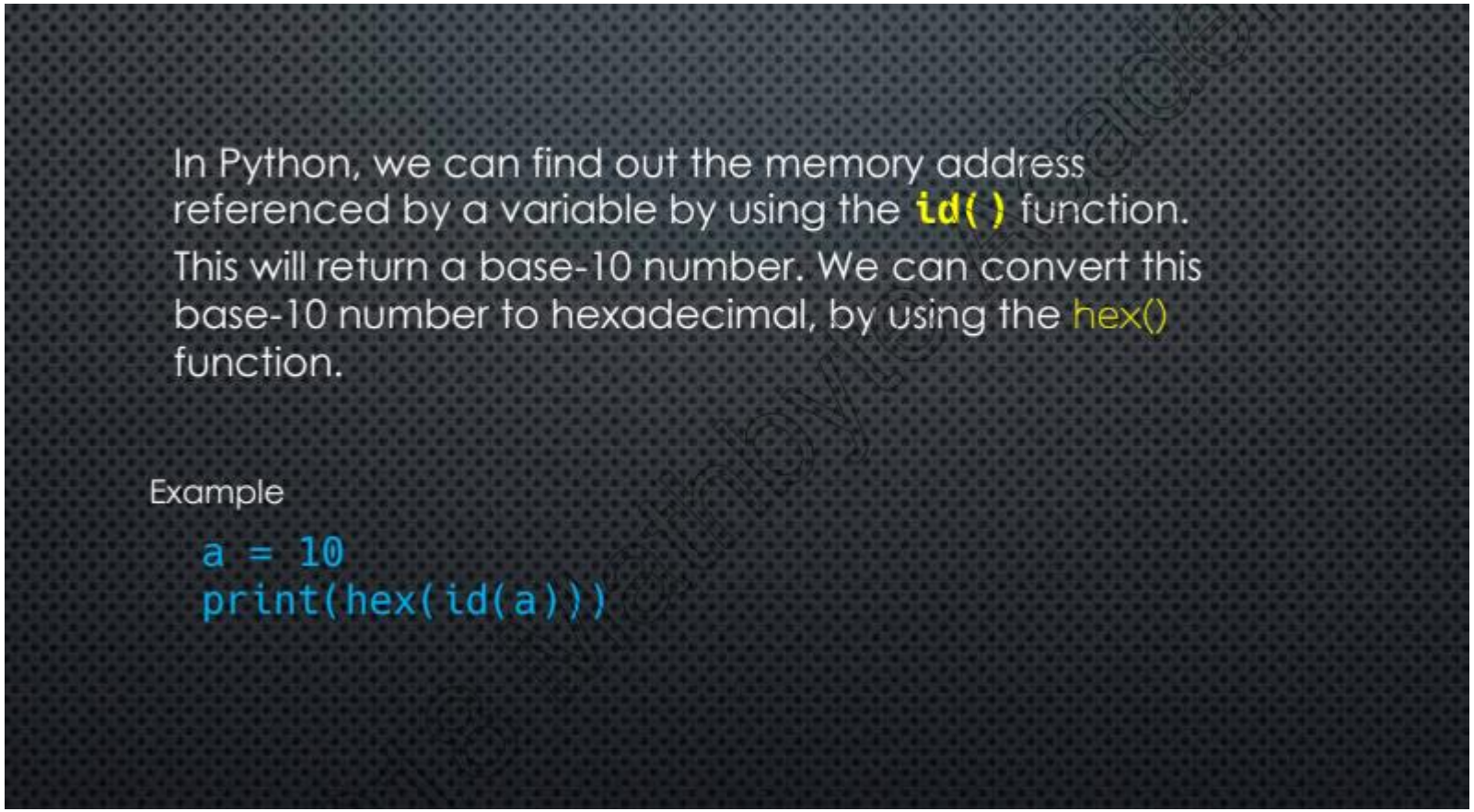
Of course, larger numbers will use more memory

and standard operators such as `+`, `*`, etc. will run slower as numbers get larger


VARIABLES ARE MEMORY REFERENCES


Python Memory Manager
Memory
0x1000
0x1001
0x1002
0x1003
0x1004
0x1005
...
object 1
object 2
object 3
Memory Address
Heap
...


my_var_1 = 10
Memory
my_var_1
reference
0x1000
0x1000
10
0x1001
my_var_2
reference
0x1002
0x1002
hello
0x1003
0x1004
my_var_2 = 'hello'
0x1005
...
my_var_1 references the object at 0x1000
my_var_2 references the object at 0x1002
...


In Python, we can find out the memory address referenced by a variable by using the id() function.
This will return a base-10 number. We can convert this base-10 number to hexadecimal, by using the hex() function.
Example
a = 10
print(hex(id(a)))


INTEGERS
OPERATIONS

Integers support all the standard arithmetic operations:

addition	+
subtraction	-
multiplication	*
division	/
exponents	**

But what is the resulting type of each operation?

```
int + int     → int
int - int     → int
int * int     → int
int ** int    → int
int / int     → float     obviously   3 / 4     → 0.75    (float)
                          but, also   10 / 2    → 5       (float)
```

Two more operators in integer arithmetic

First, we revisit long integer division...

$\frac{155}{4}$ numerator / denominator

155 // 4

```
  38
4 | 155
    12
    ---
    35
    32
    --
     3
```

155 % 4

155 ÷ 4 = 38 with remainder 3

put another way:

155 = 4 * 38 + 3

155 = 4 * (155 // 4) + (155 % 4)
= 4 * 38 + 3

// is called floor division (div) % is called the modulo operator (mod)

and they always satisfy: `n = d * (n // d) + (n % d)`

What is floor division exactly?

First define the floor of a (real) number

The floor of a real number a is the largest (in the standard number order) integer <= a

```
floor(3.14) → 3
floor(1.9999) → 1
floor(2) → 2
```

But watch out for negative numbers!

floor(-3.1) → -4

So, floor is not quite the same as truncation!

```
a // b = floor(a / b)
```

```
a = b * (a // b) + a % b
```

a = 135

b = 4

135 / 4 = 33.75 (33 3/4)

135 // 4 → 33

135 % 4 → 3

And, in fact: `a = b * (a // b) + a % b`

```
  4 * (135 // 4) + (135 % 4)
= 4 * 33 + 3
= 132 + 3
= 135
```

Negative Numbers

Be careful, `a//b`, is not the integer portion of `a / b`, it is the floor of `a / b`

For `a > 0` and `b > 0`, these are indeed the same thing

But beware when dealing with negative numbers!

```
a = -135
b = 4
```

-135 / 4 = -33.75 (-33 3/4)

```
-135 // 4  → -34
-135 % 4   → 1
```

```
135 // 4  → 33
135 % 4   → 3
```

And, in fact: `a = b * (a // b) + a % b`

```
  4 * (-135 // 4) + (-135 % 4)
= (4 * -34) + 1
= -136 + 1
= -135
```

Expanding this further...

a = 13 b = 4	a = -13 b = 4	a = 13 b = -4	a = -13 b = -4
13 / 4 → 3.25	-13 / 4 → -3.25	13 / -4 → -3.25	-13 / -4 → 3.25
13 // 4 → 3	-13 // 4 → -4	13 // -4 → -4	-13 // -4 → 3
13 % 4 → 1	-13 % 4 → 3	13 % -4 → -3	-13 % -4 → -1

In each of these cases: `a = b * (a // b) + a % b`

`4 * (13 // 4) + 13 % 4` = 12 + 1 = 13 ✓	`4 * (-13 // 4) + -13 % 4` = -16 + 3 = -13 ✓	`-4 * (13 // -4) + 13 % -4` = 16 + -3 = 13 ✓	`-4 * (-13 // -4) + -13 % -4` = -12 + -1 = -13 ✓


RATIONAL NUMBERS

Rational numbers are fractions of integer numbers

Ex: $\frac{1}{2}$ $-\frac{22}{7}$

Any real number with a finite number of digits after the decimal point is also a rational number

$0.45 \rightarrow \frac{45}{100}$ $0.123456789 \rightarrow \frac{123456789}{10^9}$

So $\frac{8.3}{4}$ is also rational $\frac{8.3}{4} = \frac{83/10}{4} = \frac{83}{10} \times \frac{1}{4} = \frac{83}{40}$

as is $\frac{8.3}{1.4}$ since $\frac{8.3}{1.4} = \frac{83/10}{14/10} = \frac{83}{10} \times \frac{10}{14} = \frac{83}{14}$

The Fraction Class

Rational numbers can be represented in Python using the Fraction class in the fractions module

```
from fractions import Fraction
x = Fraction(3, 4)
y = Fraction(22, 7)
z = Fraction(6, 10)
```

Fractions are automatically reduced:

```
Fraction(6, 10) → Fraction(3, 5)
```

Negative sign, if any, is always attached to the numerator:

```
Fraction(1, -4) → Fraction(-1, 4)
```

Constructors

```
Fraction(numerator=0, denominator=1)
Fraction(other_fraction)
Fraction(float)
Fraction(decimal)
Fraction(string)
```

```
Fraction('10')     → Fraction(10, 1)
Fraction('0.125')  → Fraction(1, 8)
Fraction('22/7')   → Fraction(22, 7)
```

Standard arithmetic operators are supported: `+, -, *, /`

and result in `Fraction` objects as well

$\frac{2}{3} \times \frac{1}{2} = \frac{2}{6} = \frac{1}{3}$ `Fraction(2, 3) * Fraction(1, 2) → Fraction(1, 3)`

$\frac{2}{3} + \frac{1}{2} = \frac{4}{6} + \frac{3}{6} = \frac{7}{6}$ `Fraction(2, 3) + Fraction(1, 2) → Fraction(7, 6)`

getting the numerator and denominator of `Fraction` objects:

```
x = Fraction(22, 7)
x.numerator          → 22
x.denominator        → 7
```

`float` objects have finite precision ⇒ any `float` object can be written as a fraction!

```
Fraction(0.75)     → Fraction(3, 4)
Fraction(1.375)    → Fraction(11, 8)
```

```
import math

x = Fraction(math.pi)          → Fraction(884279719003555, 281474976710656)
y = Fraction(math.sqrt(2))     → Fraction(6369051672525773, 4503599627370496)
```

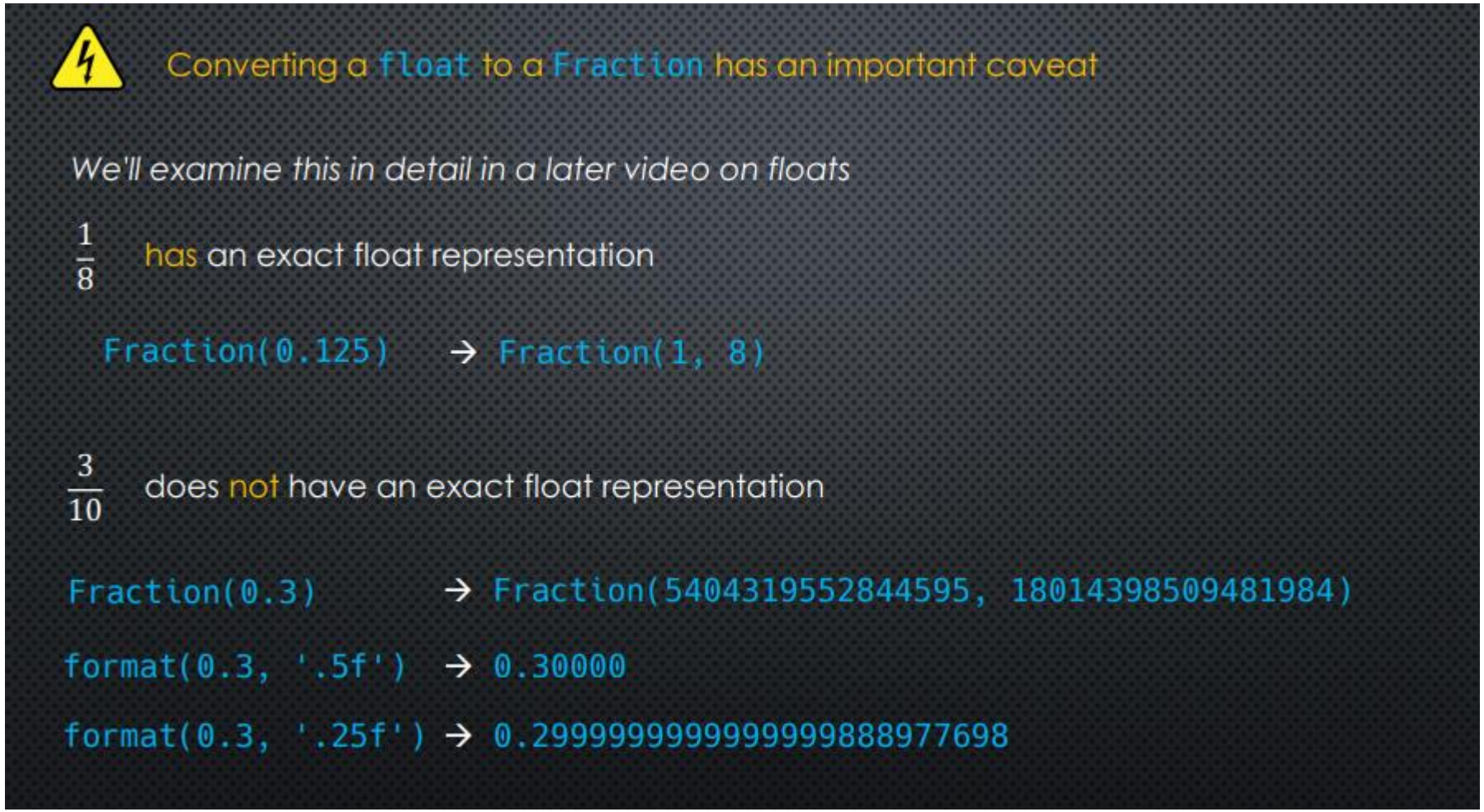
Even though π and $\sqrt{2}$ are both irrational

internally represented as floats

⇒ finite precision real number

⇒ expressible as a rational number

but it is an approximation


Converting a float to a Fraction has an important caveat
We'll examine this in detail in a later video on floats
1/8 has an exact float representation
Fraction(0.125) → Fraction(1, 8)
3/10 does not have an exact float representation
Fraction(0.3) → Fraction(5404319552844595, 18014398509481984)
format(0.3, '.5f') → 0.30000
format(0.3, '.25f') → 0.2999999999999999888977698

Constraining the denominator

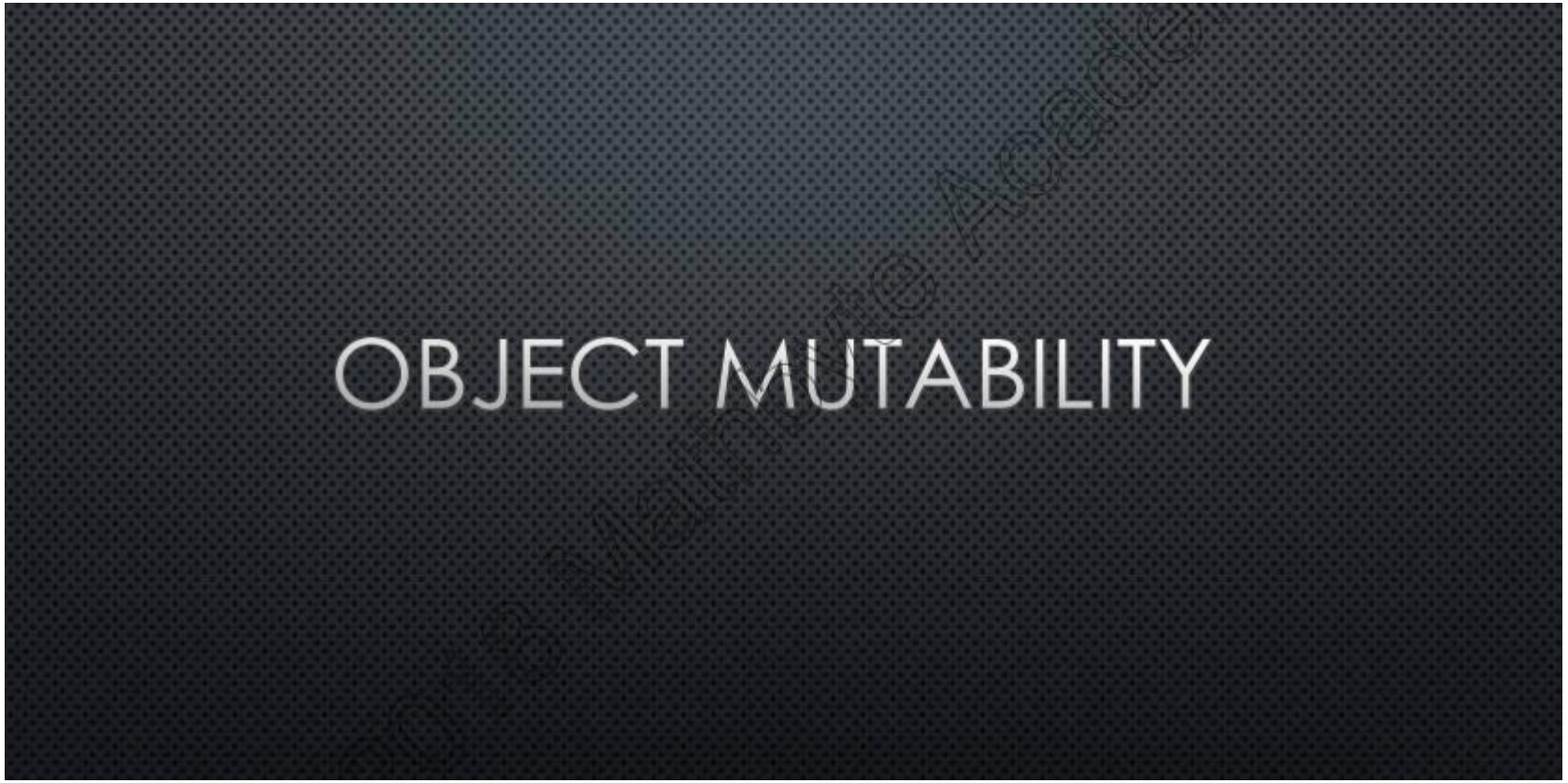
Given a `Fraction` object, we can find an approximate equivalent fraction with a constrained denominator

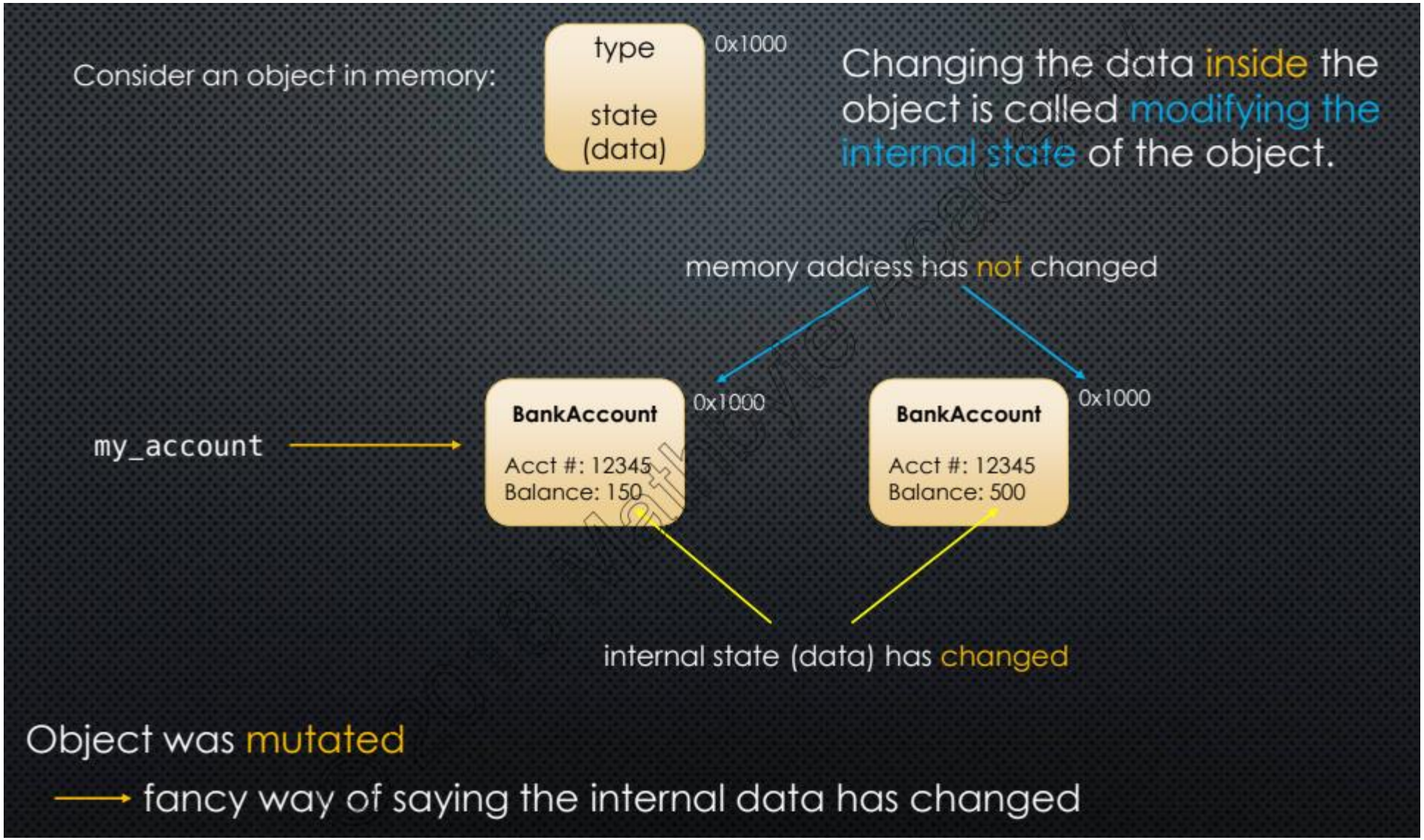
using the `limit_denominator(max_denominator=1000000)` instance method

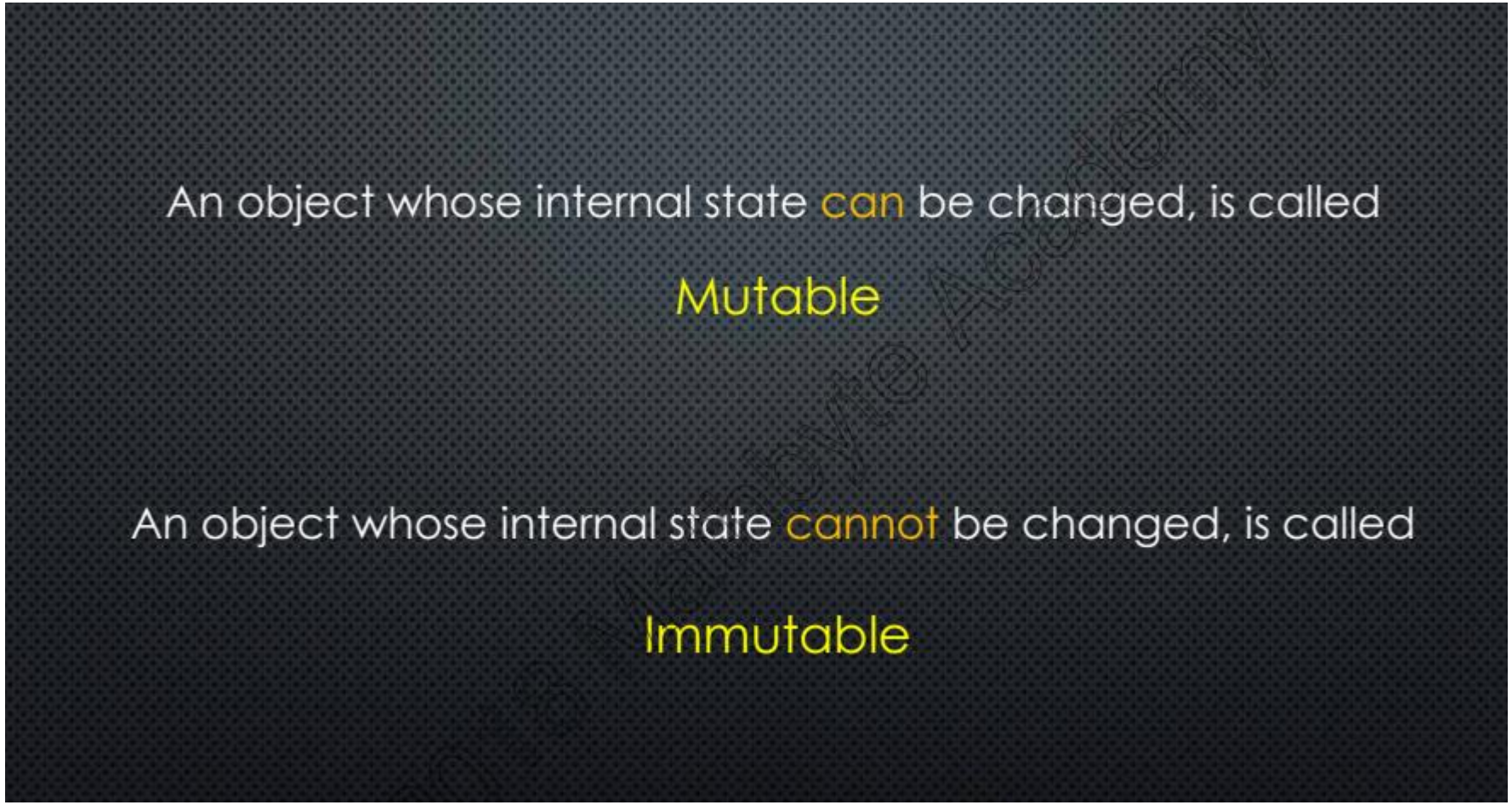
i.e. finds the closest rational (which could be precisely equal)

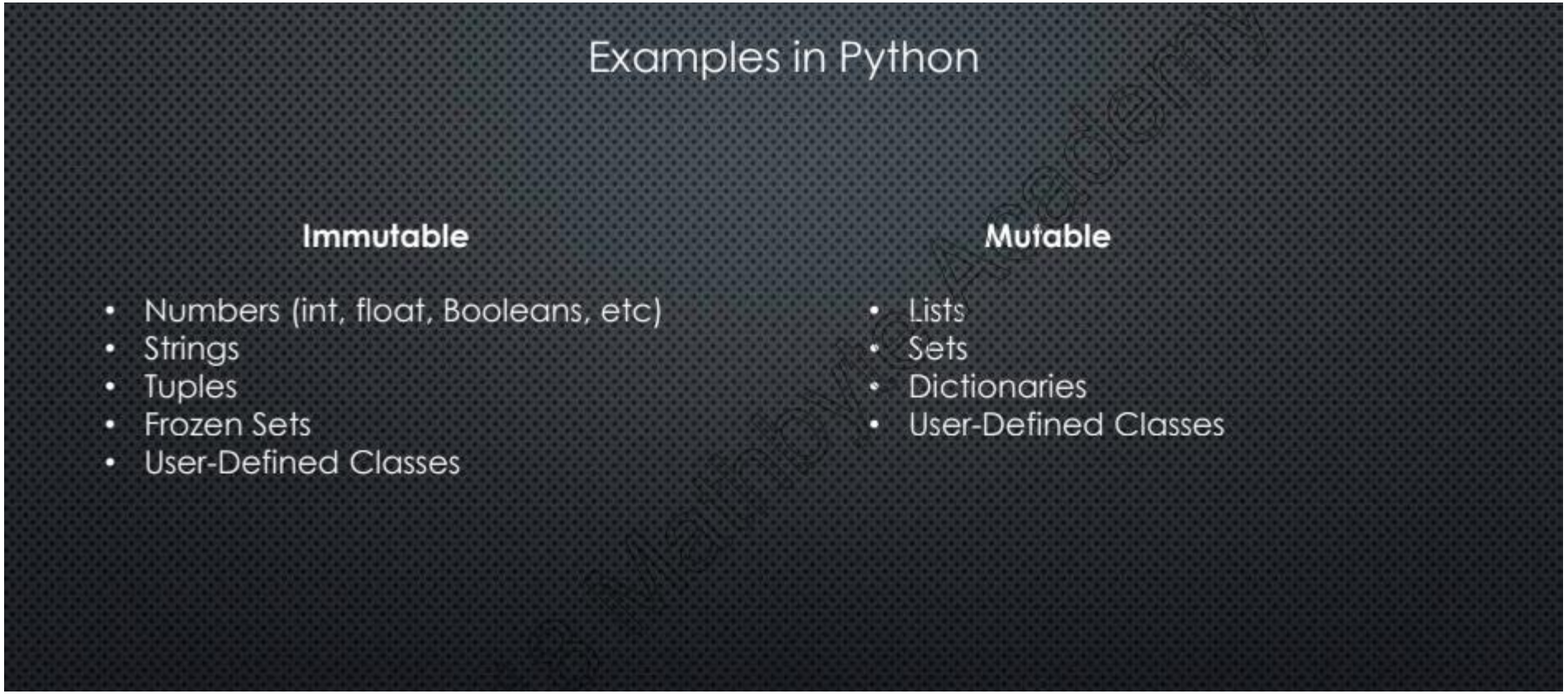
with a denominator that does not exceed `max_denominator`

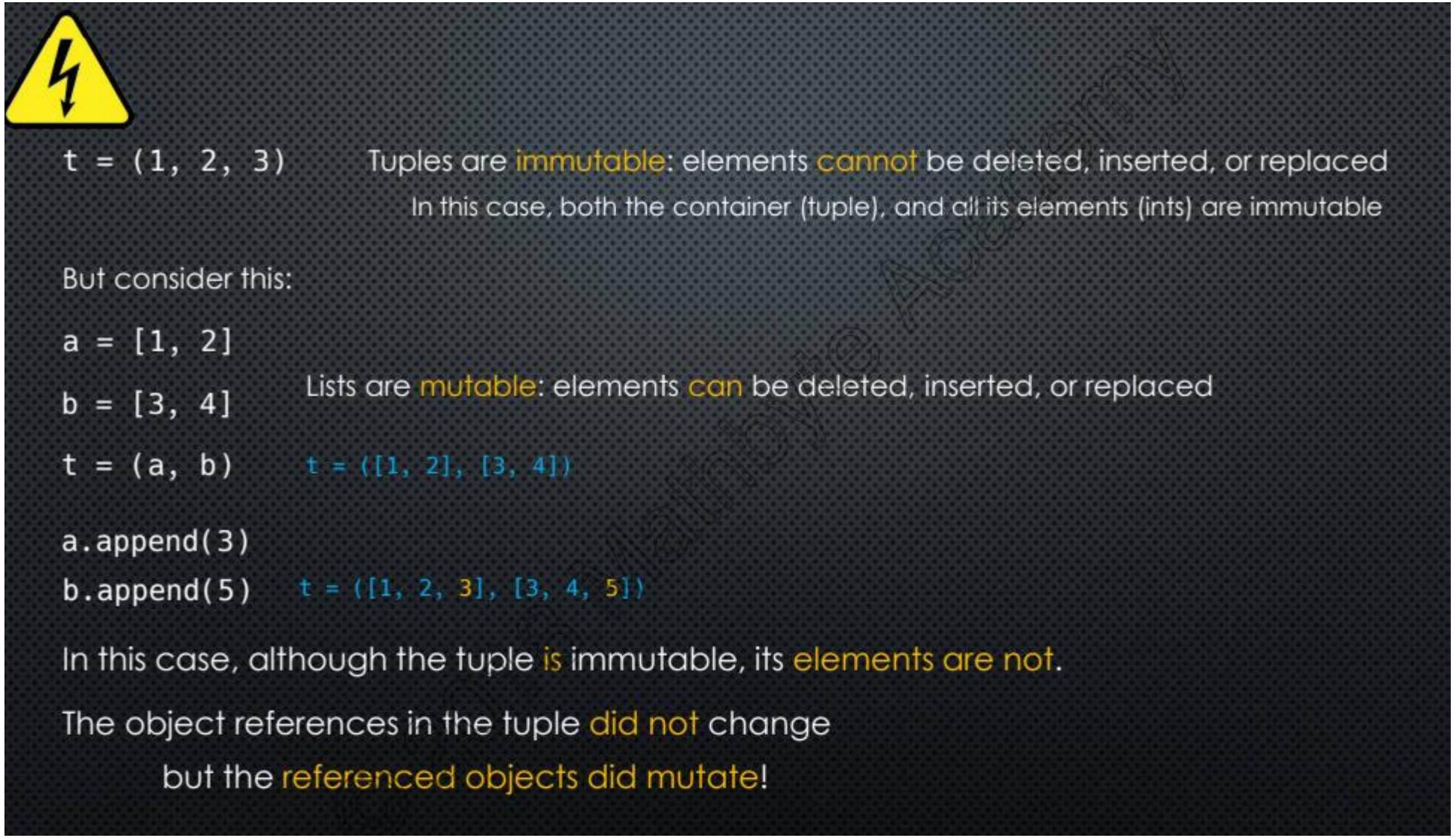
```
x = Fraction(math.pi)          → Fraction(884279719003555, 281474976710656)
                                 3.141592653589793

x.limit_denominator(10)        → Fraction(22, 7)
                                 3.142857142857143

x.limit_denominator(100)       → Fraction(311, 99)
                                 3.141414141414141

x.limit_denominator(500)       → Fraction(355, 113)
                                 3.141592920353983
```

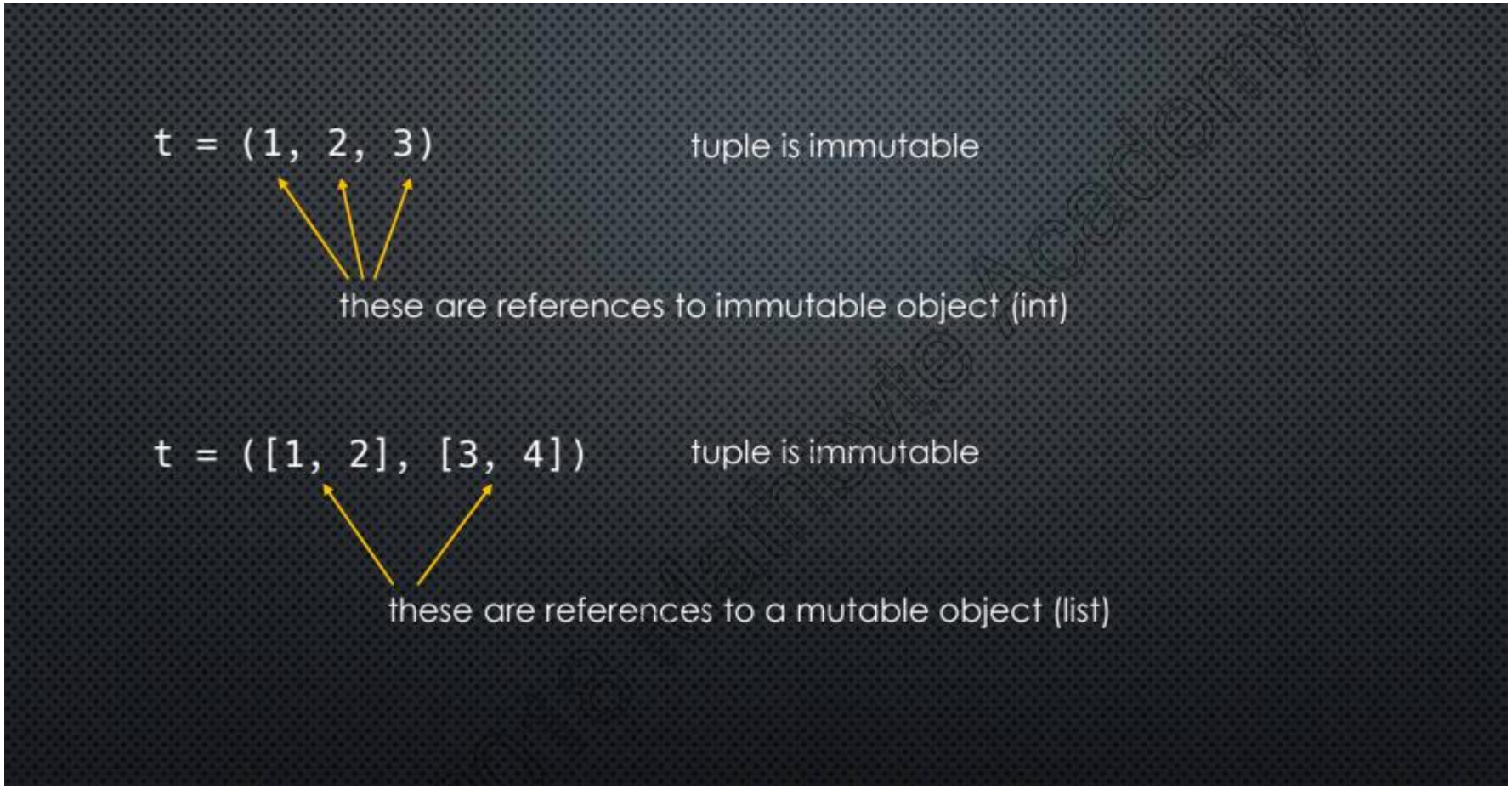

OBJECT MUTABILITY


Consider an object in memory:
type
state
(data)
0x1000
Changing the data inside the object is called modifying the internal state of the object.
memory address has not changed
my_account
BankAccount
Acct #: 12345
Balance: 150
0x1000
BankAccount
Acct #: 12345
Balance: 500
0x1000
internal state (data) has changed
Object was mutated
fancy way of saying the internal data has changed


An object whose internal state can be changed, is called
Mutable
An object whose internal state cannot be changed, is called
Immutable


Examples in Python
Immutable
Numbers (int, float, Booleans, etc)
Strings
Tuples
Frozen Sets
User-Defined Classes
Mutable
Lists
Sets
Dictionaries
User-Defined Classes


t = (1, 2, 3)
Tuples are immutable: elements cannot be deleted, inserted, or replaced
In this case, both the container (tuple), and all its elements (ints) are immutable
But consider this:
a = [1, 2]
b = [3, 4]
Lists are mutable: elements can be deleted, inserted, or replaced
t = (a, b)
t = ([1, 2], [3, 4])
a.append(3)
b.append(5)
t = ([1, 2, 3], [3, 4, 5])
In this case, although the tuple is immutable, its elements are not.
The object references in the tuple did not change
but the referenced objects did mutate!


t = (1, 2, 3)
tuple is immutable
these are references to immutable object (int)
t = ([1, 2], [3, 4])
tuple is immutable
these are references to a mutable object (list)


MULTI-LINE STATEMENTS
AND
STRINGS

Python Program

→ physical lines of code — end with a physical newline character

→ logical lines of code — end with a logical NEWLINE token

→ tokenized

physical newline vs logical newline

sometimes, physical newlines are ignored

in order to combine multiple physical lines

into a single logical line of code

terminated by a logical NEWLINE token

Conversion can be implicit or explicit

Implicit

Expressions in:

- list literals: [ ]
- tuple literals: ( )
- dictionary literals: { }
- set literals: { }
- function arguments / parameters

supports inline comments

```
[1,
 2,
 3]
```

```
[1, #item 1
 2, #item 2
 3  #item 3
]
```

```
def my_func(a,
            b, #comment
            c):
    print(a, b, c)

my_func(10, #comment
        20, 30)
```

Explicit

You can break up statements over multiple lines explicitly, by using the \ (backslash) character

Multi-line statements are not implicitly converted to a single logical line.

```
if a
   and b
   and c:
```

```
if a \
   and b \
   and c:
```

Comments cannot be part of a statement, not even a multi-line statement.

```
if a \
   and b \ #comment
   and c:
```

Multi-Line String Literals

Multi-line string literals can be created using triple delimiters (' single or " double)

```
'''This is
a multi-line string'''
```

```
"""This is
a multi-line string"""
```

Be aware that non-visible characters such as newlines, tabs, etc. are actually part of the string – basically anything you type.

You can use escaped characters (e.g. `\n`, `\t`), use string formatting, etc.

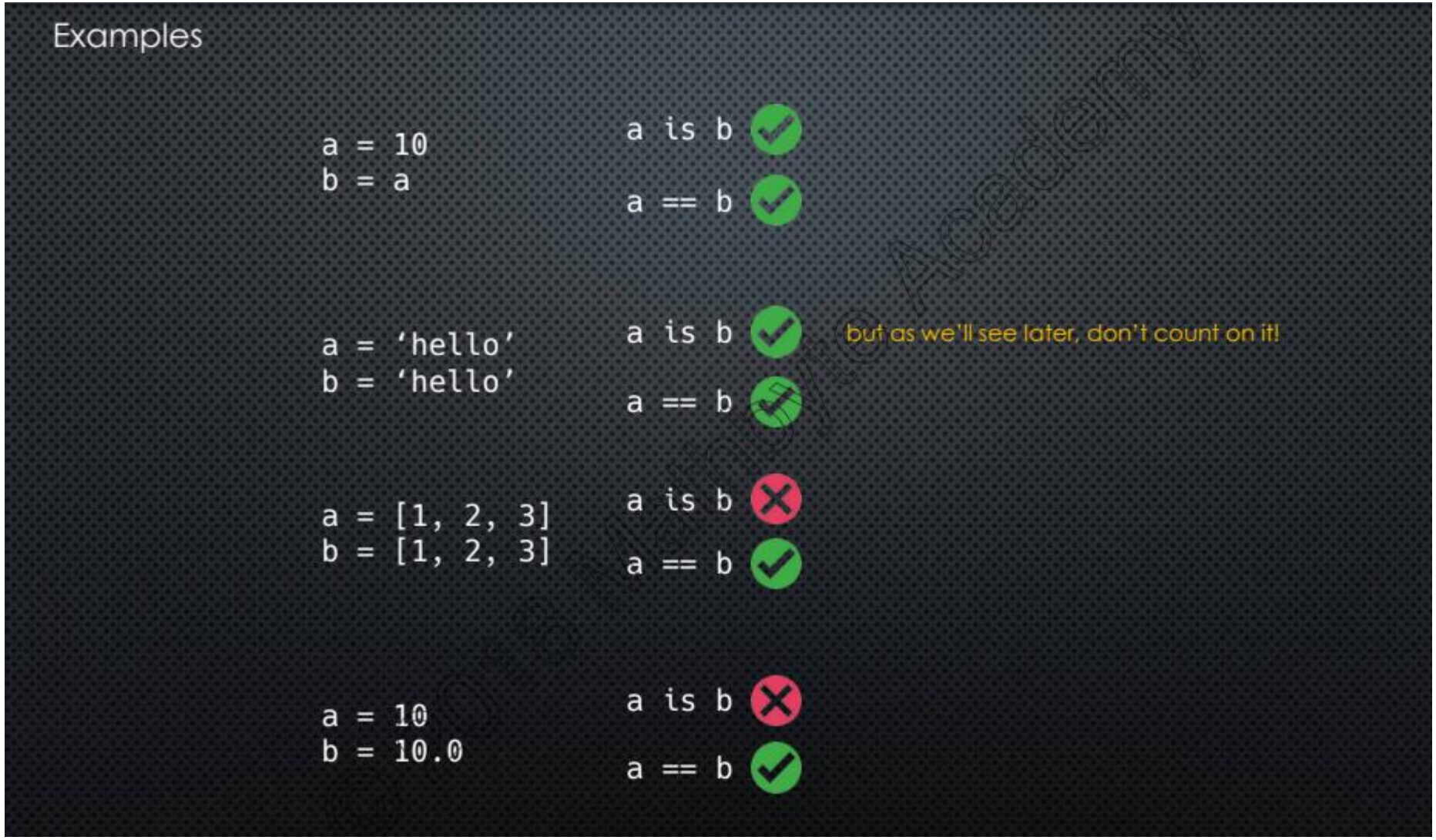
A multi-line string is just a regular string.

Multi-line strings are not comments, although they can be used as such, especially with special comments called `docstrings`.


VARIABLE EQUALITY

We can think of variable equality in two fundamental ways:

	Memory Address	Object State (data)
	is	==
	identity operator	equality operator
	`var_1 is var_2`	`var_1 == var_2`
Negation	`is not`	`!=`
	`var_1 is not var_2`	`var_1 != var_2`
	`not(var_1 is var_2)`	`not(var_1 == var_2)`


Examples
a = 10
b = a
a is b
a == b
a = 'hello'
b = 'hello'
a is b
a == b
but as we'll see later, don't count on it!
a = [1, 2, 3]
b = [1, 2, 3]
a is b
a == b
a = 10
b = 10.0
a is b
a == b

The None object

The None object can be assigned to variables to indicate that they are not set (in the way we would expect them to be), *i.e. an "empty" value (or null pointer)*

But the None object is a real object that is managed by the Python memory manager

Furthermore, the memory manager will always use a shared reference when assigning a variable to None

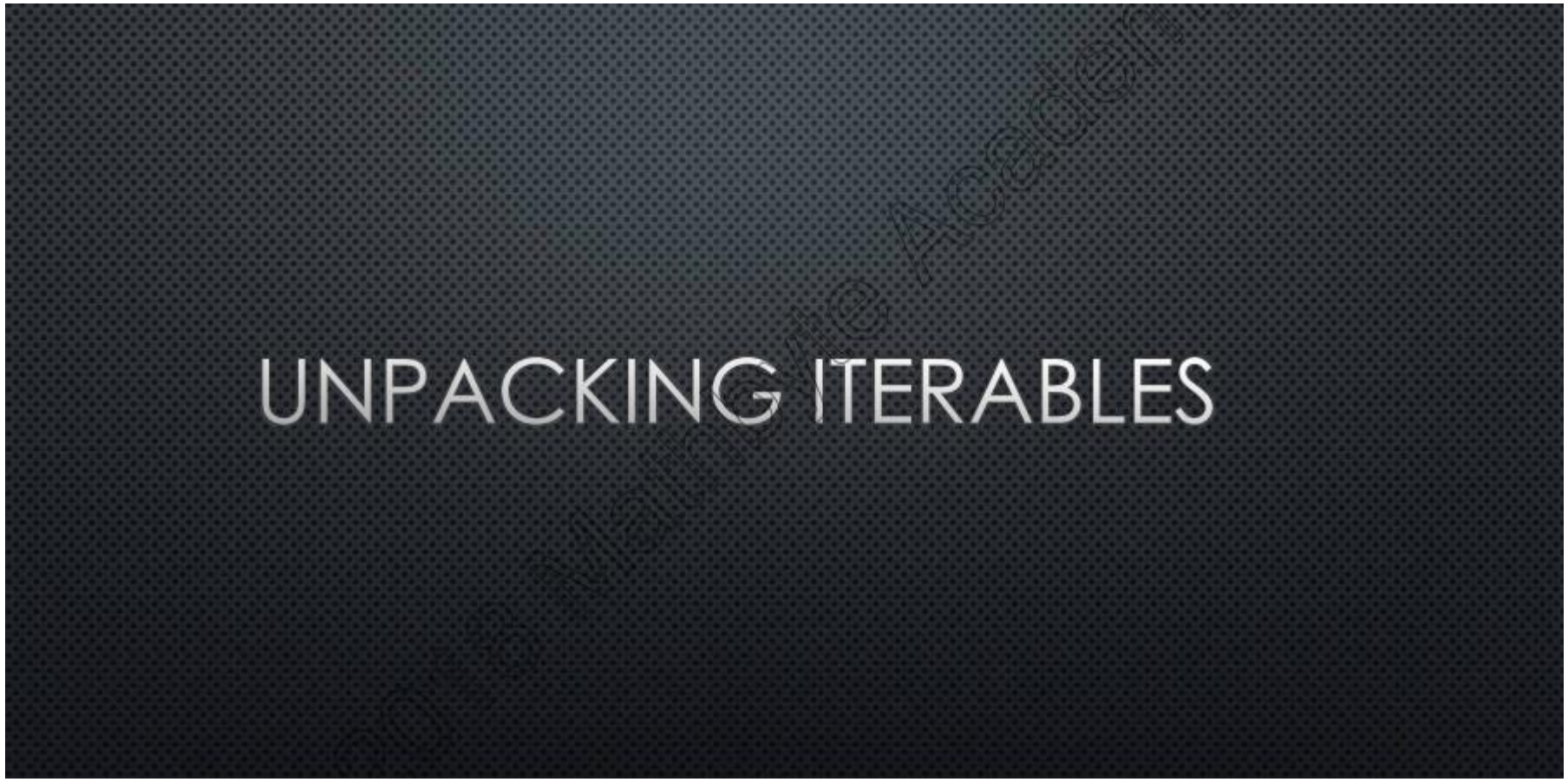
```
a = None
b = None
c = None
```

So we can test if a variable is "not set" or "empty" by comparing it's memory address to the memory address of None using the `is` operator

```
a is None
```

```
x = 10
```

```
x is None
x is not None
```


UNPACKING ITERABLES

A Side Note on Tuples

`(1, 2, 3)`

What defines a tuple in Python, is not `( )`, but `,`

`1, 2, 3` is also a tuple → `(1, 2, 3)` The `( )` are used to make the tuple clearer

To create a tuple with a single element:

`(1)` will not work as intended → `int`

`1,` or `(1, )` → `tuple`

The only exception is when creating an empty tuple: `()` or `tuple()`

Packed Values

Packed values refers to values that are bundled together in some way

Tuples and Lists are obvious

```
t = (1, 2, 3)
l = [1, 2, 3]
```

Even a string is considered to be a packed value:

```
s = 'python'
```

Sets and dictionaries are also packed values:

```
set1 = {1, 2, 3}
d = {'a': 1, 'b': 2, 'c': 3}
```

In fact, any iterable can be considered a packed value

Unpacking Packed Values

Unpacking is the act of splitting packed values into individual variables contained in a list or tuple

```
a, b, c = [1, 2, 3]
```

3 elements in `[1, 2, 3]` → need 3 variables to unpack

this is actually a tuple of 3 variables: a, b and c

a → 1 b → 2 c → 3

The unpacking into individual variables is based on the relative positions of each element

Does this remind you of how positional arguments were assigned to parameters in function calls?

Unpacking other Iterables

```
a, b, c = 10, 20, 'hello'        → a = 10   b = 20   c = 'hello'
```

this is actually a tuple containing 3 values

```
a, b, c = 'XYZ'                  → a = 'X'   b = 'Y'   c = 'Z'
```

instead of writing

```
a = 10
b = 20
```

we can write `a, b = 10, 20`

In fact, unpacking works with any iterable type

```
for e in 10, 20, 'hello'         → loop returns 10, 20, 'hello'
for e in 'XYZ'                   → loop returns 'X', 'Y', 'Z'
```

Simple Application of Unpacking

swapping values of two variables

```
a = 10
b = 20
```

→

```
b = 20
a = 10
```

"traditional" approach

```
tmp = a
a = b
b = tmp
```

using unpacking

```
a, b = b, a
```

this works because in Python, the entire RHS is evaluated first and completely

then assignments are made to the LHS

Unpacking Sets and Dictionaries

```
d = {'key1': 1, 'key2': 2, 'key3': 3}
```

`for e in d` → `e` iterates through the keys: `'key1'`, `'key2'`, `'key3'`

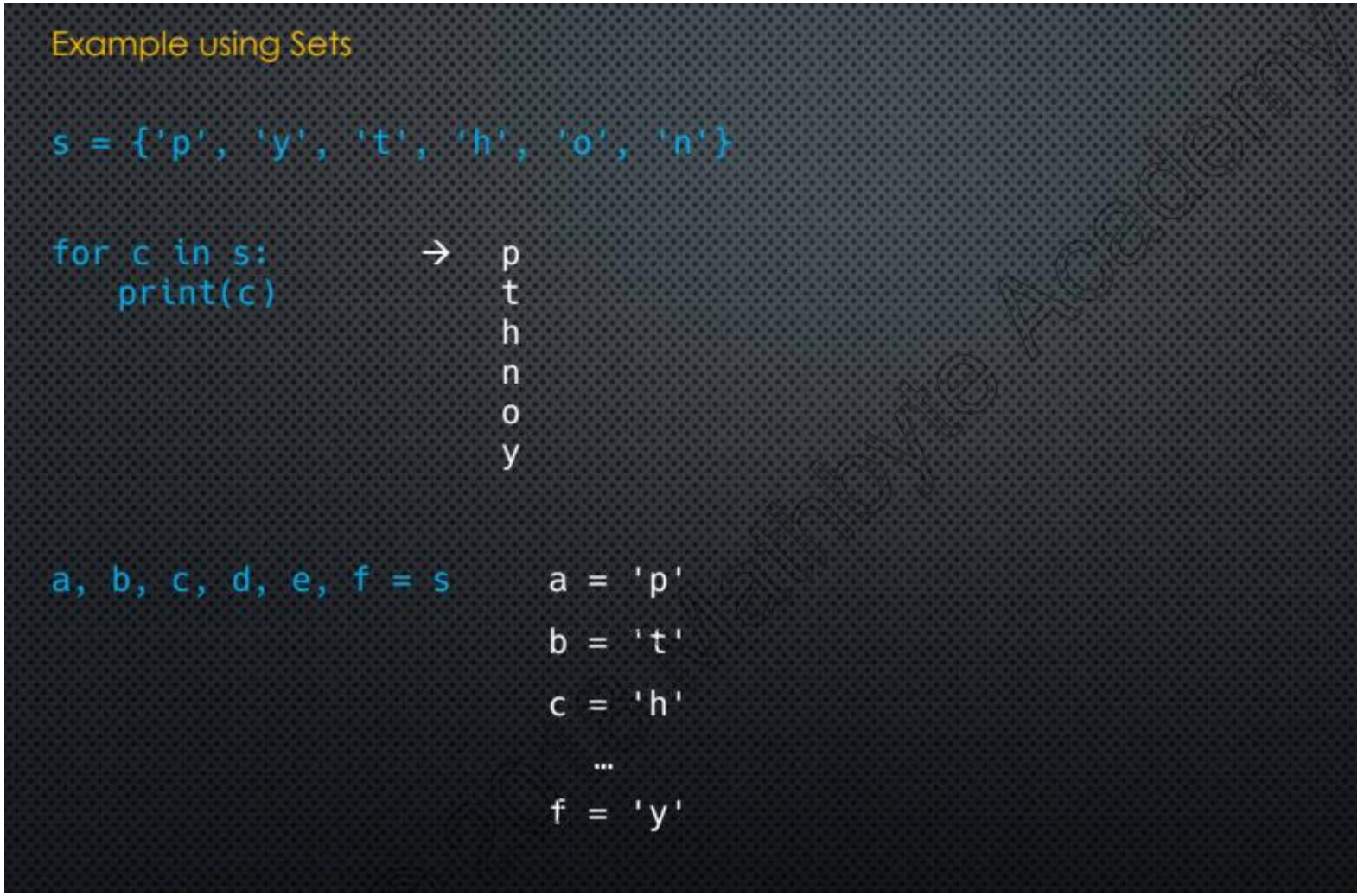
so, when unpacking `d`, we are actually unpacking the keys of `d`

```
a, b, c = d     → a = 'key1', b = 'key2', c='key3'
          or    → a = 'key2', b = 'key1', c='key3'
          or    → a = 'key3', b = 'key1', c='key2'
                etc...
```

Dictionaries (and Sets) are unordered types.

They can be iterated, but there is no guarantee the order of the results will match your literal!

In practice, we rarely unpack sets and dictionaries in precisely this way.


Example using Sets
s = {'p', 'y', 't', 'h', 'o', 'n'}
for c in s:
print(c)
→
p
t
h
n
o
y
a, b, c, d, e, f = s
a = 'p'
b = 't'
c = 'h'
…
f = 'y'


EVERYTHING IS AN OBJECT

Throughout this course, we'll encounter many data types.

- Integers (`int`)
- Booleans (`bool`)
- Floats (`float`)
- Strings (`str`)
- Lists (`list`)
- Tuples (`tuple`)
- Sets (`set`)
- Dictionaries (`dict`)
- None (`NoneType`)

We'll also see other constructs:

- Operators (`+, -, ==, is, …`)
- Functions
- Classes
- Types

and many more…

But the one thing in common with all these things, is that they are all objects (instances of classes)

- Functions (`function`)
- Classes (`class`) [not just instances, but the class itself]
- Types (`type`)

This means they all have a memory address!

As a consequence:

Any object can be assigned to a variable including functions…

Any object can be passed to a function including functions…

Any object can be returned from a function including functions…

```
def my_func():
    …
```

my_func → function state (0x1000)

id(my_func) → 0x1000

`my_func` is the name of the function

`my_func()` invokes the function


COMPARISON OPERATORS

Categories of Comparison Operators

- binary operators
- evaluate to a `bool` value

Identity Operations	`is`	`is not`			compares memory address – any type
Value Comparisons	`==`	`!=`			compares values – different types OK, but must be compatible
Ordering Comparisons	`<`	`<=`	`>`	`>=`	doesn't work for all types
Membership Operations	`in`	`not in`			used with iterable types

Numeric Types

We will examine other types, including iterables, later in this course

Value comparisons will work with all numeric types

Mixed types (except complex) in value and ordering comparisons is supported

Note: Value equality operators work between floats and Decimals, but as we have seen before, using value equality with floats has some issues!

```
10.0 == Decimal('10.0')   → True

0.1 == Decimal('0.1')     → False

Decimal('0.125') == Fraction(1, 8)  → True

True == 1    → True

True == Fraction(3, 3)  → True
```

Ordering Comparisons

Again, these work across all numeric types, except for complex numbers

```
1 < 3.14 → True

Fraction(22, 7) > math.pi    → True

Decimal('0.5') <= Fraction(2, 3)    → True

True < Decimal('3.14')  → True

Fraction(2, 3) > False    → True
```

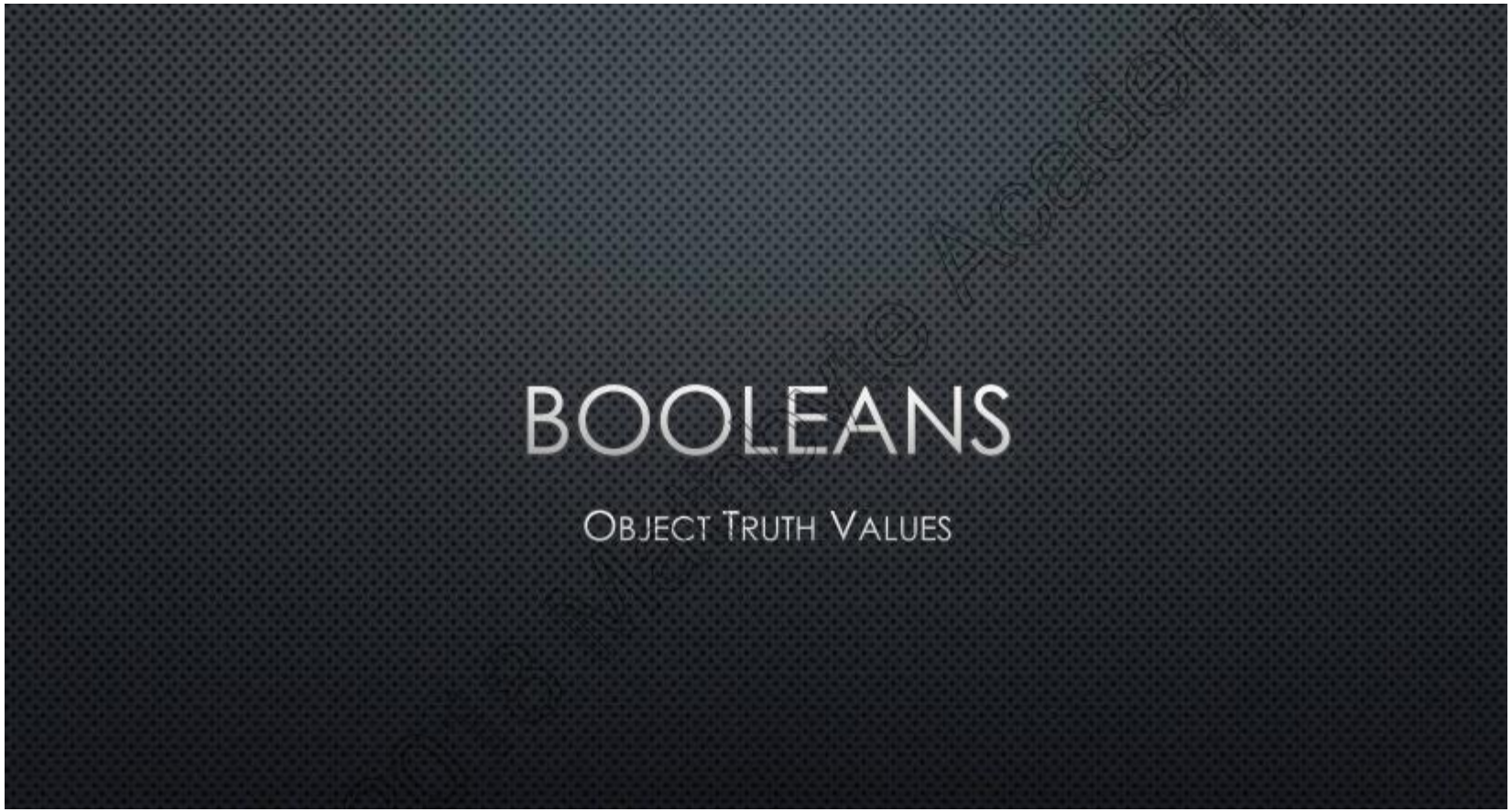
Chained Comparisons

```
a == b == c   →   a == b and b == c

a < b < c     →   a < b and b < c

1 == Decimal('1.0') == Fraction(1,1)     → True

1 == Decimal('1.5') == Fraction(3, 2)    → False

1 < 2 < 3     → 1 < 2 and 2 < 3   → True

1 < math.pi < Fraction(22, 7)
   → 1 < math.pi and math.pi < Fraction(22, 7)
   → True
```

Chained Comparisons

```
a < b > c     → a < b and b > c

5 < 6 > 2     → 5 < 6 and 6 > 2     → True

5 < 6 > 10    → 5 < 6 and 6 > 10    → False

a < b < c < d     → a < b and b < c and c < d

1 < 2 < 3 < 4     → 1 < 2 and 2 < 3 and 3 < 4     → True

1 < 10 > 4 < 5    → 1 < 10 and 10 > 4 and 4 < 5   → True
```

```
if my_min == cnt < val > other <= my_max not in lst:
    # do something
```


BOOLEANS
OBJECT TRUTH VALUES

Objects have Truth Values

All objects in Python have an associated truth value

We already saw this with integers (although to be fair, `bool` is a subclass of `int`)

But this works the same for any object

In general, the rules are straightforward

Every object has a `True` truth value, except:

- `None`
- `False`
- `0` in any numeric type (e.g. `0`, `0.0`, `0+0j`, ...)
- empty sequences (e.g. `list`, `tuple`, `string`, ...)
- empty mapping types (e.g. `dictionary`, `set`, ...)
- custom classes that implement a `__bool__` or `__len__` method that returns `False` or `0`

which have a `False` truth value

Under the hood

Classes define their truth values by defining a special instance method:

`__bool__(self)` (or `__len__`)

Then, when we call `bool(x)` Python will actually executes `x.__bool__()`

or `__len__` if `__bool__` is not defined

if neither is defined, then `True`

Example: Integers

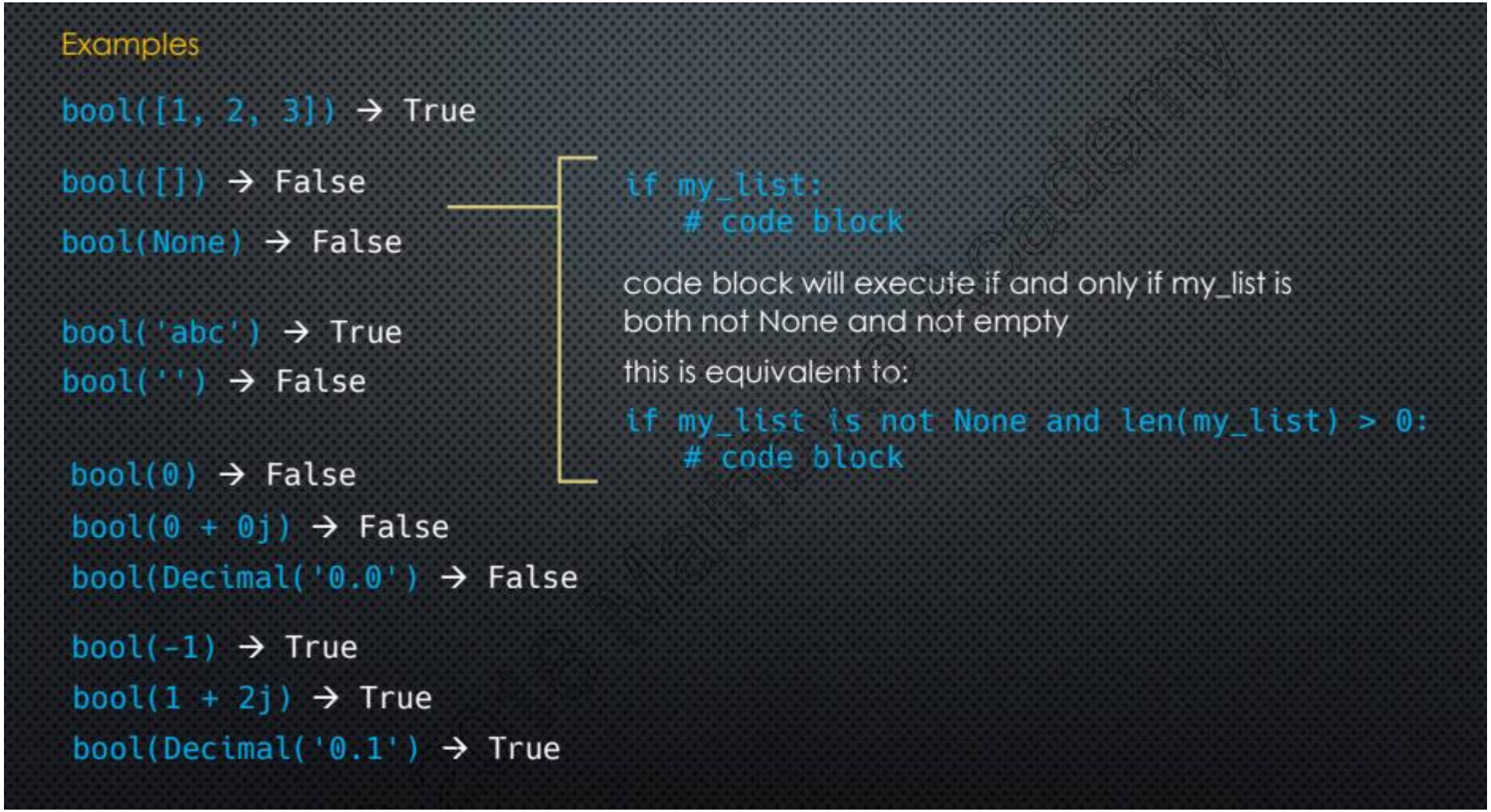
```
def __bool__(self):
    return self != 0
```

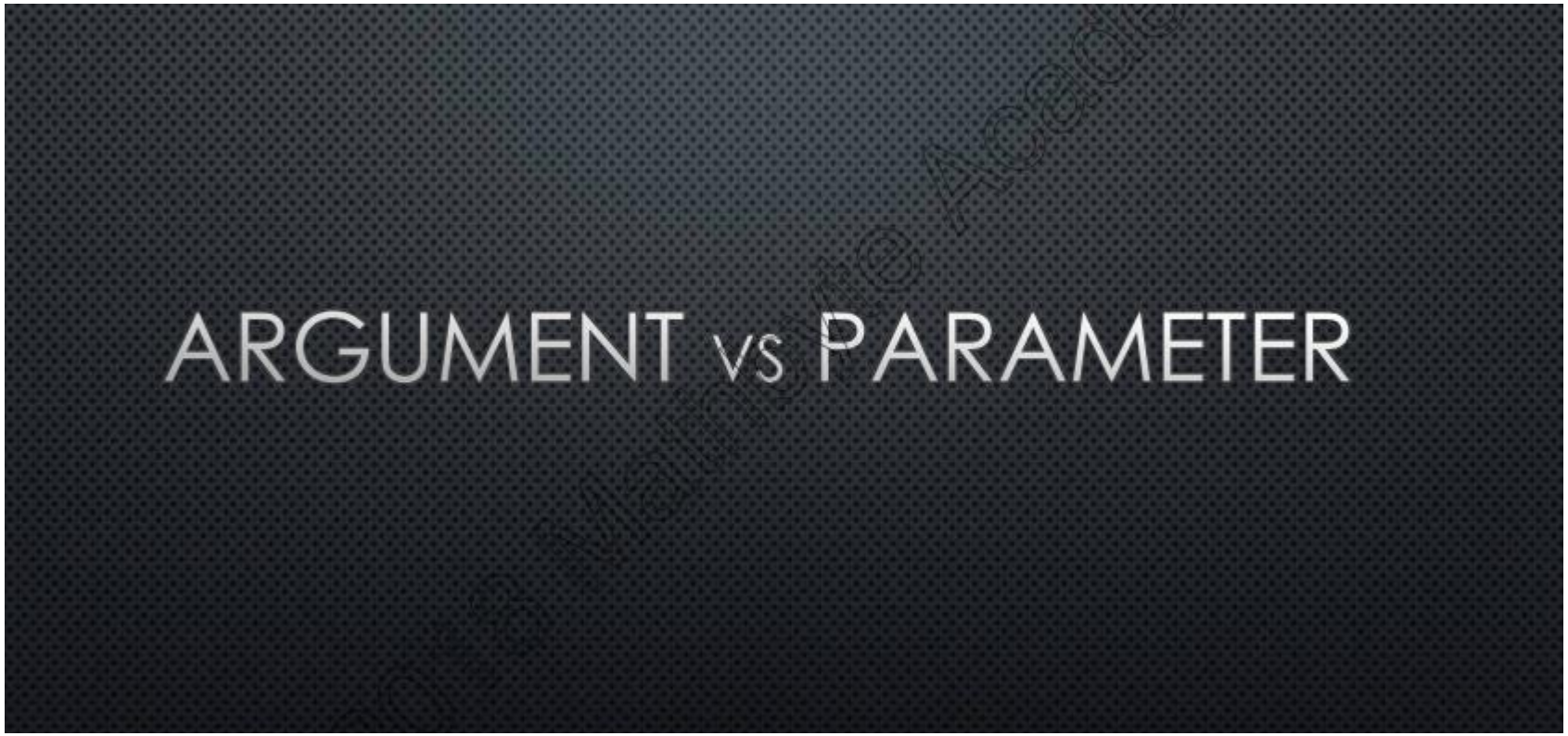
When we call `bool(100)` Python actually executes `100.__bool__()`

and therefore returns the result of `100 != 0` which is `True`

When we call `bool(0)` Python actually executes `0.__bool__()`

and therefore returns the result of `0 != 0` which is `False`


Examples
bool([1, 2, 3]) → True
bool([]) → False
bool(None) → False
bool('abc') → True
bool('') → False
bool(0) → False
bool(0 + 0j) → False
bool(Decimal('0.0') → False
bool(-1) → True
bool(1 + 2j) → True
bool(Decimal('0.1') → True
if my_list:
# code block
code block will execute if and only if my_list is both not None and not empty
this is equivalent to:
if my_list is not None and len(my_list) > 0:
# code block


ARGUMENT vs PARAMETER

Semantics!

```
def my_func(a, b):
    # code here
```

In this context, a and b are called parameters of my_func

Also note that a and b are variables, local to my_func

When we call the function:

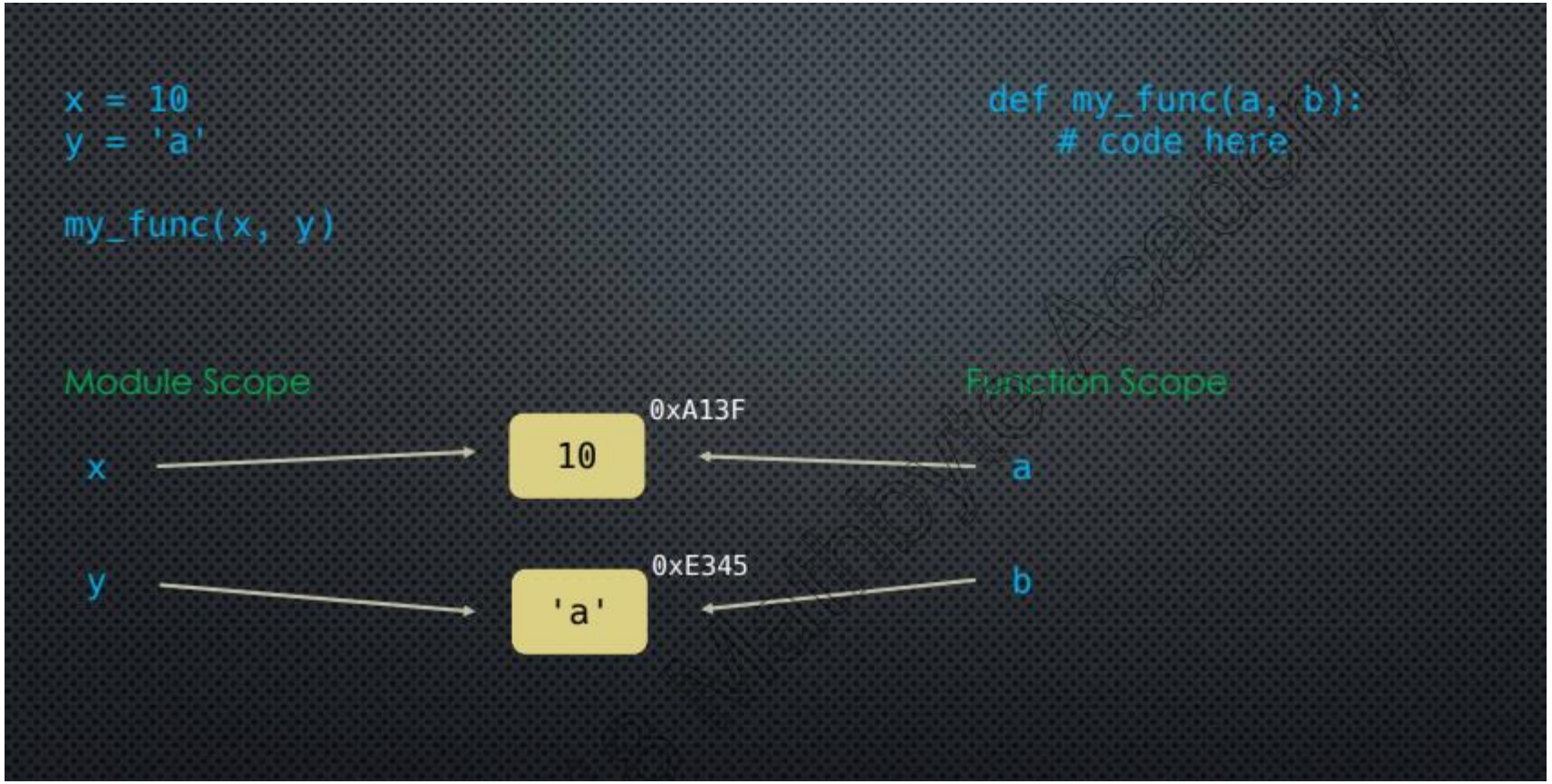
```
x = 10
y = 'a'

my_func(x, y)
```

x and y are called the arguments of my_func

Also note that x and y are passed by reference

i.e. the memory addresses of x and y are passed

It's OK if you mix up these terms – everyone will understand what you mean!


x = 10
y = 'a'
my_func(x, y)
def my_func(a, b):
# code here
Module Scope
Function Scope
0xA13F
10
x
a
0xE345
'a'
y
b