



WELCOME!





python



Hi, I'm Colt

I've lead in-person programming bootcamps in the Bay Area for a decade.

I've been a lead-instructor and curriculum director for multiple bootcamps including Galvanize and Rithm School where we guarantee students jobs.

I've taught millions of students to code online and was selected as the Best New Instructor on Udemy!



NEVER KNOW WHAT TO PUT UP HERE

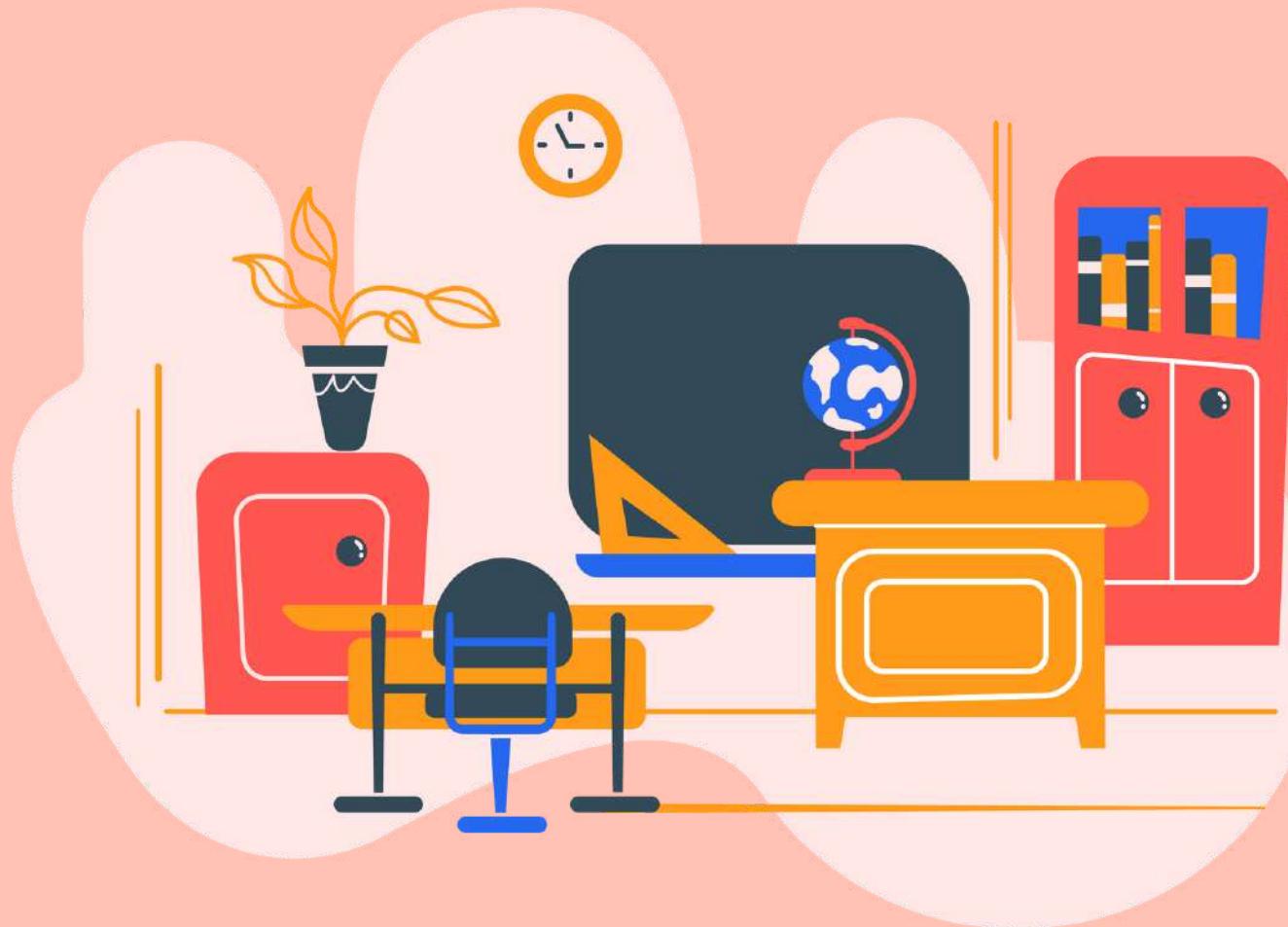
My students work at
companies including...

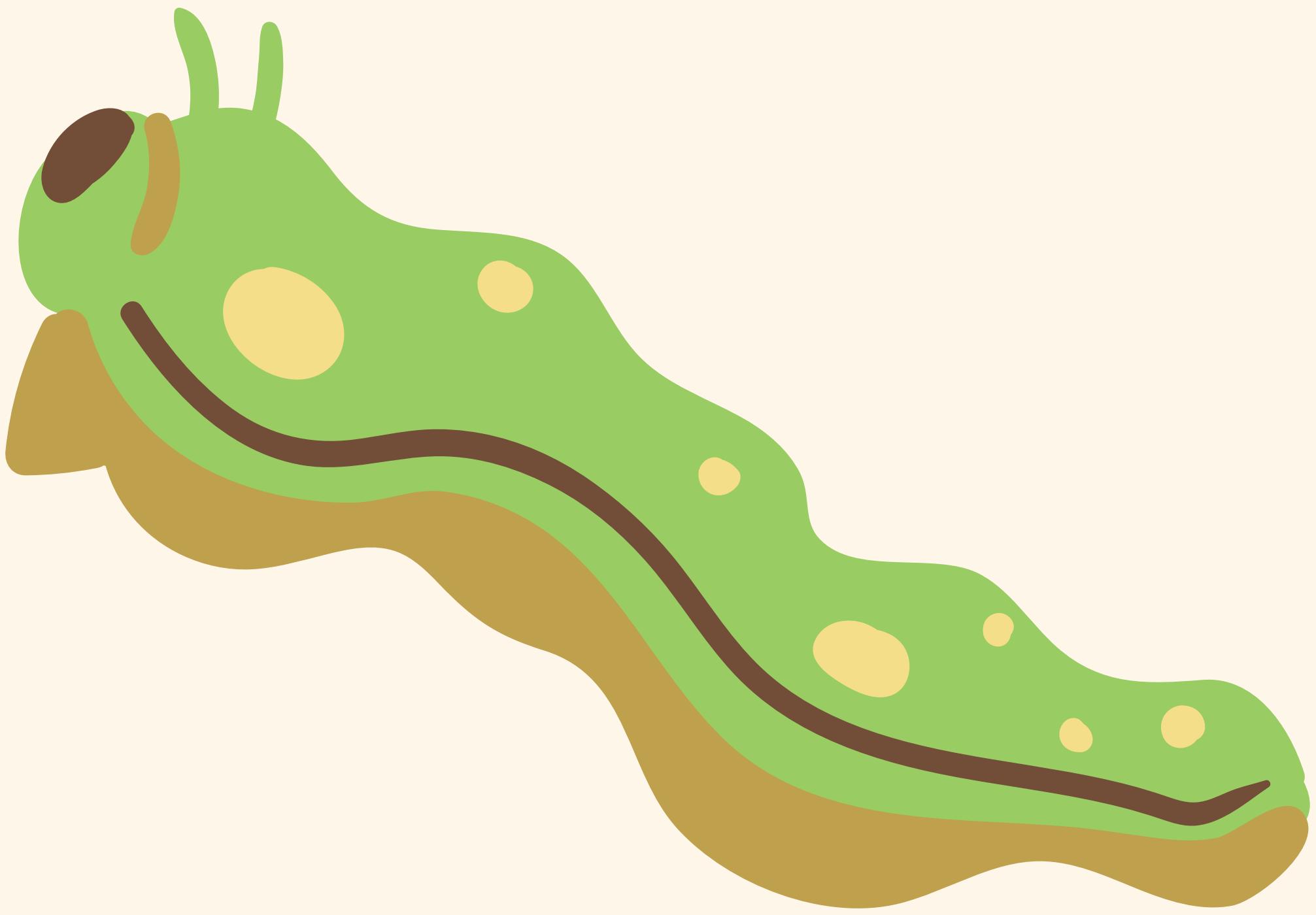


What really matters...

Is that I've spent years trying to perfect teaching
programming to beginners

I've learned how to keep students engaged from my
years of in-person teaching experience.





This Course Is...

- A fast, effective pathway to learn Python
- Full of exercises/challenges/quizzes
- Made for normal people who can't stand 20 minute videos. Average video is 4 minutes.
- An ideal on-ramp to data science, ML, or web development courses.



WHAT THIS COURSE IS NOT



This course isn't

- A 60 hour Python encyclopedia course that covers every possible feature of Python
- An advanced level Python course for experts
- A web development, data science, or machine learning oriented course



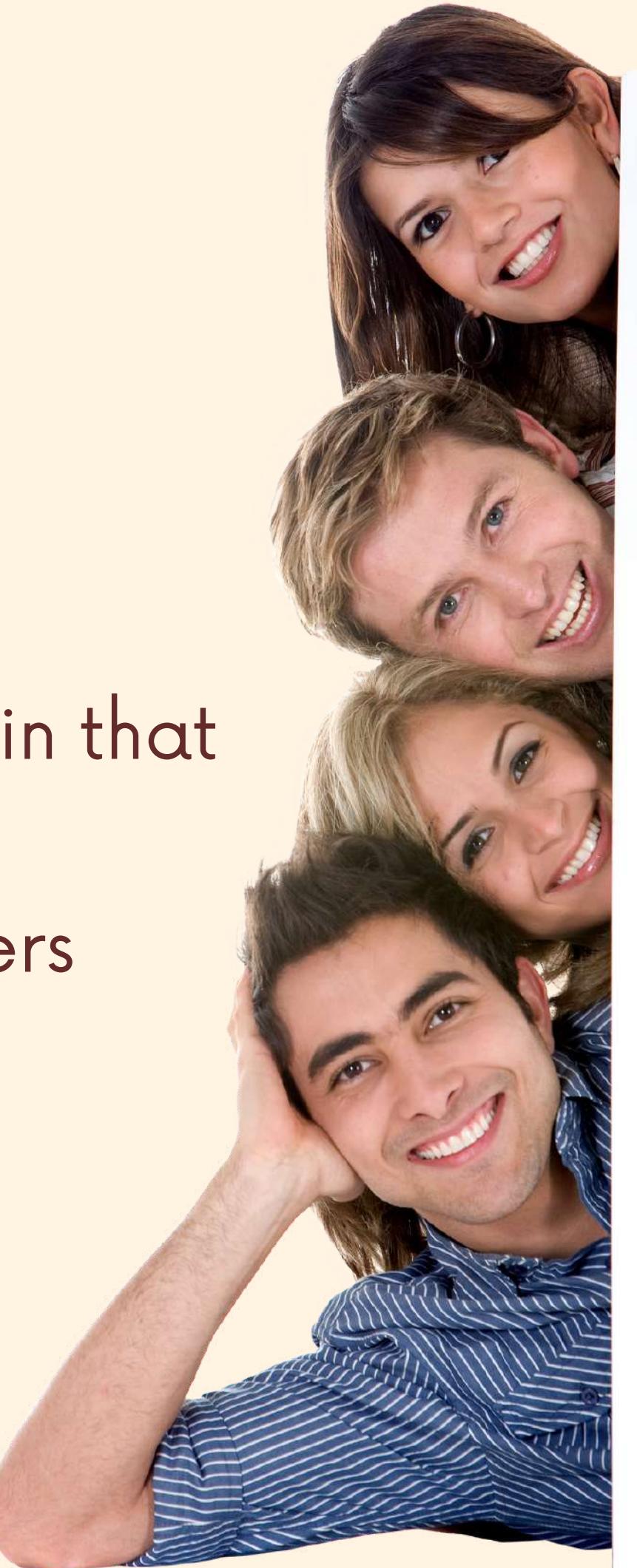
High Demand

There are more Python jobs today than ever before, and salaries are at an all-time high. If you could only pick one language to learn, it should be Python.



Huge Community

Python has been around for 30+ years now, and in that time a massive, supportive online community has formed. You can always get help and find answers (relatively) easily.



Libraries

There are thousands of Python libraries, frameworks and tools available. From django to matplotlib to scikitlearn, there's usually something for every need.



Easy To Learn

Python is (in my opinion) the easiest programming language to learn. It has far fewer quirks and oddities than languages like JavaScript.



Versatility

You can use Python for simple scripting tasks.

You can use it to build huge web applications.

You can use it for data processing and analysis.

And of course, you can use it for machine learning!



What can we build with python



Game Dev

Hugely popular games including the Sims 4, Civilization, EVE Online, and many others are built using Python!



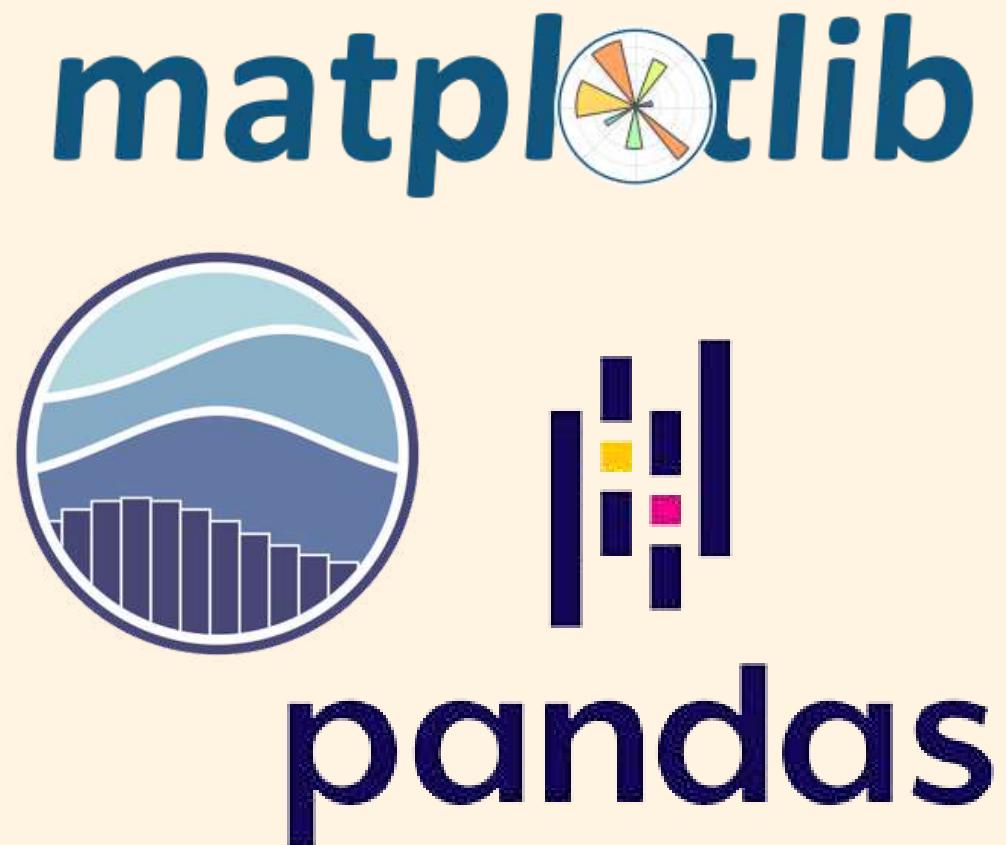
Web Dev

Companies ranging from tiny startups to massive unicorns use Python to build web applications.



Data Stuff

Python is hugely popular in the world of data science and data analysis. Python tools like numpy, pandas, and matplotlib are widely used.



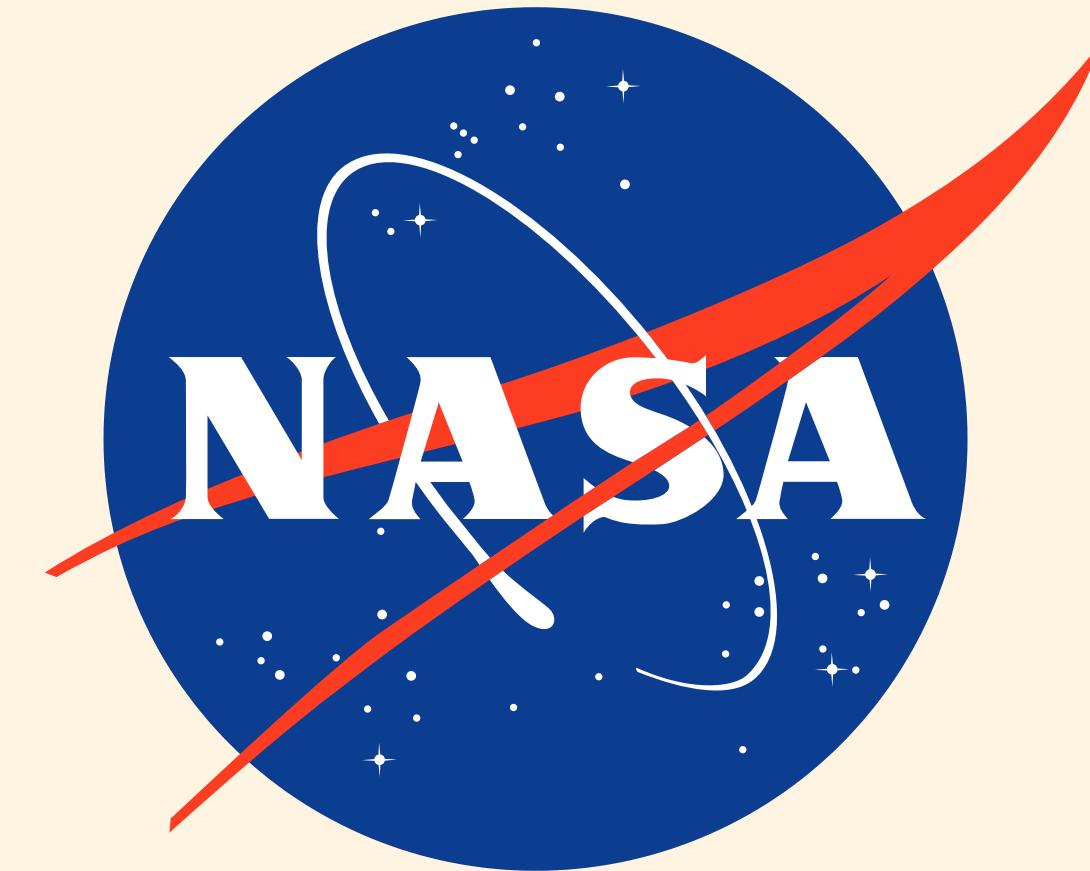
Machine Learning

Python is "the" language for machine learning. Tools including TensorFlow, Scikitlearn, OpenCV, and NLTK require Python knowledge

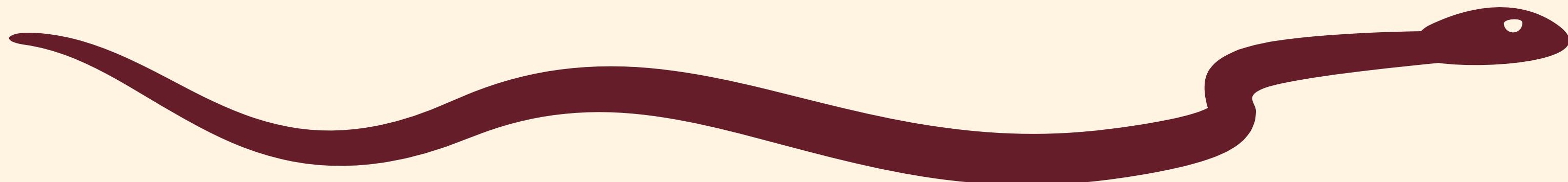


Science

Python is used across academia and the science world, from biology research labs to NASA engineering teams

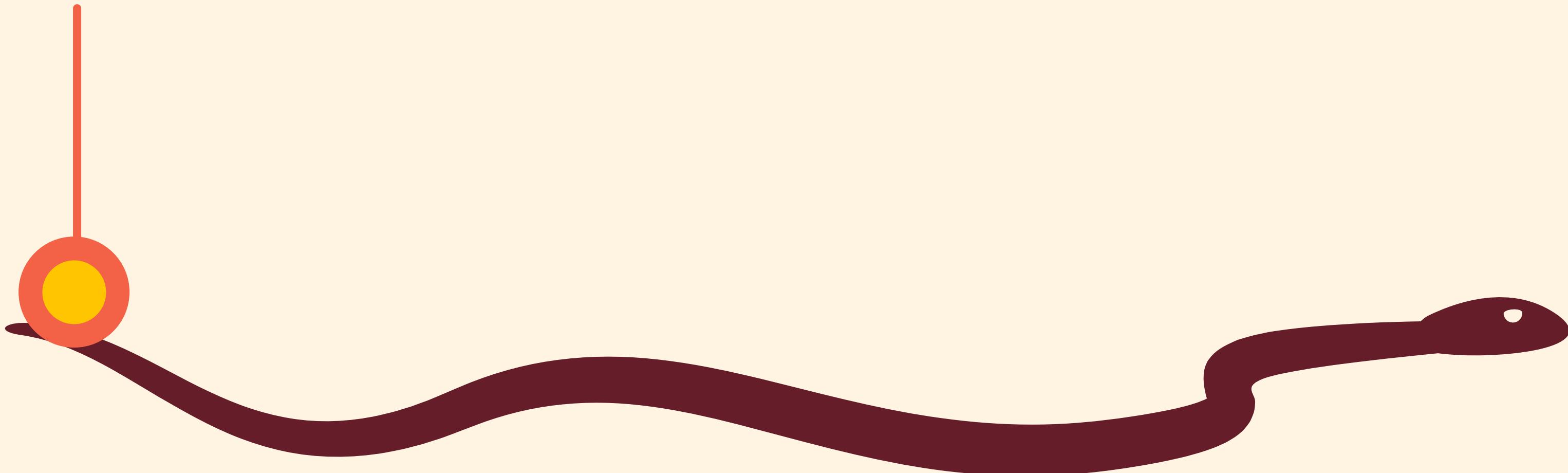


A Tiny Little Bit Of Python History



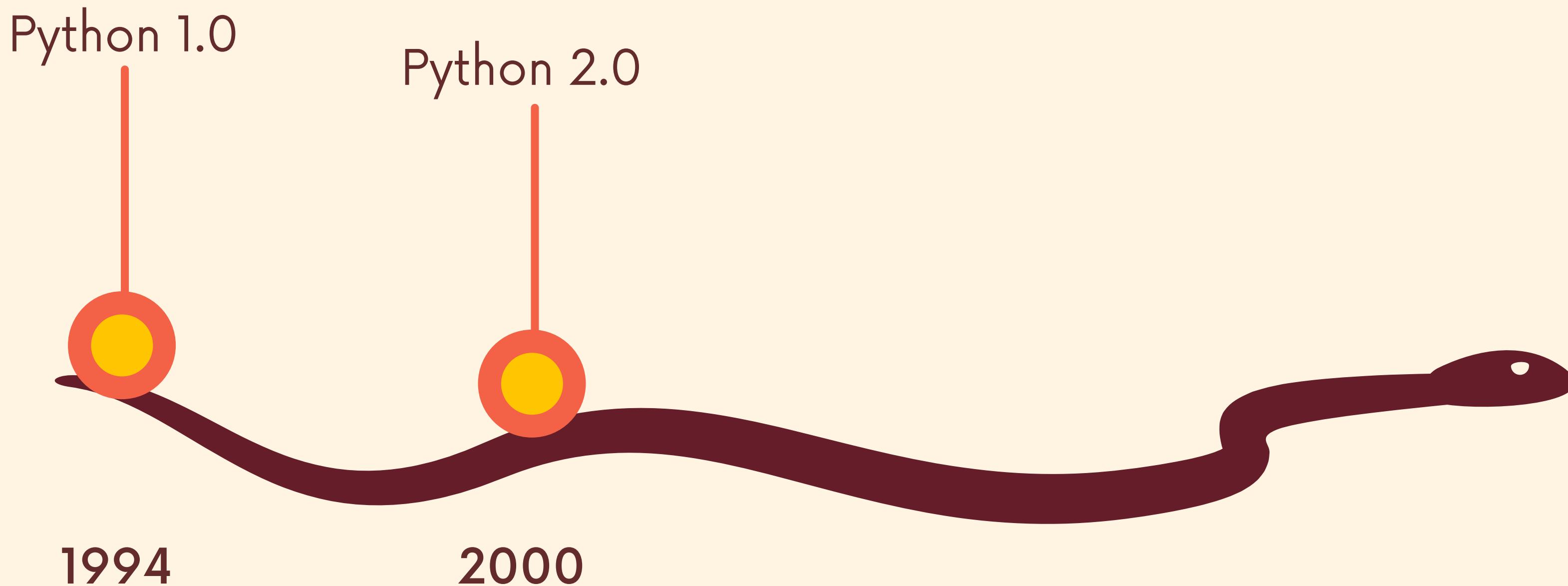
A Tiny Little Bit Of Python History

Python 1.0

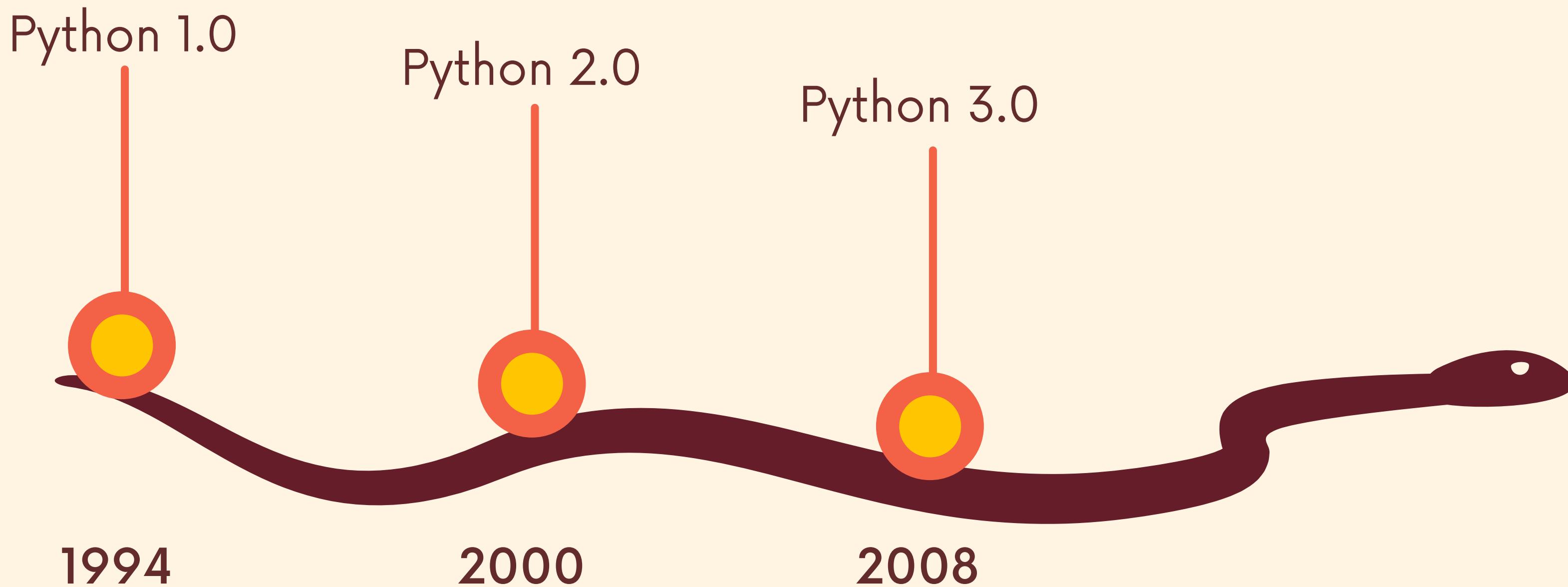


1994

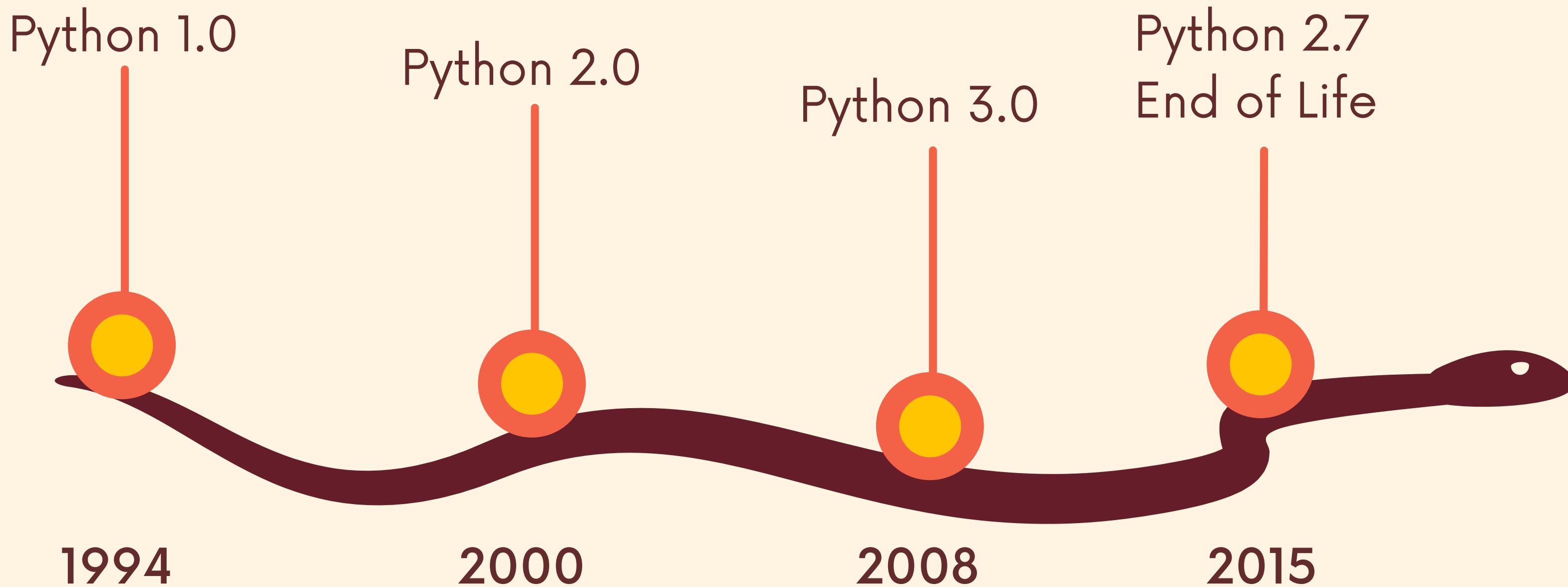
A Tiny Little Bit Of Python History



A Tiny Little Bit Of Python History



A Tiny Little Bit Of Python History



A Tiny Little Bit Of

Python History

Python 1.0

Python 2.0

Python 3.0

Python 2.7
End of Life

Python 2
End of Life

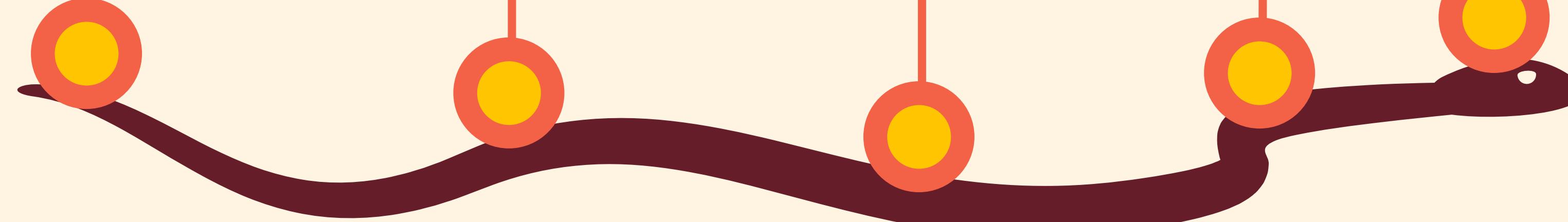
1994

2000

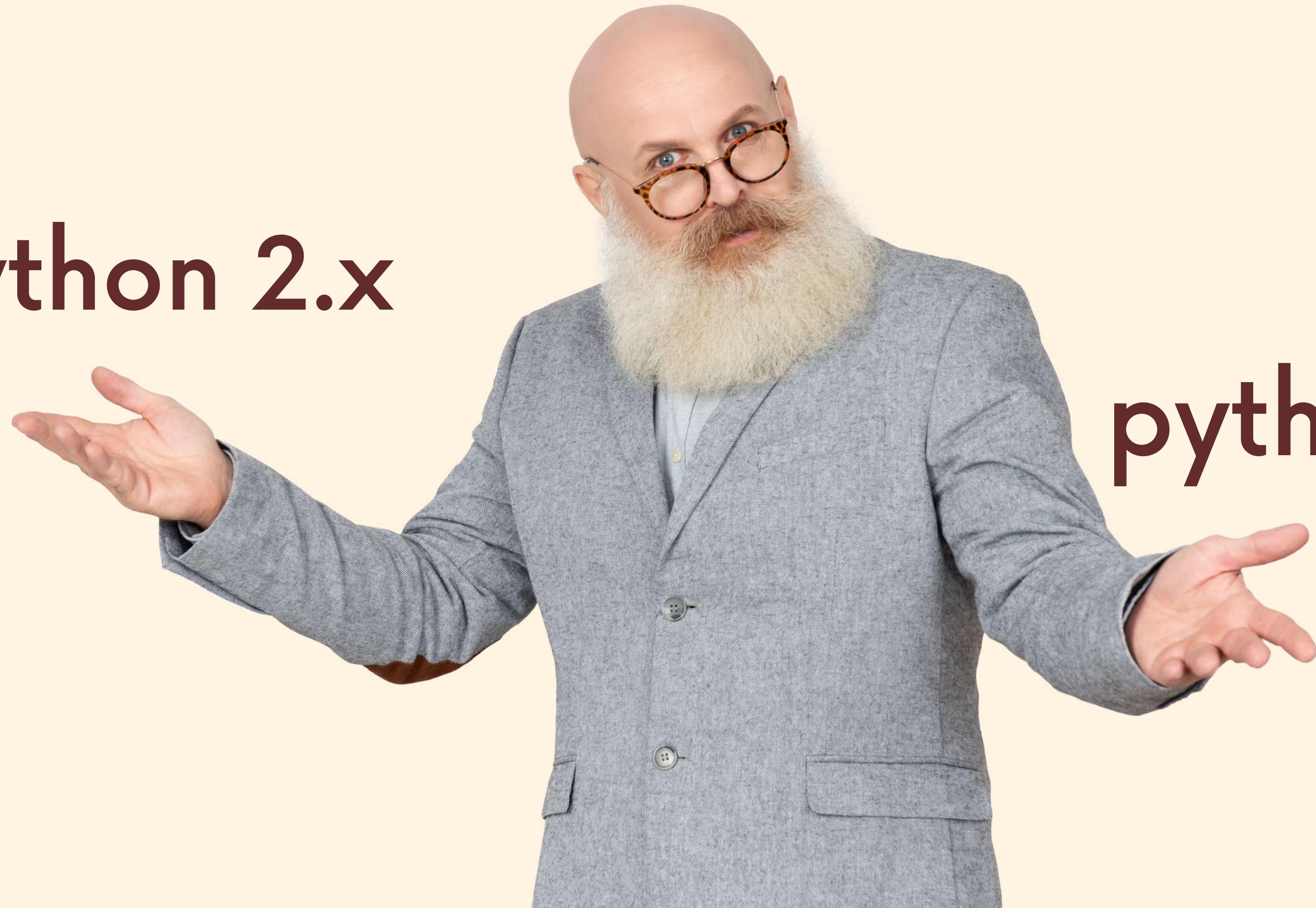
2008

2015

2020



python 2.x

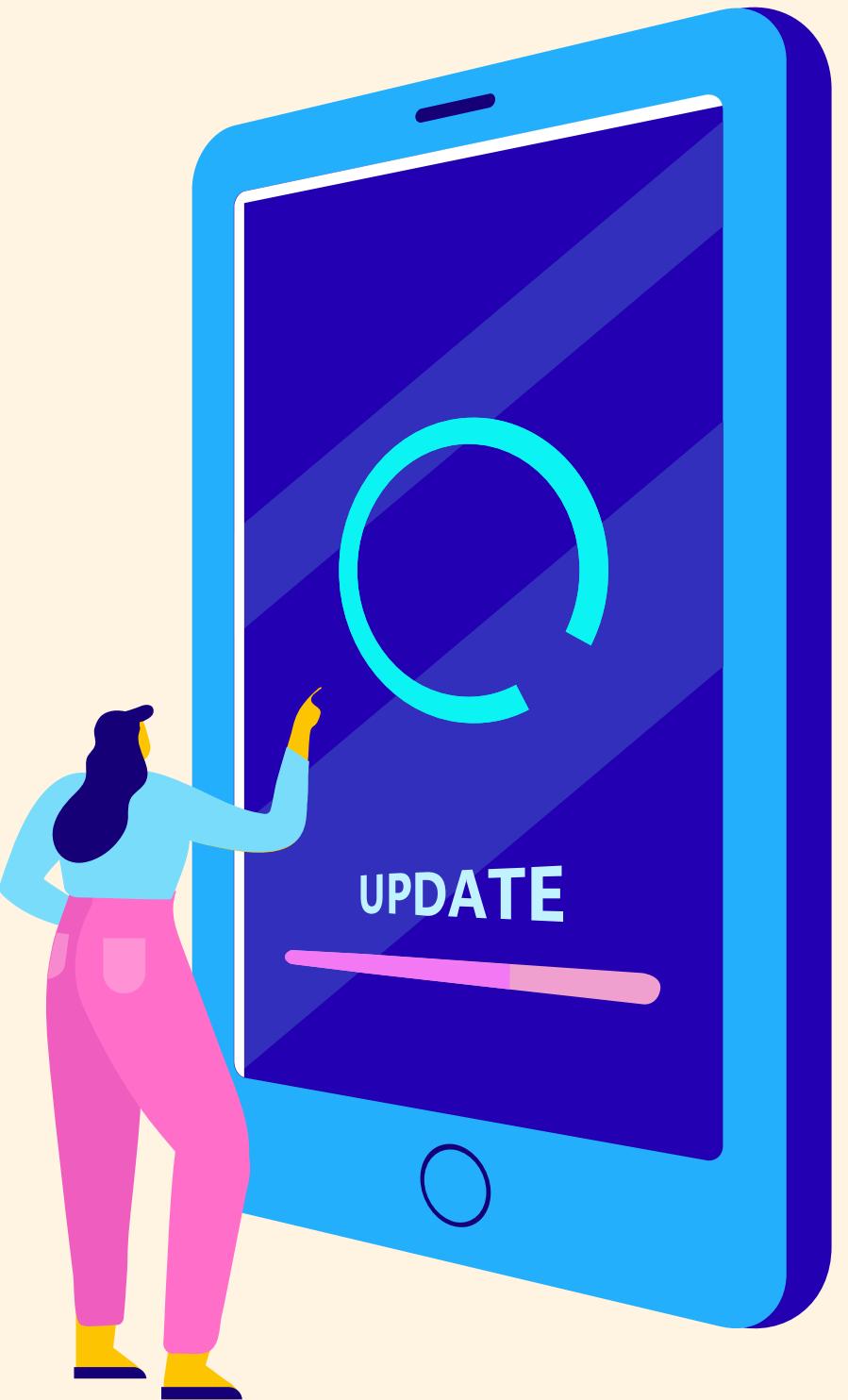


python 3.x

python 2 vs. 3

Python 3 was a major change

- Not backwards compatible with python 2
- It included some syntax changes
- Featured major changes under the hood
- It took 10+ years for Python 2 to be officially declared dead



python 2.x

```
>>> print "Hello"  
Hello
```

```
>>> 5/2  
2
```

python 3.x

```
>>> print("Hello")  
Hello
```

```
>>> 5/2  
2.5
```

Who Cares?

At this point, there is no real reason to learn 2.x, but it's worth knowing the history because...

- Lots of the tutorials online are for 2.x
- Your computer may have 2.x pre-installed
- Some dead libraries/tools still haven't been updated to work with 3.x





Installation

(it's actually not bad!)

Running Python

There are two main ways we can run Python code



Interactively



From A File



Interactive Python

In interactive mode, any code you enter is immediately run.

Python executes each line as you enter it and then displays the output.

Great for learning and trying things out, but not great for "real" applications



```
>>> 1 + 2
3
>>> "heyyoo" * 3
' heyyooheyyooheyyoo'
>>>
```

Python Scripts

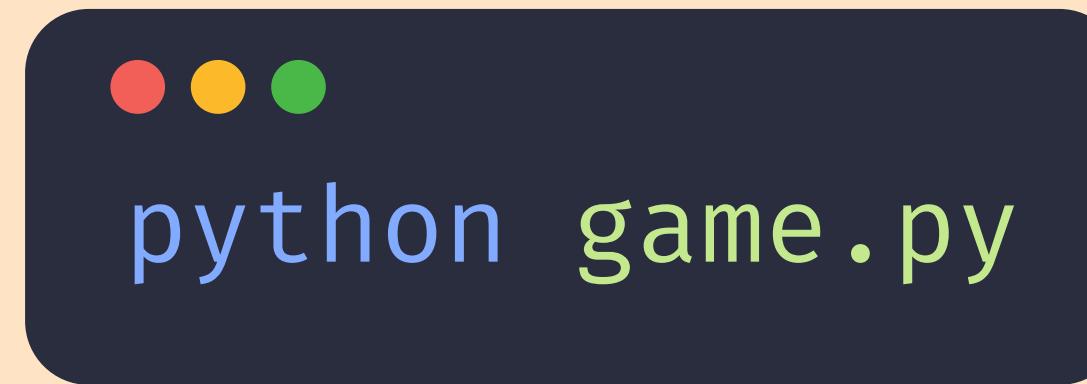
Alternatively, you can write python code in a file.



game.py



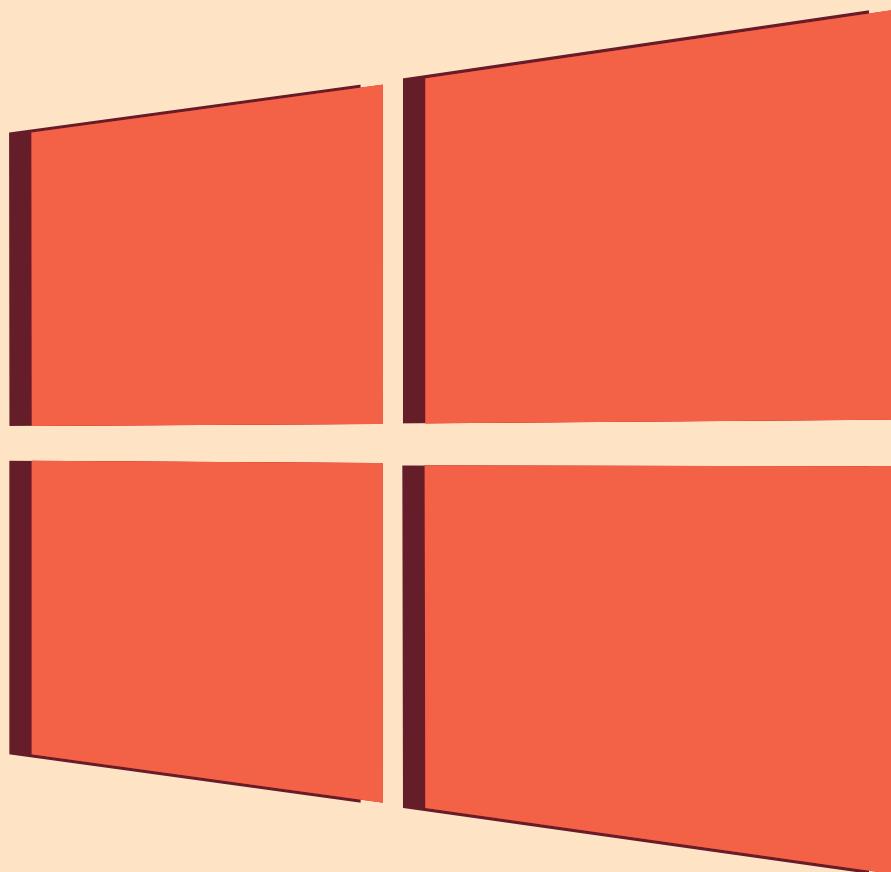
And then you can separately execute that script with Python



The code only runs when you manually tell it to.



Mac
Install



Windows Install



Visual Studio Code

The "I want to
get started right
now" option



replicon

**Numbers,
Variables,
And More**



Write Your Review



Click on a star to change your rating 1 - 5, where 5 = great! and 1 = really bad

Your Review:

Your Reivew

999 Characters remaining

Your Info:

Name:

Name

Email:

Email

I Agree to the Terms blah blah blah

Submit

Basic Data Types

Strings

Integers

Booleans

FLOATS

Data Types

Strings

Integers

Frozenset

Booleans

FLOATS

Bytes

Dictionary

Complex

Range

List

Tuple

Set

Integers

- Whole numbers only
- Positive or Negative
- No Decimal Points!

Integers

- Whole numbers only
- Positive or Negative
- No Decimal Points!

9

Integers

- Whole numbers only
- Positive or Negative
- No Decimal Points!

9

378

Integers

- Whole numbers only
- Positive or Negative
- No Decimal Points!

9

378

-21

Integers

- Whole numbers only
- Positive or Negative
- No Decimal Points!

9

378

-21

Floats

- Written With a Decimal Point
- Positive or Negative

Integers

- Whole numbers only
- Positive or Negative
- No Decimal Points!

9

378

-21

Floats

- Written With a Decimal Point
- Positive or Negative

1.5

Integers

- Whole numbers only
- Positive or Negative
- No Decimal Points!

9

378

-21

Floats

- Written With a Decimal Point
- Positive or Negative

1.5

9.99

Integers

- Whole numbers only
- Positive or Negative
- No Decimal Points!

9

378

-21

Floats

- Written With a Decimal Point
- Positive or Negative

1.5

9.99

-2.0

INT

1234987234321

+8

0

378

1_000_000_000

-99

-8363247238498

1782

FLOAT

1.3333333

0.5

+8.1

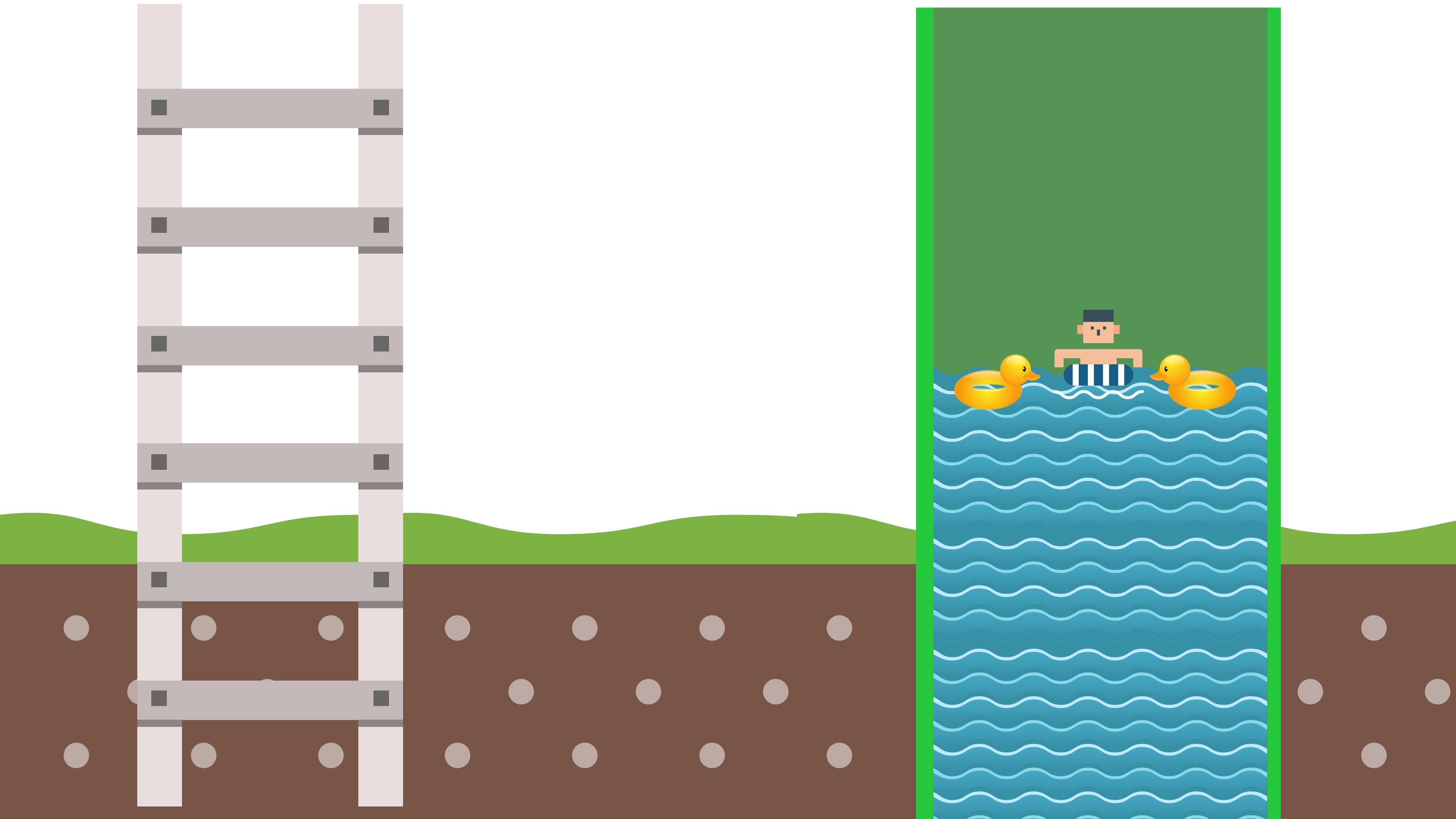
-.2

0.0

0.8372984732894733

1.234987234321e12

04.0



Ints



1_000

777



1,000

0777

FLOATS



1_000.55

1.234e12



1,000.55

1.234 e12

Operators

Operators are special characters in Python that perform operations on value(s). Below are some of the most common:

+	*	>	<=	and	is	=	*=
-	/	<	==	or	in	+=	/=
**	%	>=	!=	not	!=	-=	=

Order of Operations

Parentheses

()

Multiplication and Division

*

/

//

Addition and Subtraction

+

-

Integer Division

//

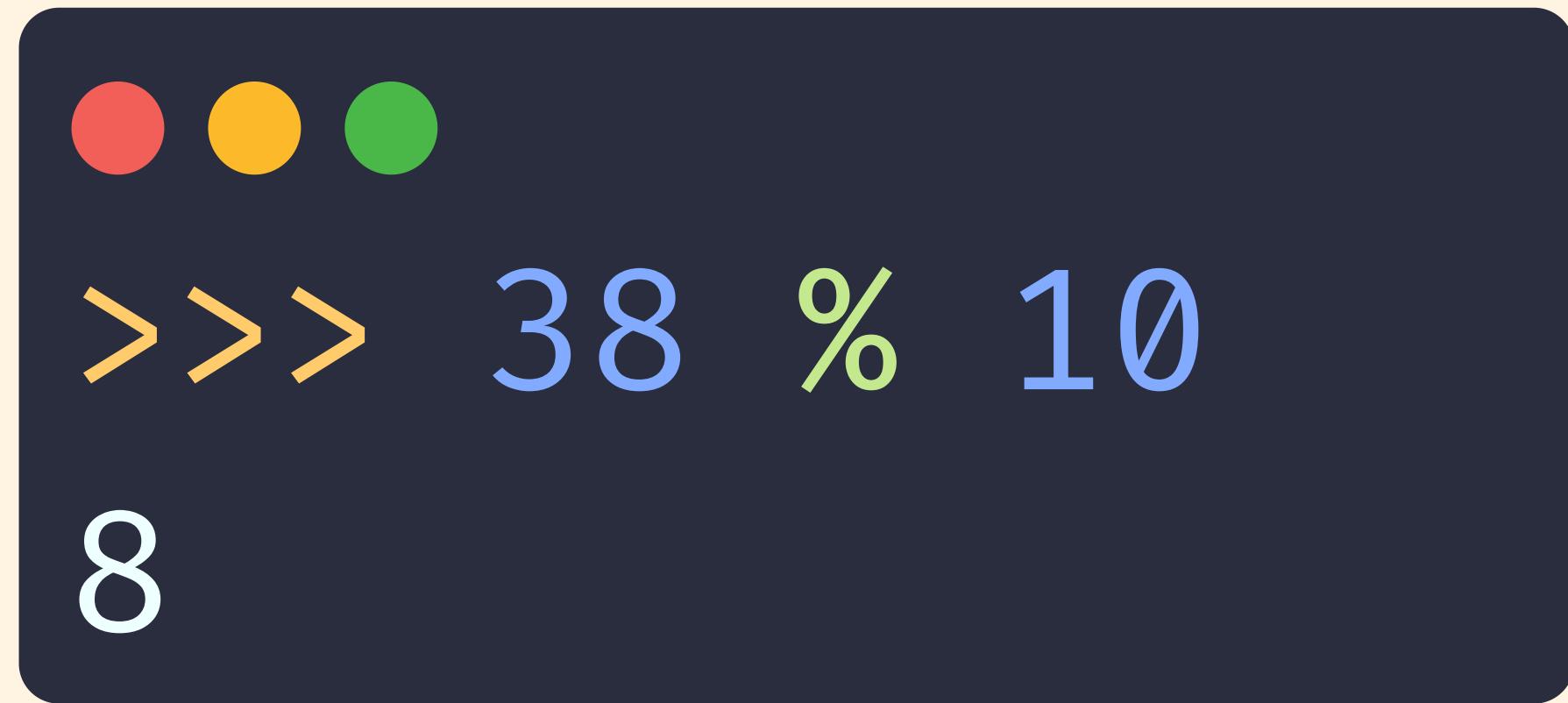
Exponentiation

* *

Modulo

%

Modulo



10 goes into 38...
3 times, with a remainder of 8

Integer Division



```
>>> 10 // 3
```

```
3
```



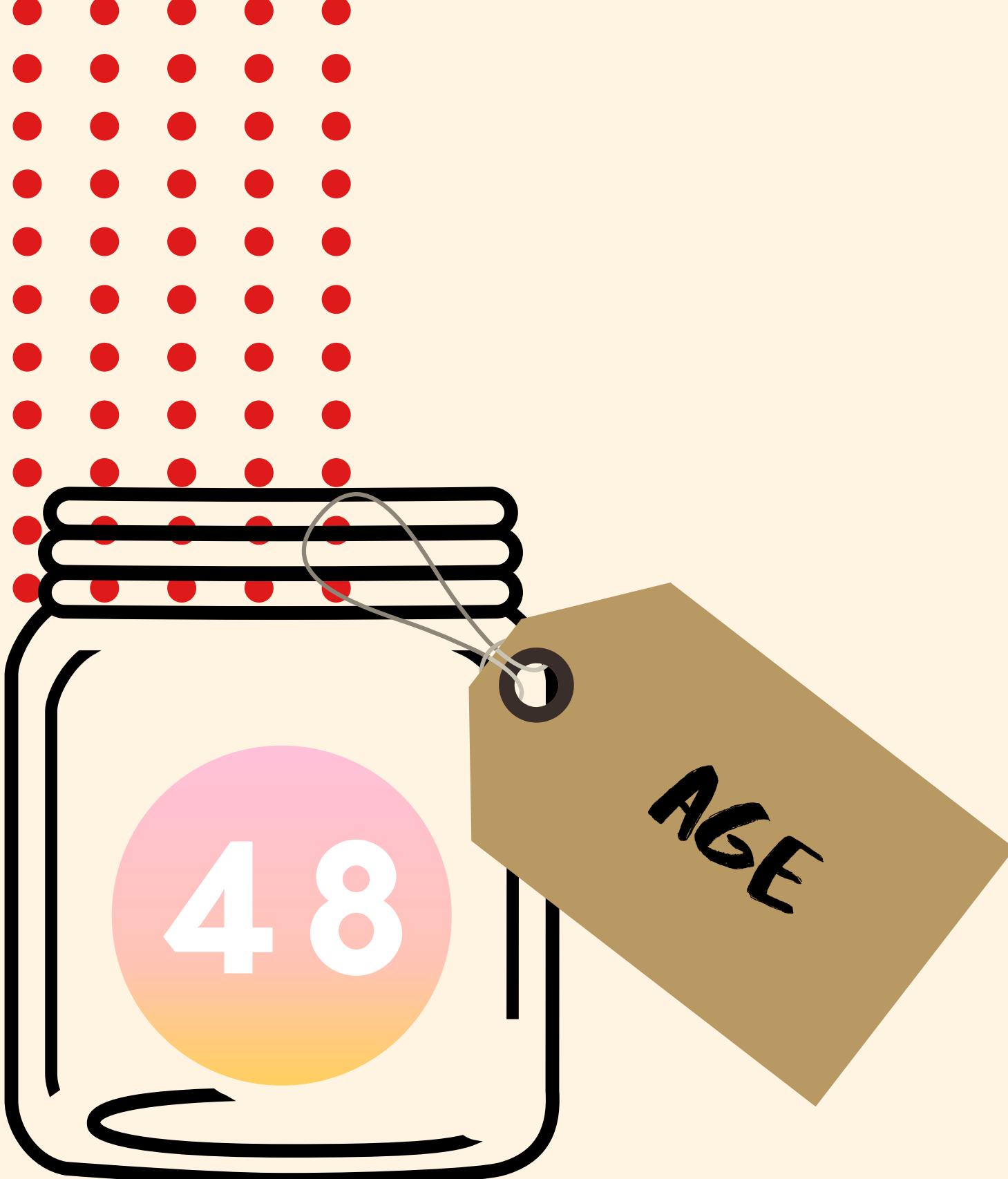
```
>>> -10 // 3
```

```
-4
```

Comments

```
# this never runs
```

Python will ignore any lines starting with the # symbol



Variables

**VARIABLES ARE LIKE
LABELS FOR VALUES**

We can store a value and give it a name so that we can:

- Refer back to it later
- Use that value to do...stuff
- Change it later on



Potential Variables

num_guesses → 0

target_word → "knoll"

max_guesses → 6

guesses →

correct_letters →

game_over → False



Potential Variables

num_guesses → 1

target_word → "knoll"

max_guesses → 6

guesses → "irate"

correct_letters →

game_over → False



Potential Variables

num_guesses → 2

target_word → "knoll"

max_guesses → 6

guesses → ""irate",
"sound"

correct_letters → "o", "n"

game_over → False



Potential Variables

num_guesses	→	3
target_word	→	"knoll"
max_guesses	→	6
guesses	→	"irate", "sound", "knoll"
correct_letters	→	"o", "n", "k", "l"
game_over	→	True

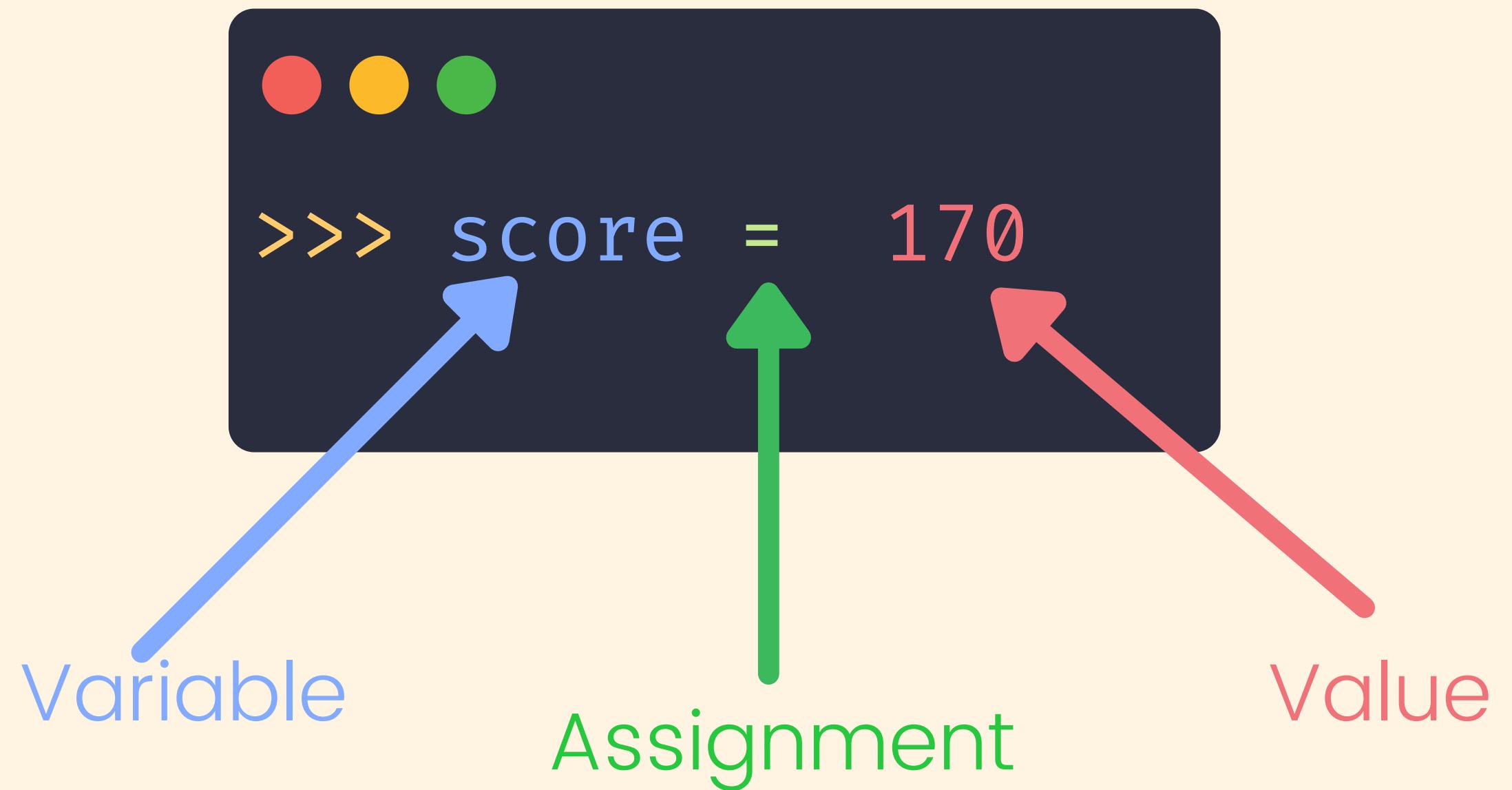


Variables



```
>>> age = 48
```







variable123

first_name

player_1



123variable

first name

False

def



I

O

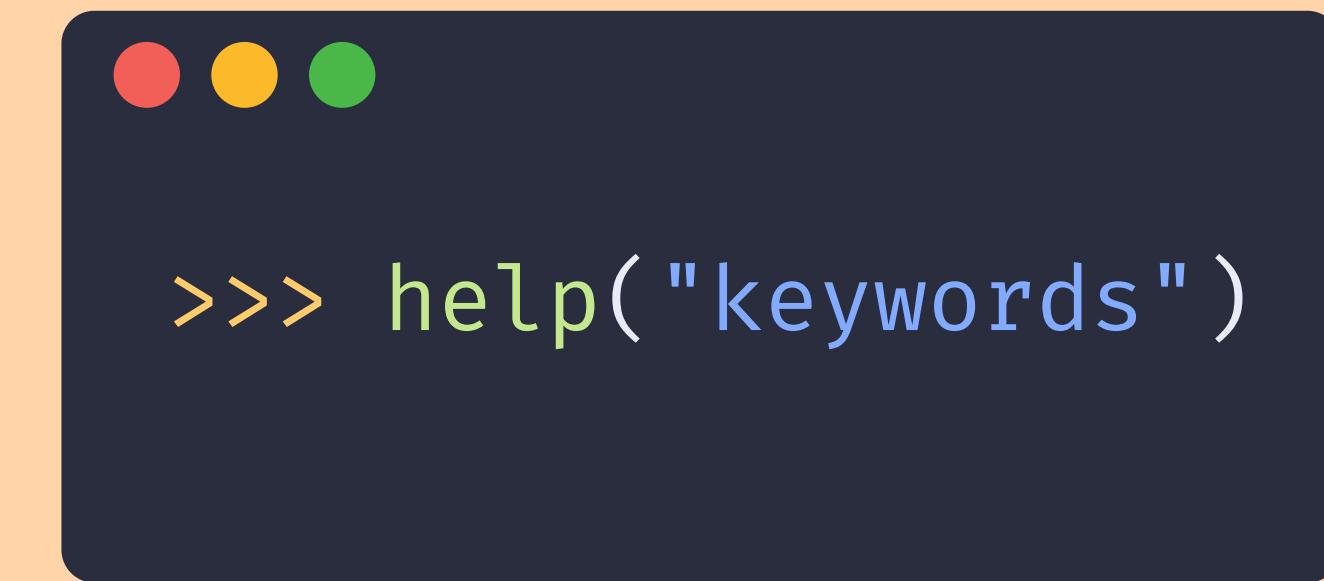
x

FirstName

FIRSTNAME



Python Keywords



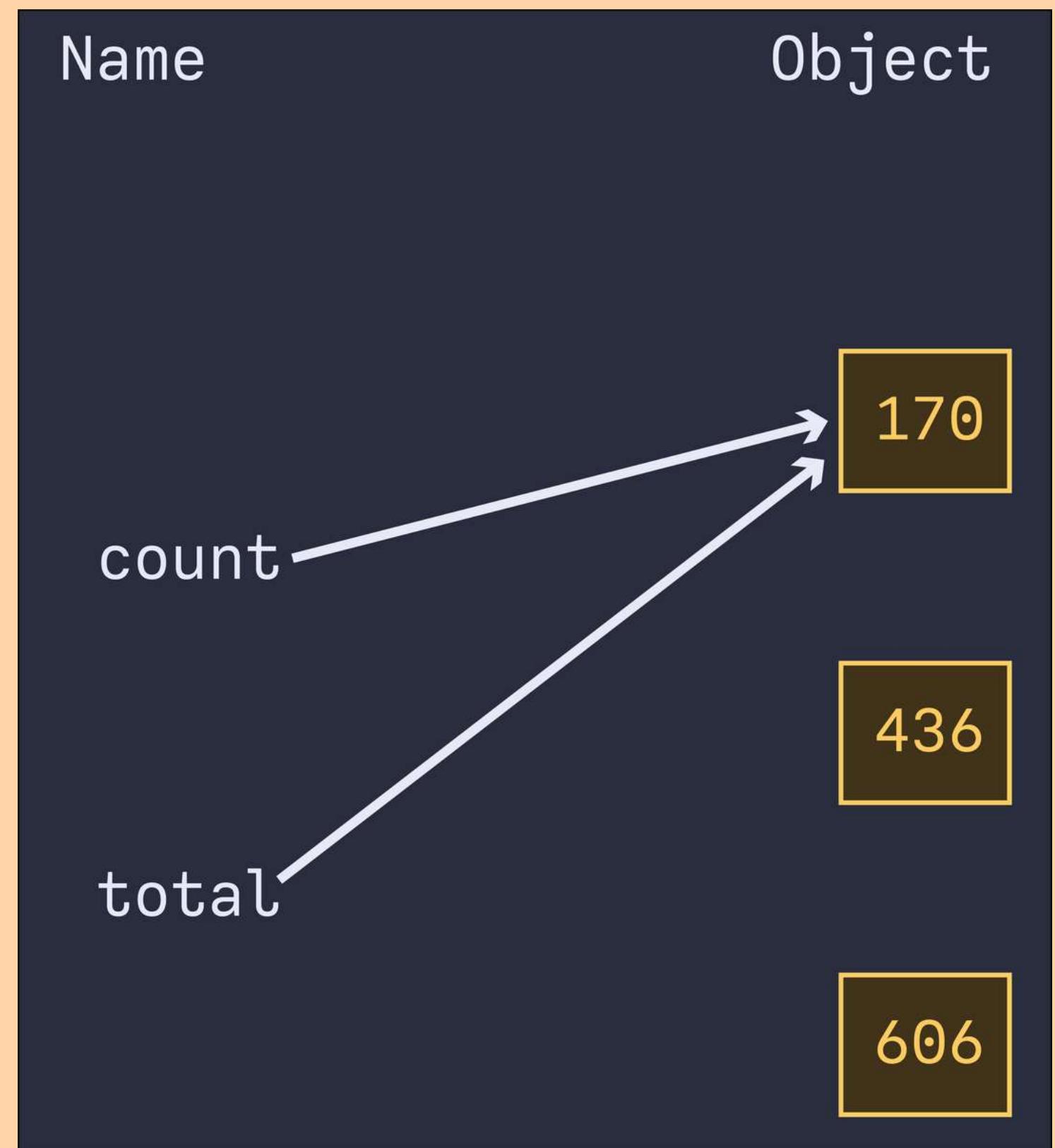
```
False      await     else      import    pass
None      break     except    in        raise
True       class    finally   is        return
and        continue for      lambda   try
as         def      from     nonlocal while
assert    del      global   not      with
async     elif     if       or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



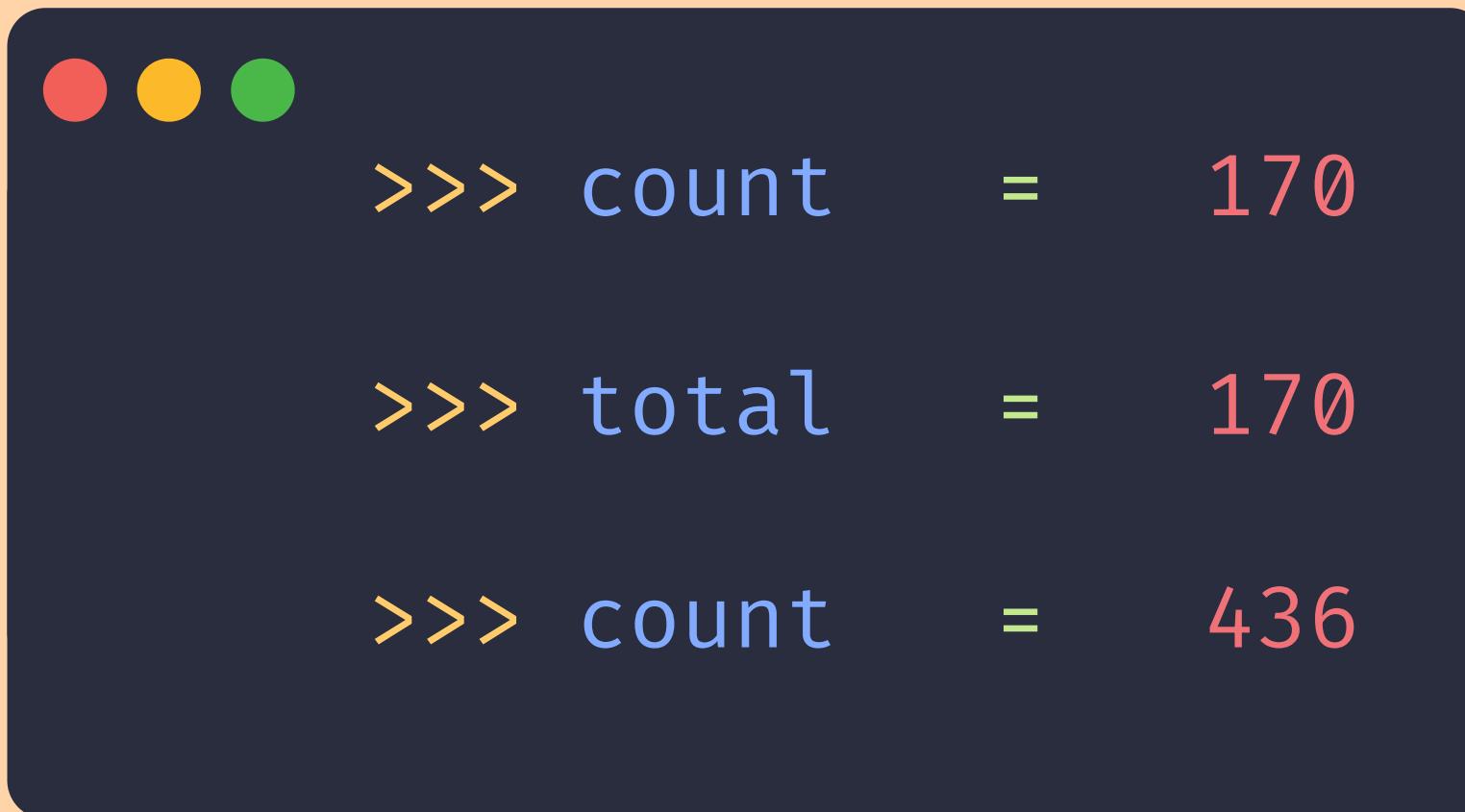
False raise in not global
True try is with del
None except as or assert
for import and class
while from continue def
if pass yield finally
else break await lambda
elif return async nonlocal

How Variables Work

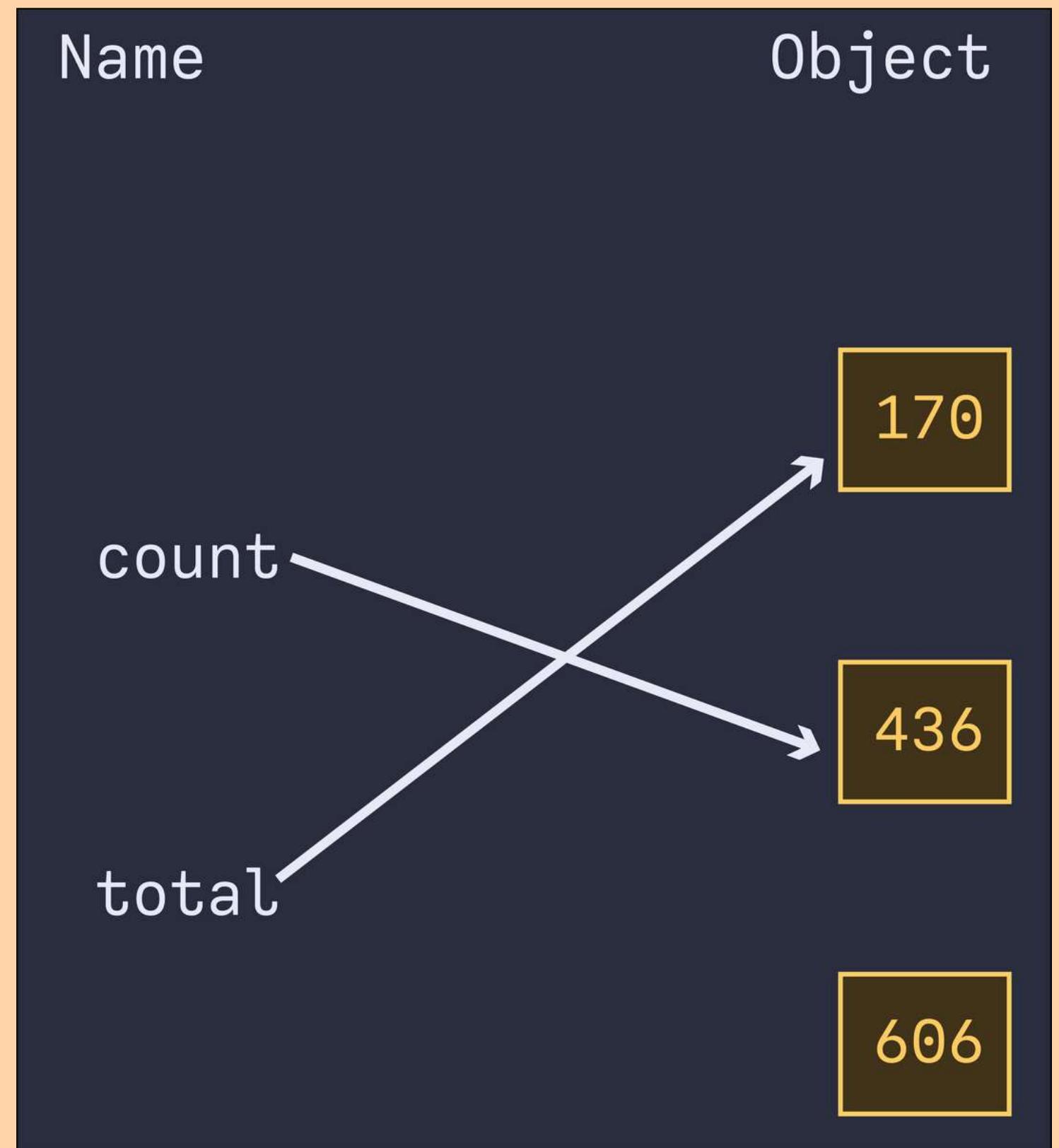
```
>>> count      =      170
>>> total      =      170
```



How Variables Work

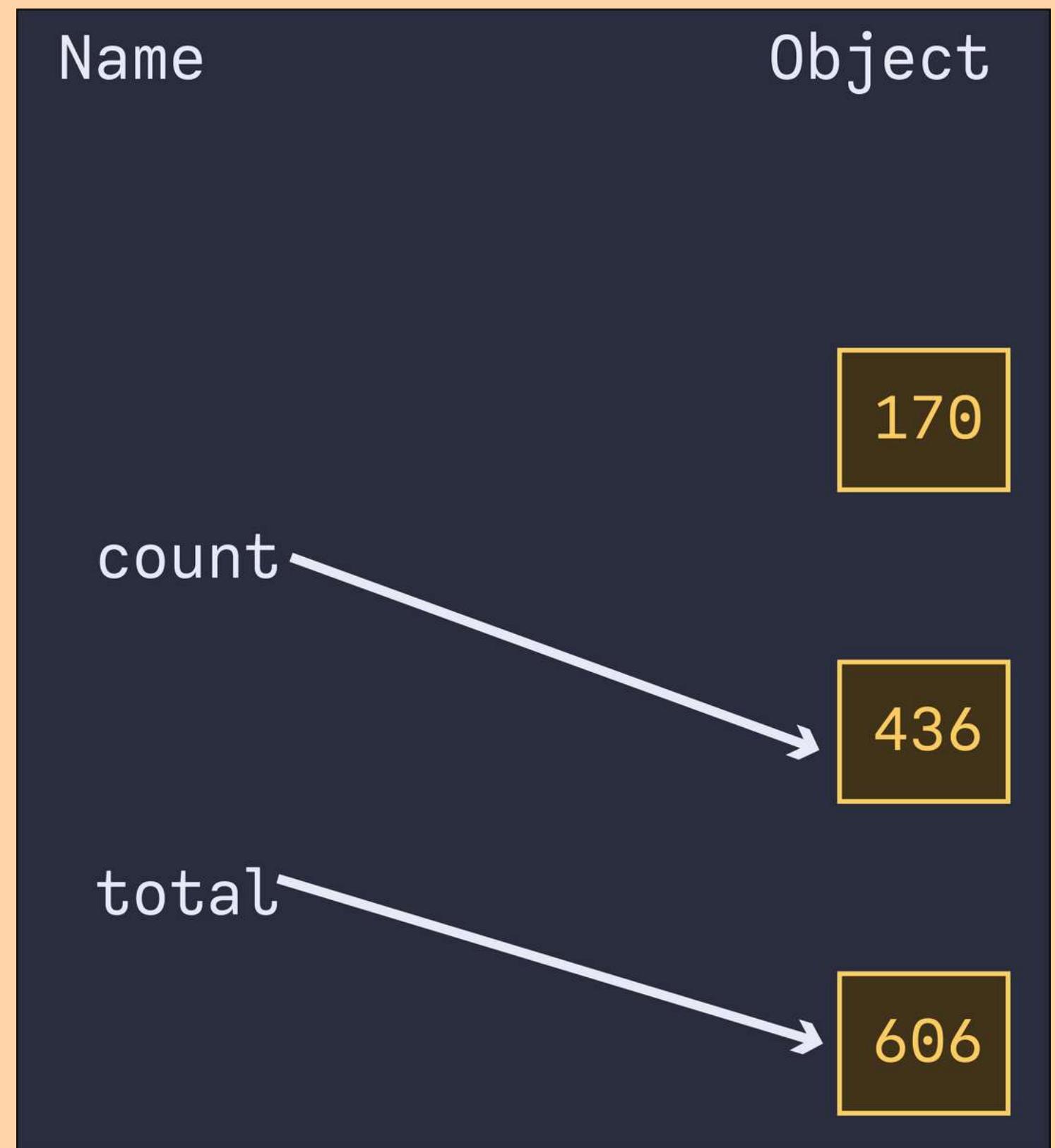


```
>>> count      =    170
>>> total      =    170
>>> count      =    436
```



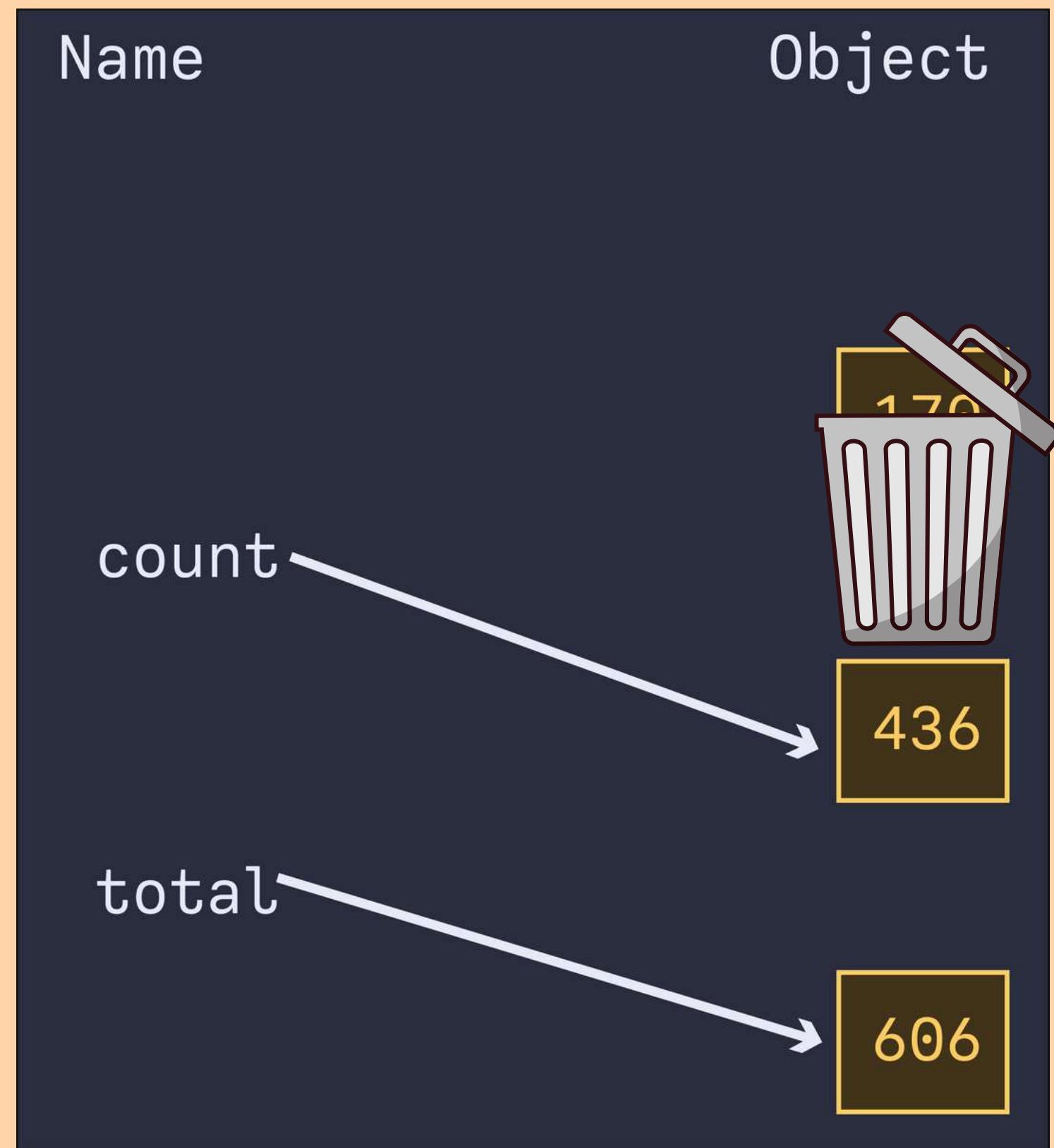
How Variables Work

```
>>> count      =    170
>>> total      =    170
>>> count      =    436
>>> total      =    606
```



How Variables Work

```
>>> count      =    170
>>> total      =    170
>>> count      =    436
>>> total      =    606
```



Assignment Operators

+ =

- =

*** =**

/ =

// =

**** =**

% =

A screenshot of a Python REPL window. At the top, there are three colored circles: red, yellow, and green. The window contains the following text:

```
>>> age = 48
>>> age += 2
>>> age
```

The output of the final command is '50'.

Update age to be
its current value
(48) plus 2

A screenshot of a Python REPL window. At the top, there are three colored circles: red, yellow, and green. The window contains the following text:

```
>>> age = 48
>>> age -= 10
>>> age
```

The output is the number 38, displayed in yellow at the bottom left.

Update age to be
its current value
(48) minus 10

Strings



Strings



Basic Data Types

Strings

Integers

Booleans

FLOATS

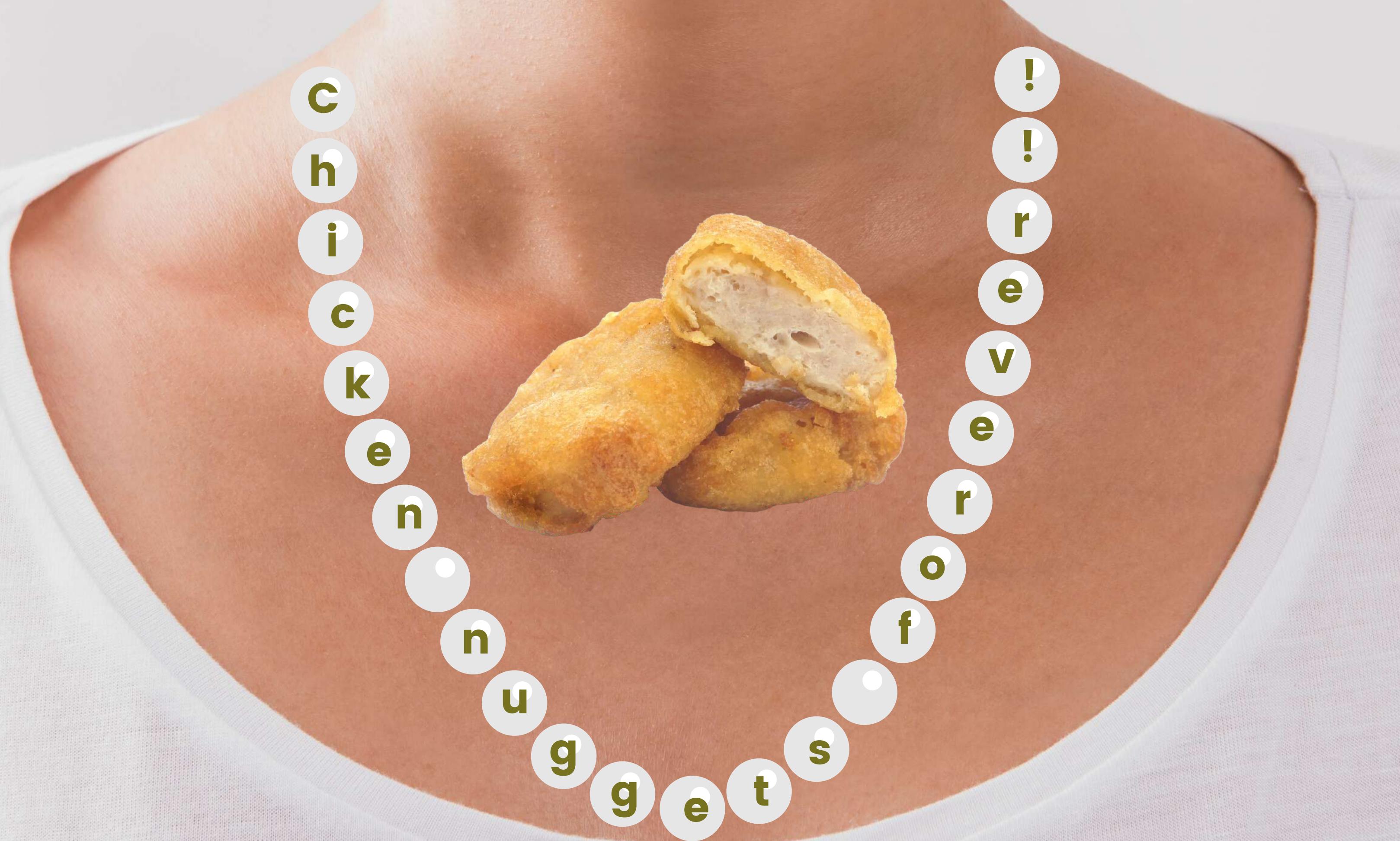


Strings

"STRINGS OF CHARACTERS"

Strings are a textual datatype and must be wrapped in quotes

chicken nuggets! never fear!



chicken nuggets for ever!



```
color = "Magenta"
```



```
twitter_handle = '@POTUS'
```



```
url = "www.reddit.com/r/formula1/"
```

Quotes

A screenshot of a Python terminal window. The command prompt shows three colored dots (red, yellow, green) followed by '>>>'. The string 'Hello World!' is assigned to a variable, indicated by the single quotes at the beginning and end of the text. A large green checkmark is positioned in the top right corner of the terminal window.

```
>>> 'Hello World!'
```

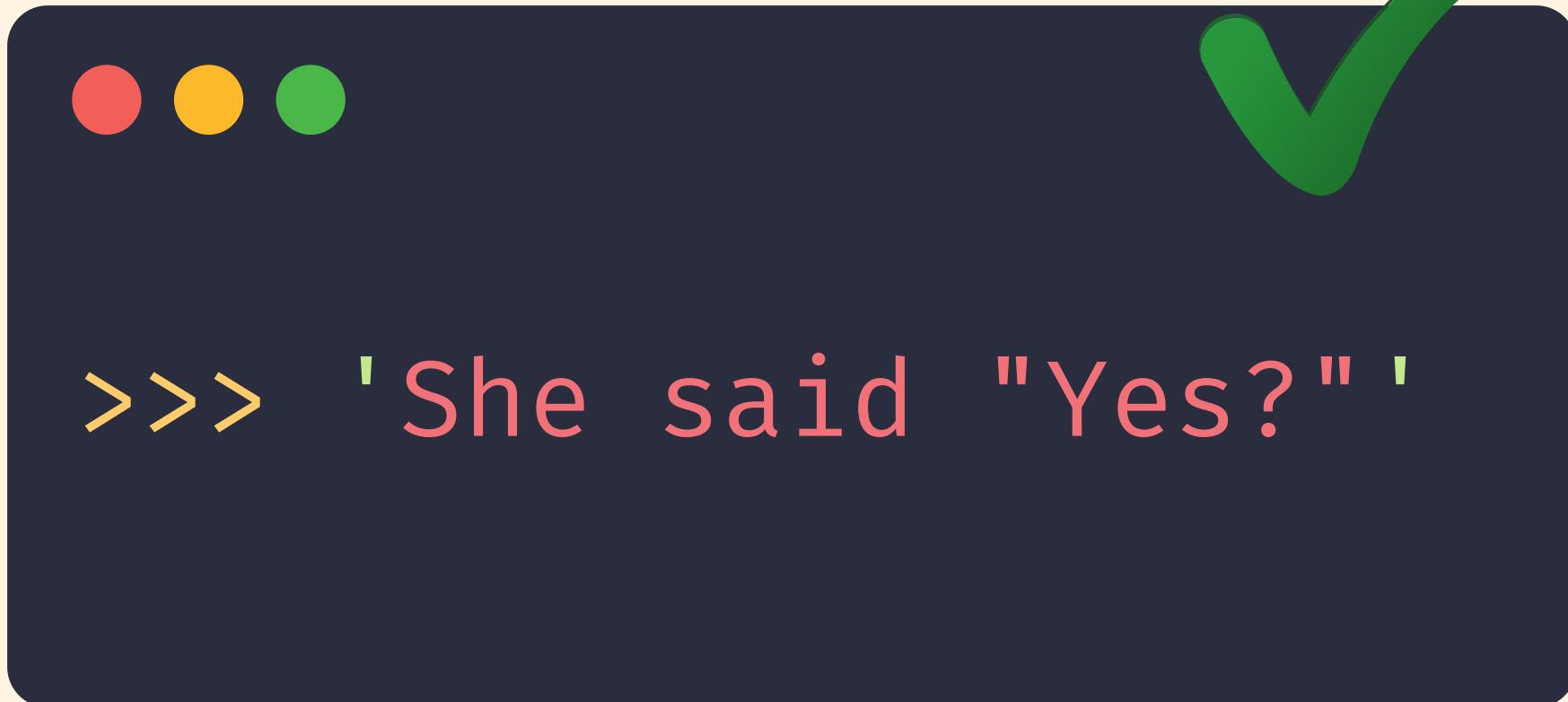
Start String End

A screenshot of a Python terminal window. The command prompt shows three colored dots (red, yellow, green) followed by '>>>'. The string 'The cat's toy' is assigned to a variable, indicated by the single quotes at the beginning and end of the text. A large red 'X' is positioned in the top right corner of the terminal window, indicating an error. Four arrows below the terminal identify the components of the string: a green arrow labeled 'Start' points to the first quote, a red arrow labeled 'String' points to the word 'cat', a green arrow labeled 'End' points to the second quote, and a purple arrow labeled 'Error' points to the apostrophe in 'cat's'.

```
>>> 'The cat's toy'
```

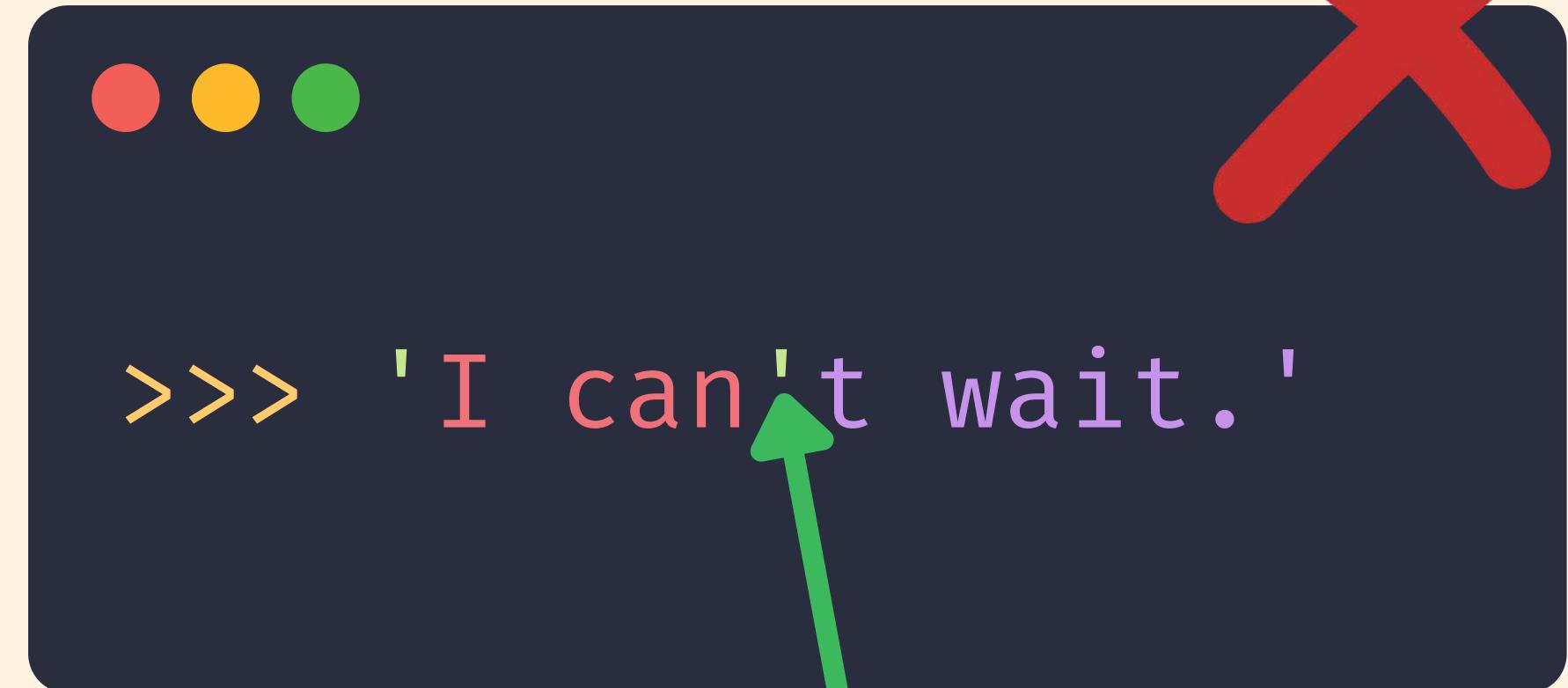
Start String End Error

Single Quotes



A dark blue terminal window icon with three colored dots (red, yellow, green) in the top-left corner. A large green checkmark is positioned in the top-right corner of the window area.

```
>>> 'She said "Yes?" '
```

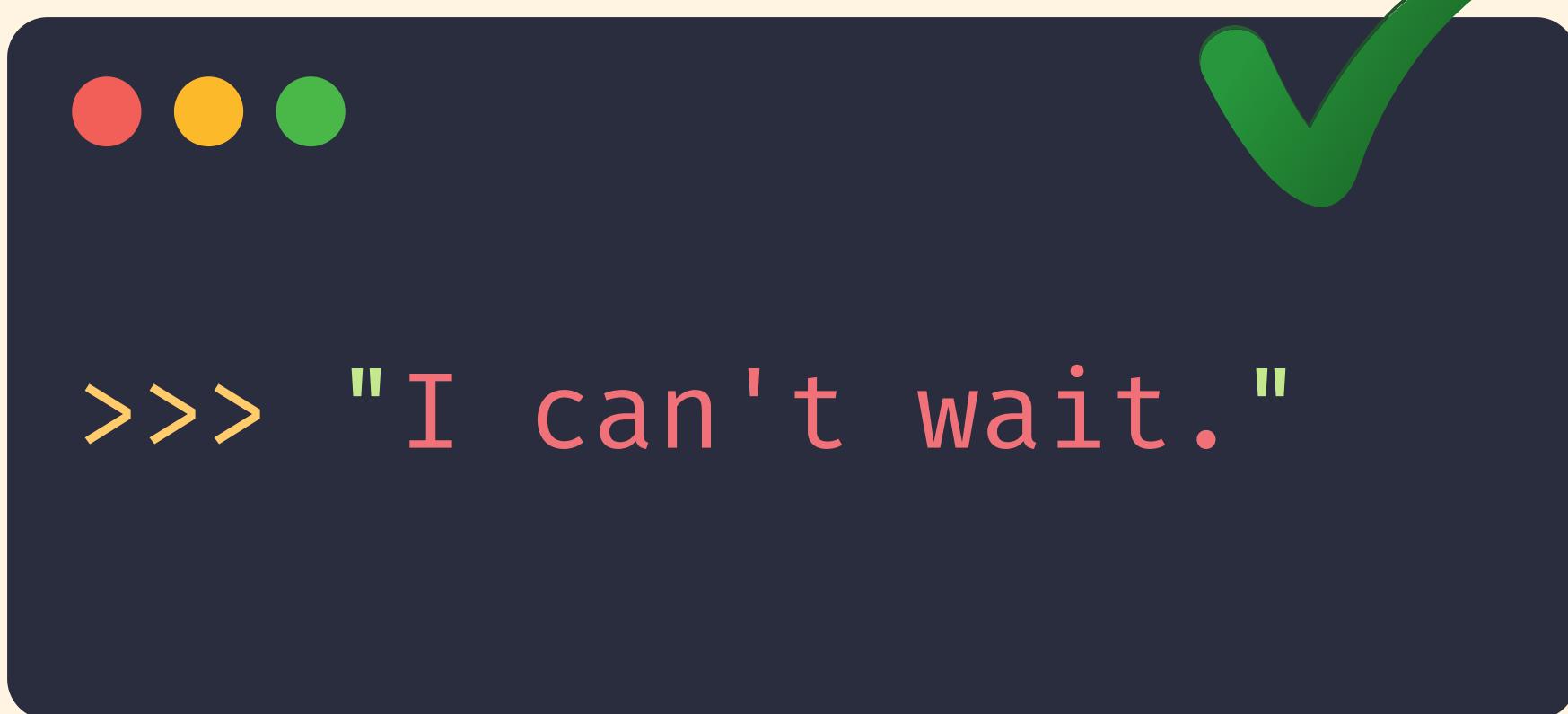


A dark blue terminal window icon with three colored dots (red, yellow, green) in the top-left corner. A large red X is positioned in the top-right corner of the window area.

```
>>> 'I can't wait.'
```

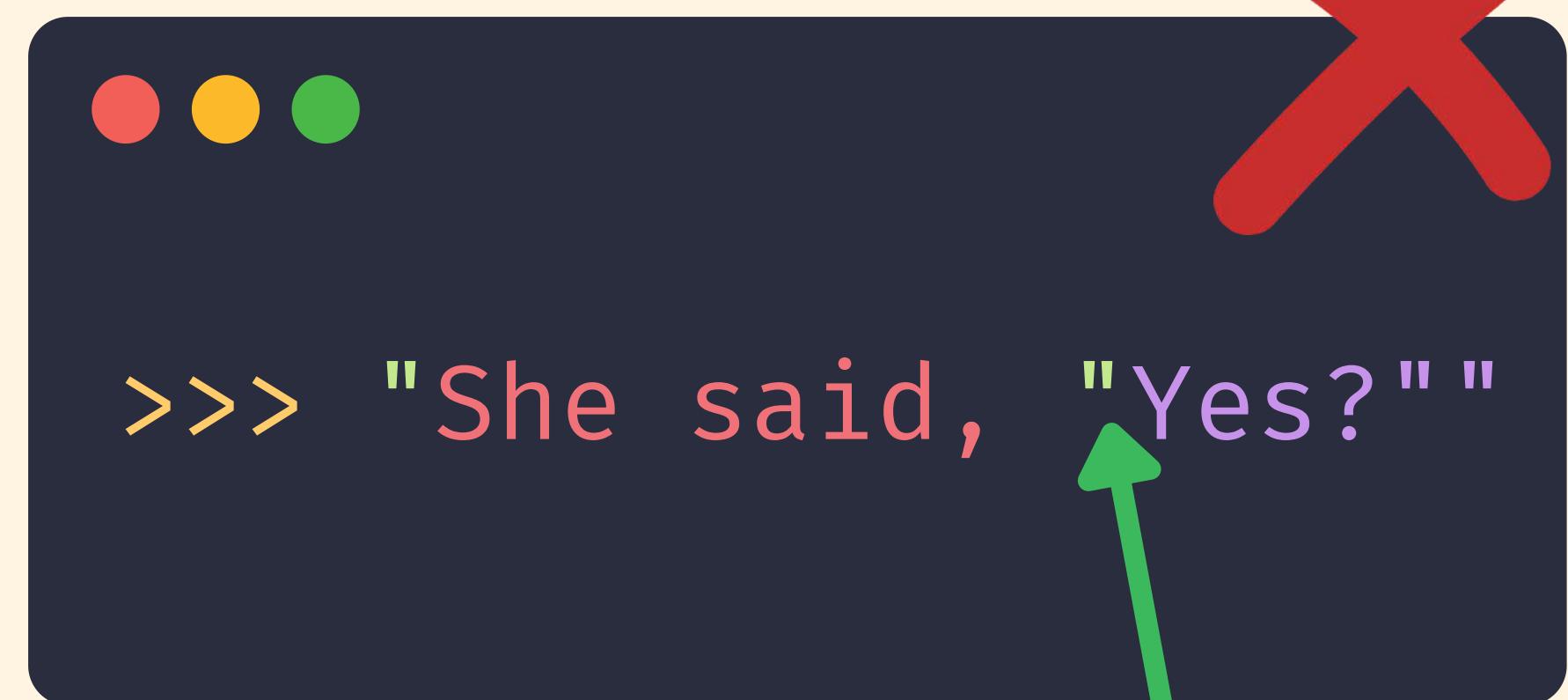
End

Double Quotes



A dark blue terminal window icon with three colored dots (red, yellow, green) in the top-left corner. A large green checkmark is positioned in the top-right corner of the window area.

```
>>> "I can't wait."
```



A dark blue terminal window icon with three colored dots (red, yellow, green) in the top-left corner. A large red X is positioned in the top-right corner of the window area.

```
>>> "She said, "Yes?""
```

End



Triple Quotes

Single Or Double



```
>>> """Colt's sourdough bread is  
the best," Paul Hollywood stated."""
```

Print

The `print()` function prints out any values we pass to it to "standard output". It does not return anything.



```
>>> print("hello")
```

Escape Characters



Newline - \n

Double Quote - \"

Tab - \t

Single Quote - \'

Backslash - \\

Concatenation

We can concatenate strings together by using the plus sign. No space will be added between them.

```
>>> 'pan' + 'cake'  
'pancake'
```

Multiplication

We can also multiply a string by a number,
which will repeat that string.

```
>>> 'ha'*4  
'hahahaha'
```

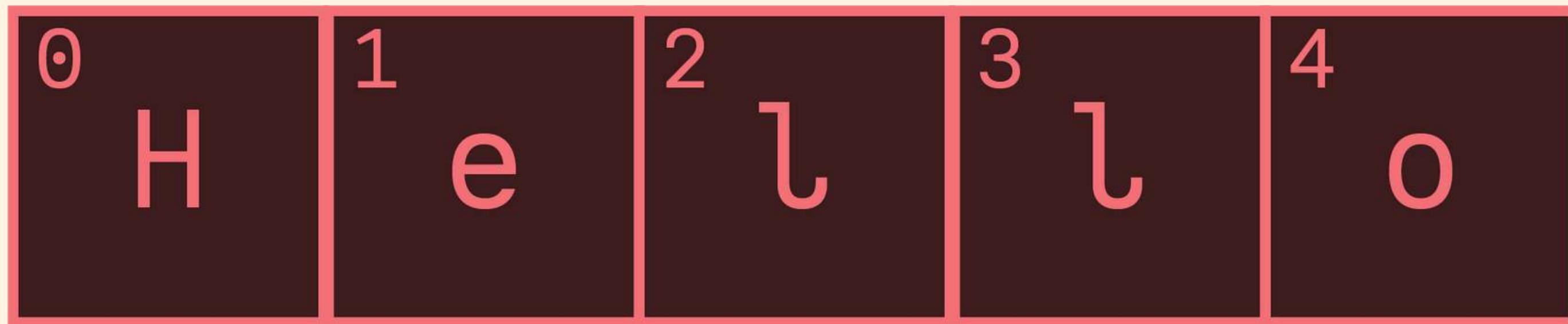
None

None is a special value in Python that denotes the **lack of value**. It is not the same as zero or an empty string (those are still values).

```
>>> user = None
```

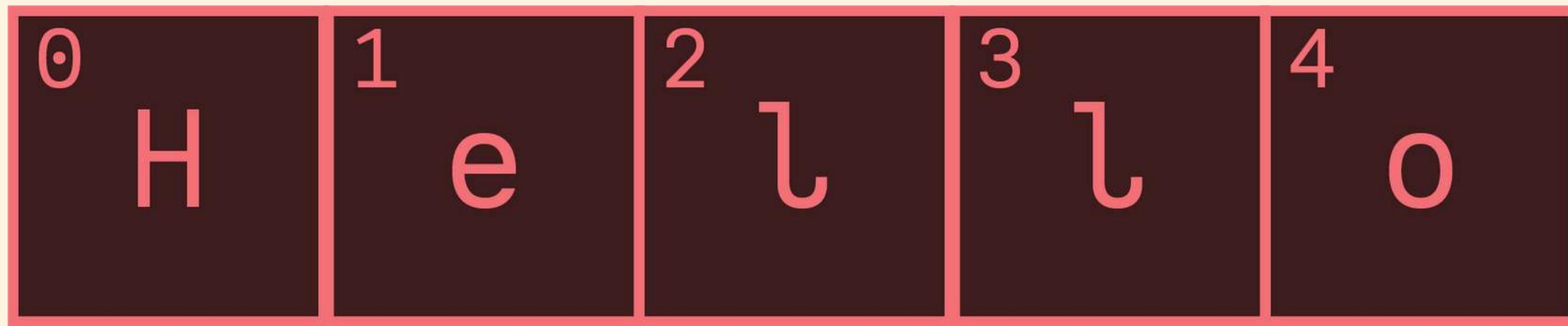
==

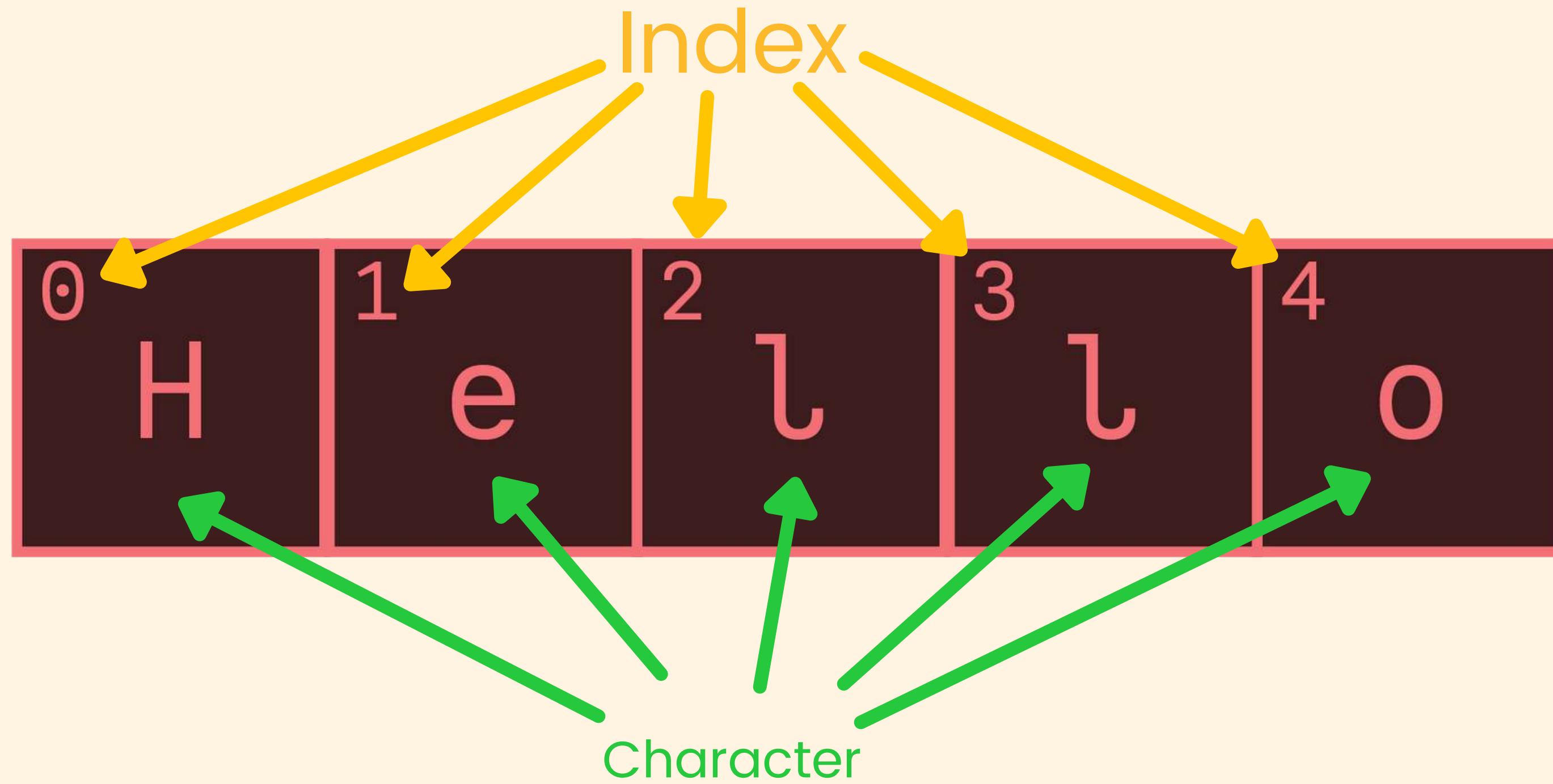
Strings Are Ordered



==

Strings Are Indexed

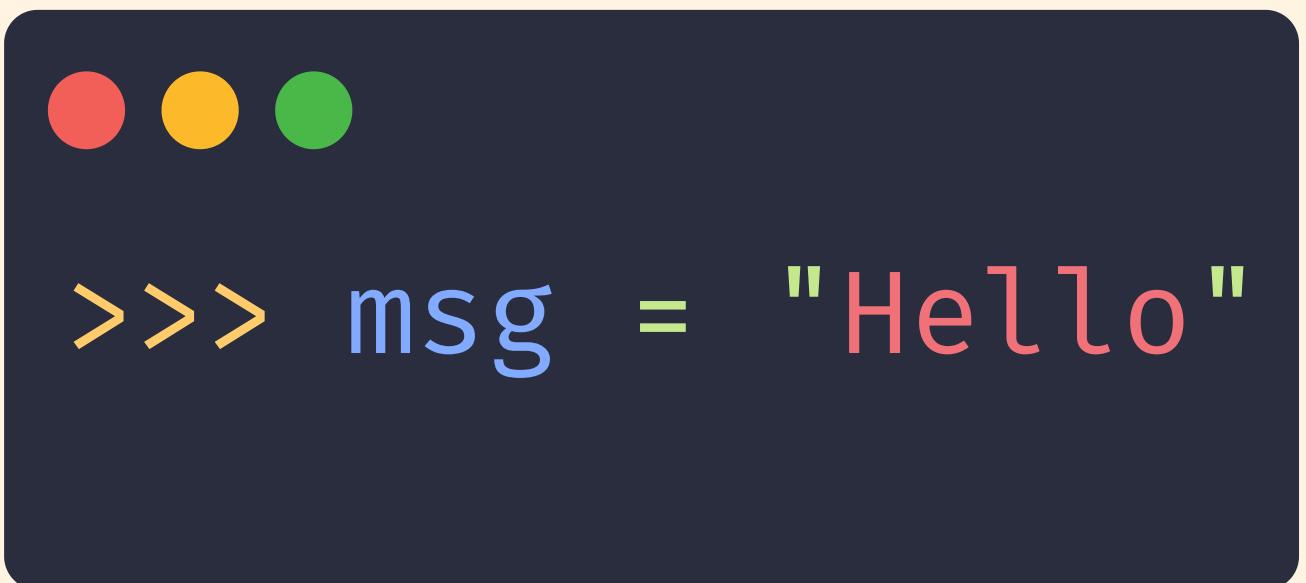






What is a name?

What is a name?
That which we
call a string
by any other name
would still say the same thing.



```
>>> msg = "Hello"
```



How Variables Work

Your Code

```
>>> msg = "Hello"
```

Names

msg  Hello

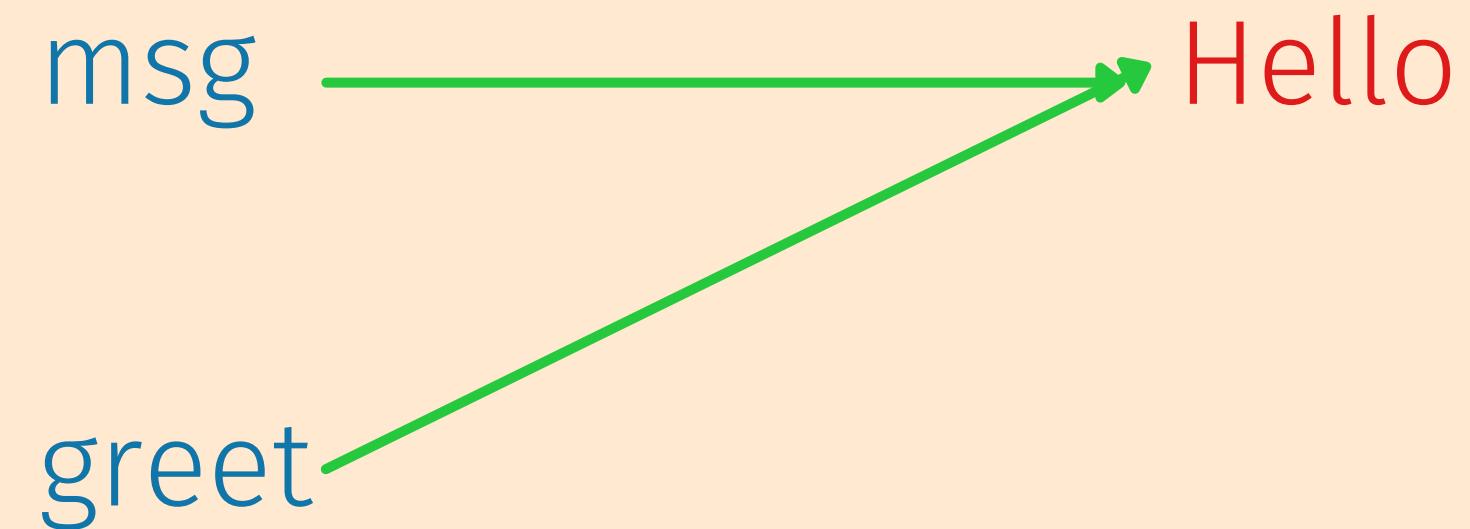
Objects

How Variables Work

Your Code

```
>>> msg = "Hello"  
>>> greet = "Hello"
```

Names



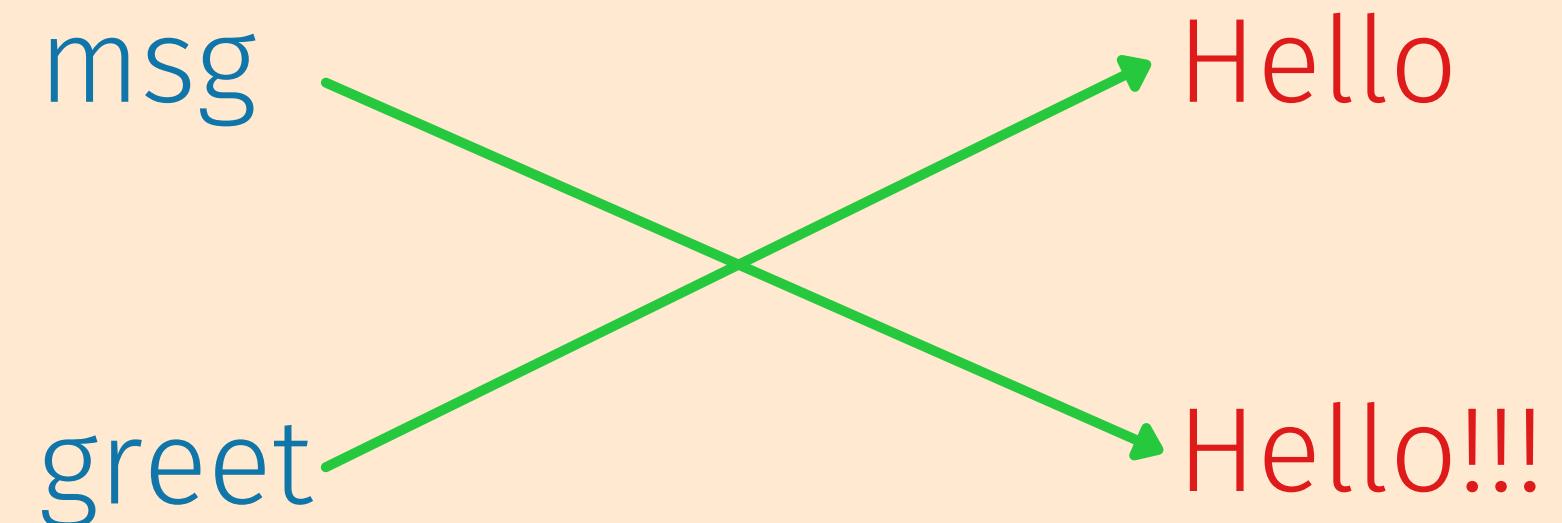
Objects

How Variables Work

Your Code

```
>>> msg = "Hello"  
>>> greet = "Hello"  
>>> msg = "Hello!!!"
```

Names



Objects

Indexes



```
>>> msg = "I <3 Cats"
>>> msg[0]
'I'
>>> msg[5]
'C'
```

A screenshot of a terminal window showing Python code. The code defines a string `msg` with the value "I <3 Cats". It then prints the first character at index 0 ('I') and the character at index 5 ('C'). A brown arrow points from the bottom right towards the index 5 in the last line of code.

0 1 2 3 4 5 6 7 8

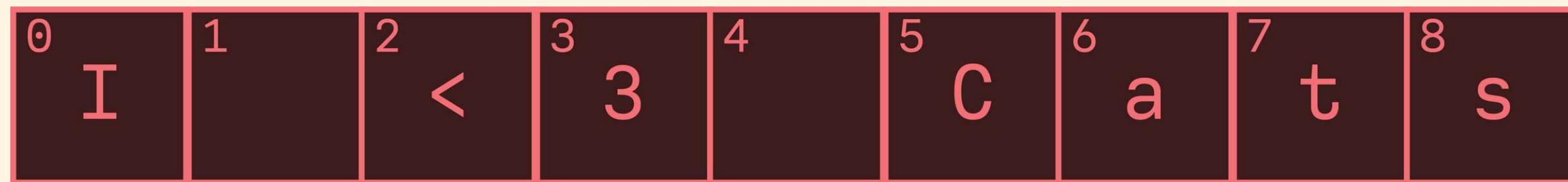
0	I	1	2	<	3	3	4	5	C	6	a	7	t	8	s
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

0 1 2 3 4 5 6 7 8

0 I	1	2 <	3 3	4	5 C	6 a	7 t	8 s
--------	---	--------	--------	---	--------	--------	--------	--------

-9 -8 -7 -6 -5 -4 -3 -2 -1

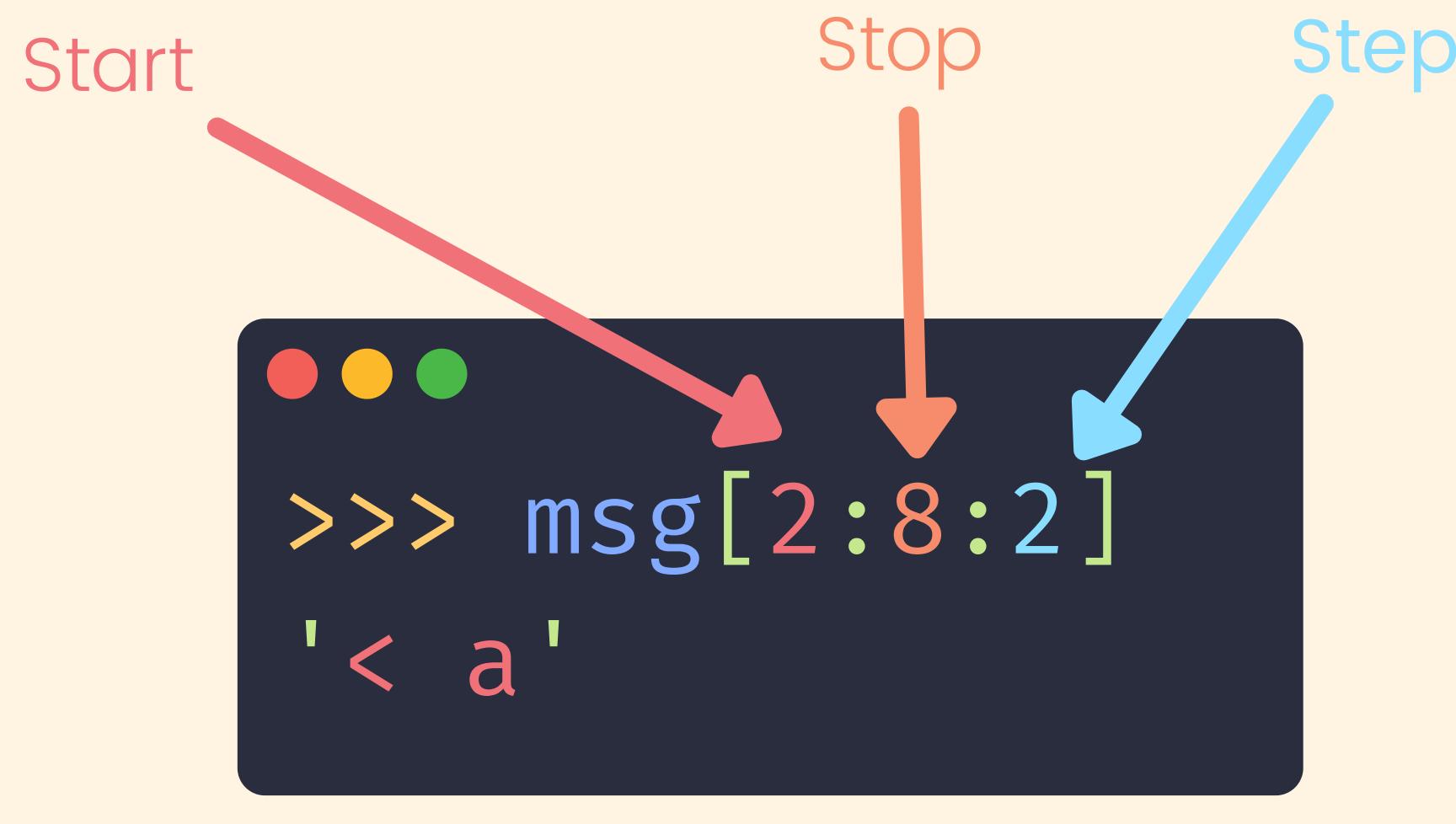
Slices



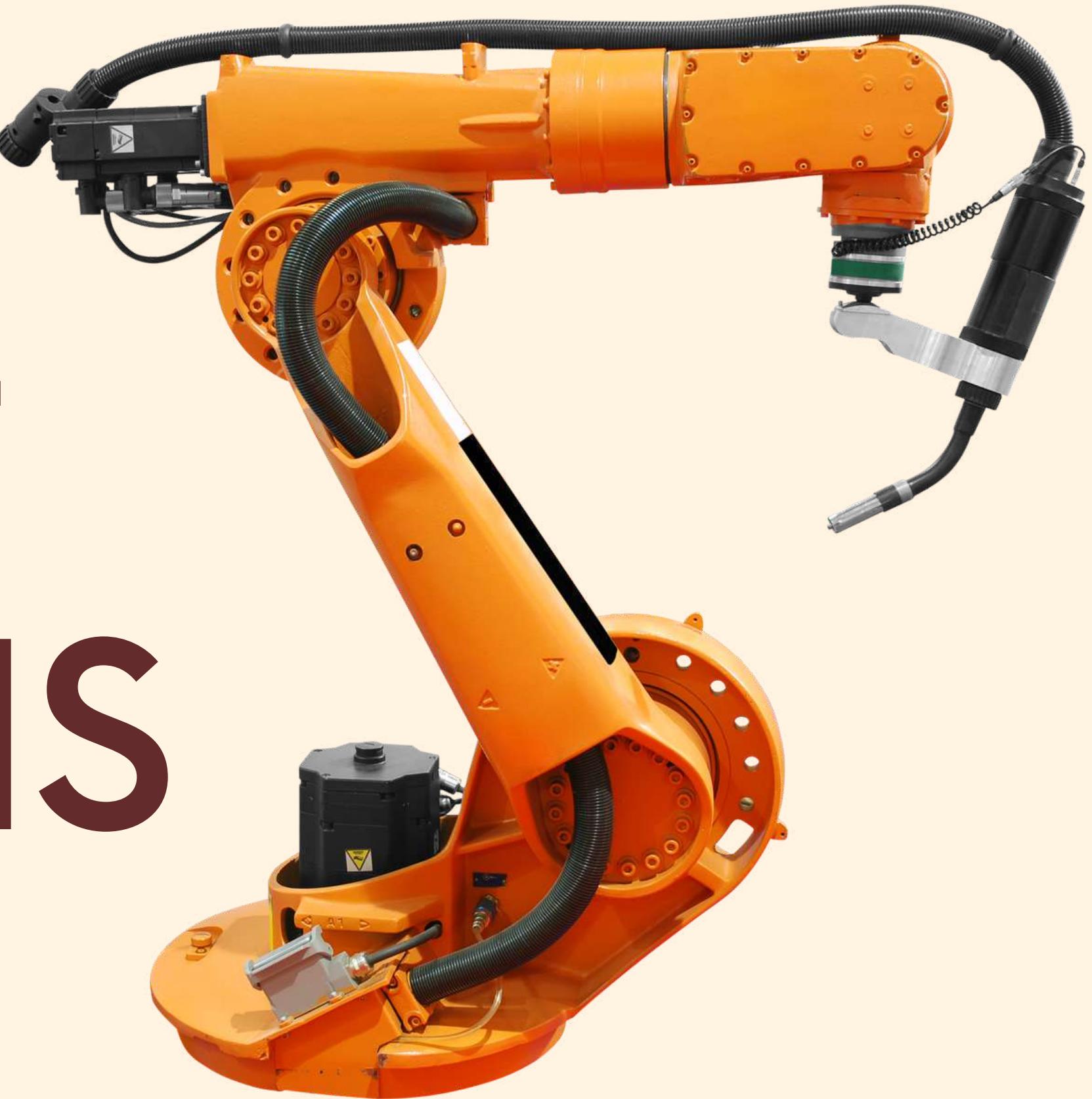
```
Python 3.8.5 (v3.8.5:481b4c2eef, Jul 22 2020, 15:53:44) [MSC v.1916 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> msg[2:6]
'<3 C'
```



Slices with a Step



STRINGS + FUNCTIONS

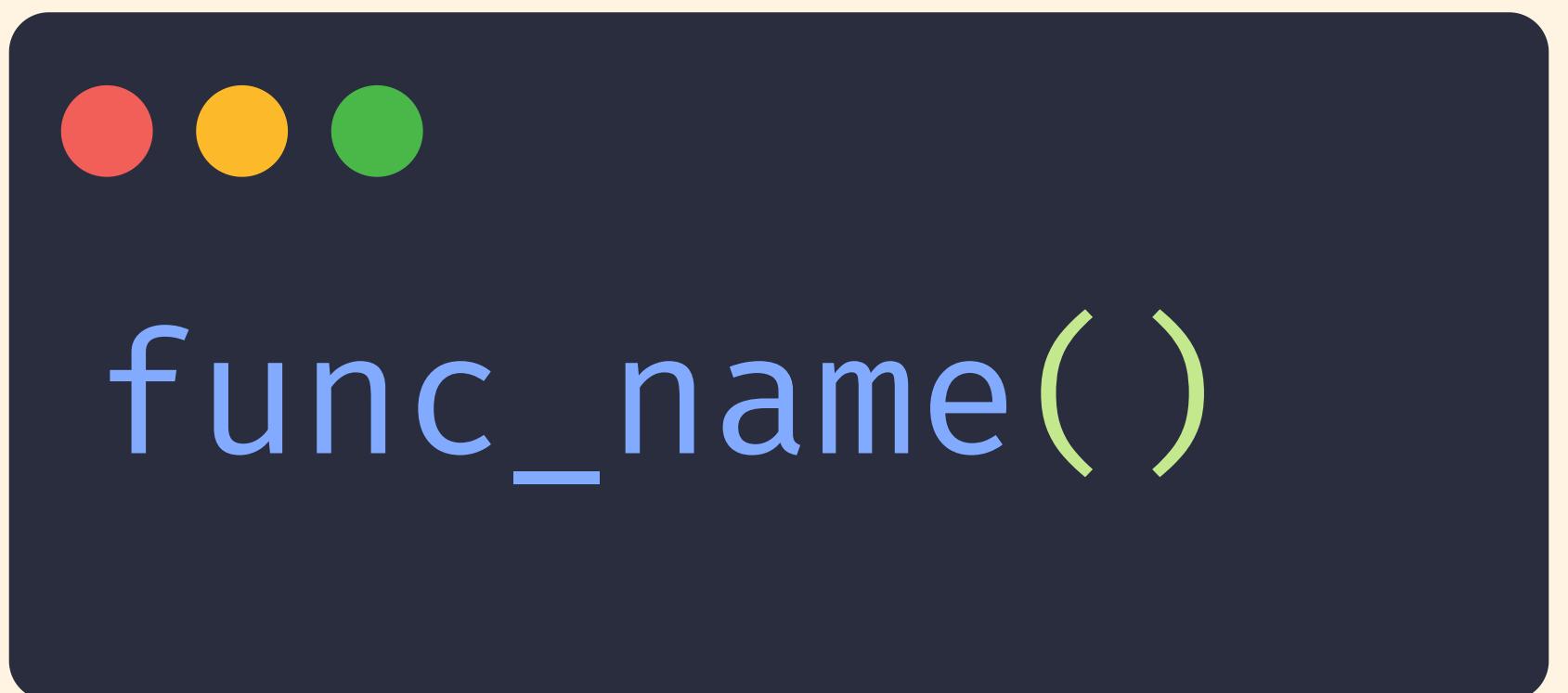


STRINGS + FUNCTIONS

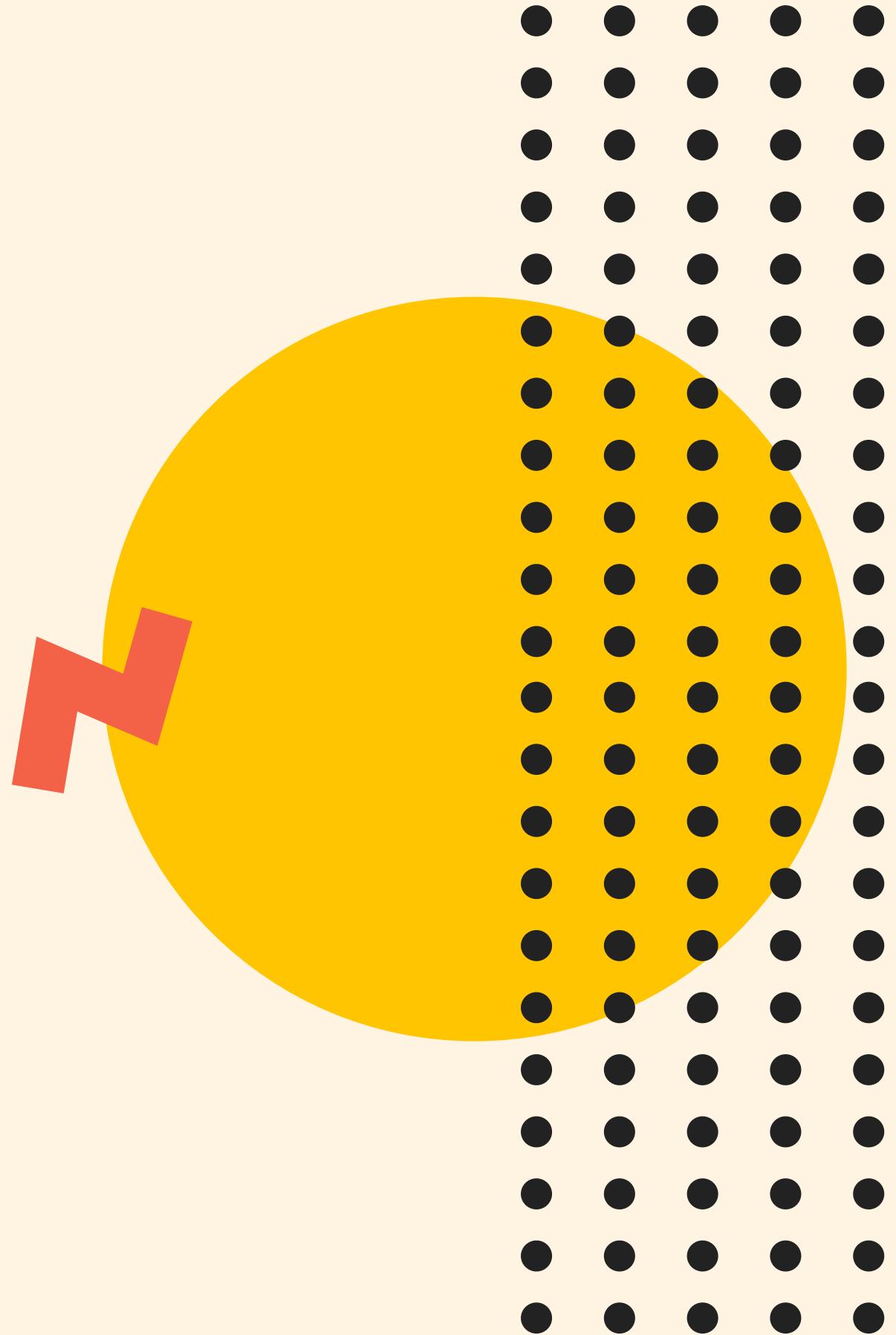


Functions

**FUNCTIONS ARE REUSABLE
ACTIONS THAT HAVE A NAME**



TO EXECUTE A FUNCTION, WE USE PARENS ()



Inputs

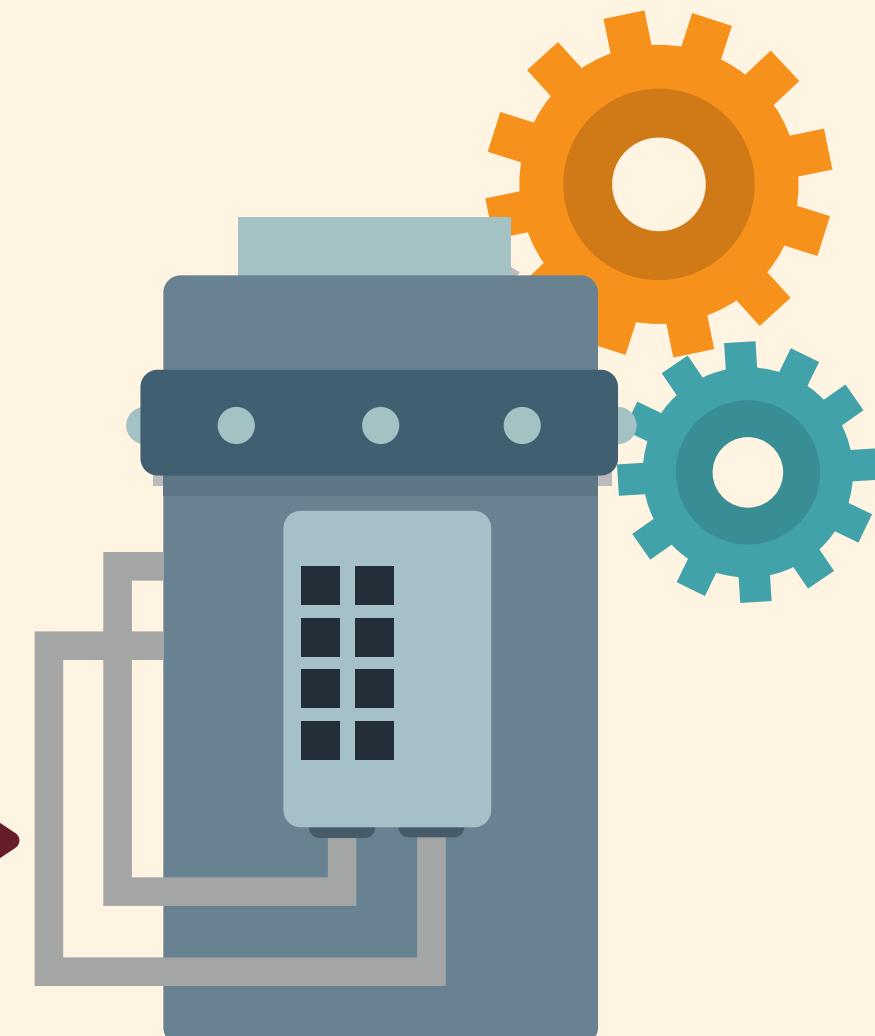
$\text{avg}(20, 25)$ →

$\text{avg}(3, 2, 5, 6)$ →

Output

→ 22.5

→ 4



Inputs

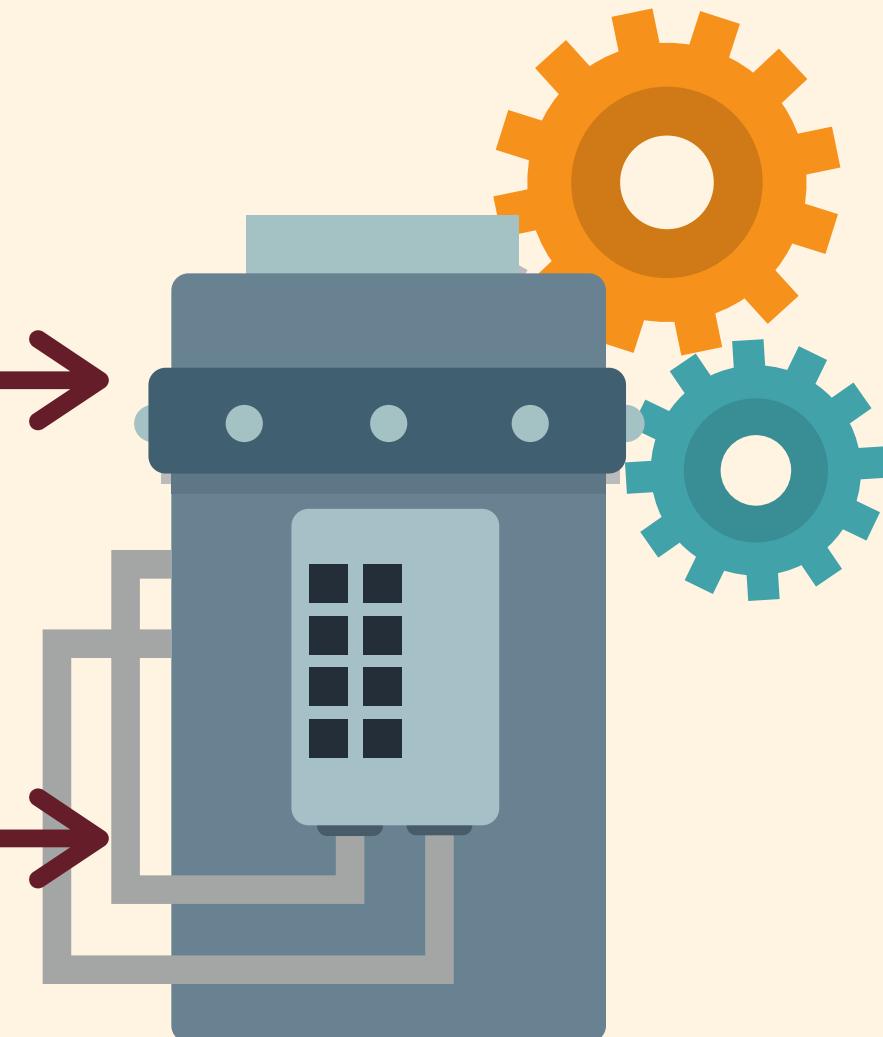
get_weather(10003)

Output

"23 f"

get_weather(92328)

"78 f"



Inputs

login('todd', 'jjkh2fj!d')



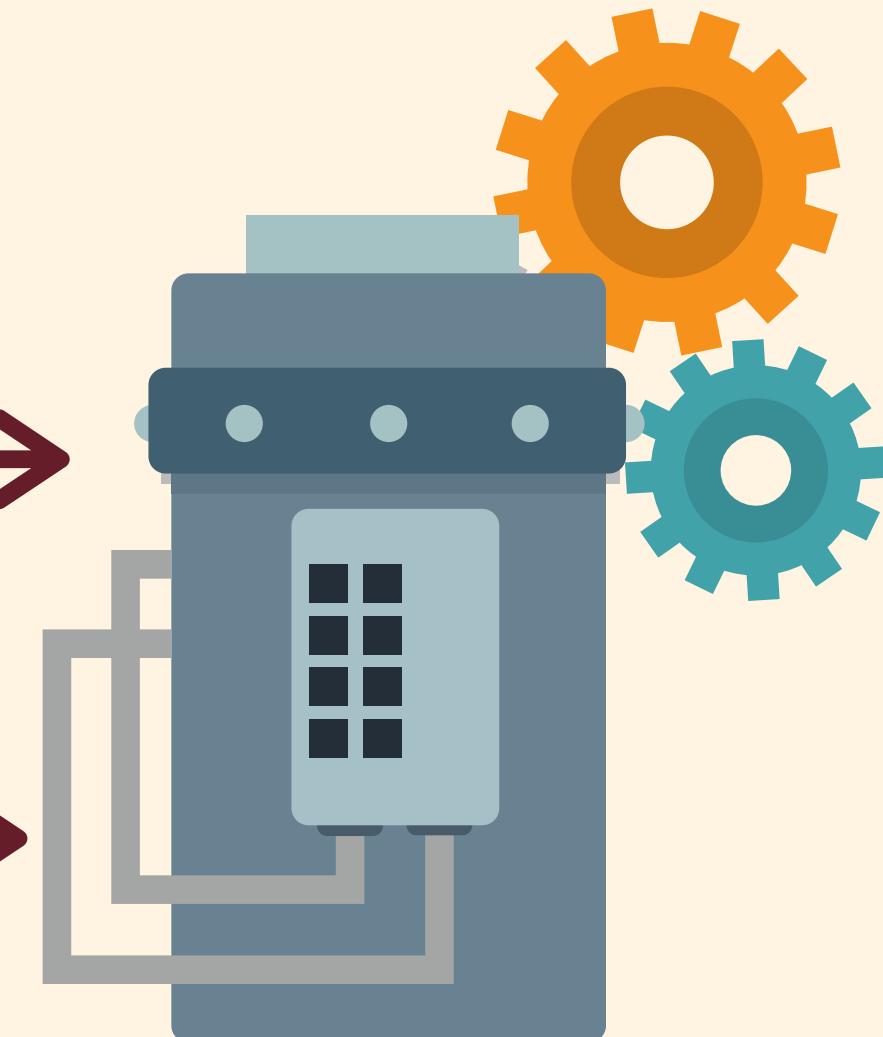
login('todd', 'kittycat')



Output

False

True





Arguments

(Fancy word for inputs)

```
>>> burger(bun,  
secret_sauce, lettuce,  
tomato, bacon, cheese,  
well_done)
```



≡

Arguments

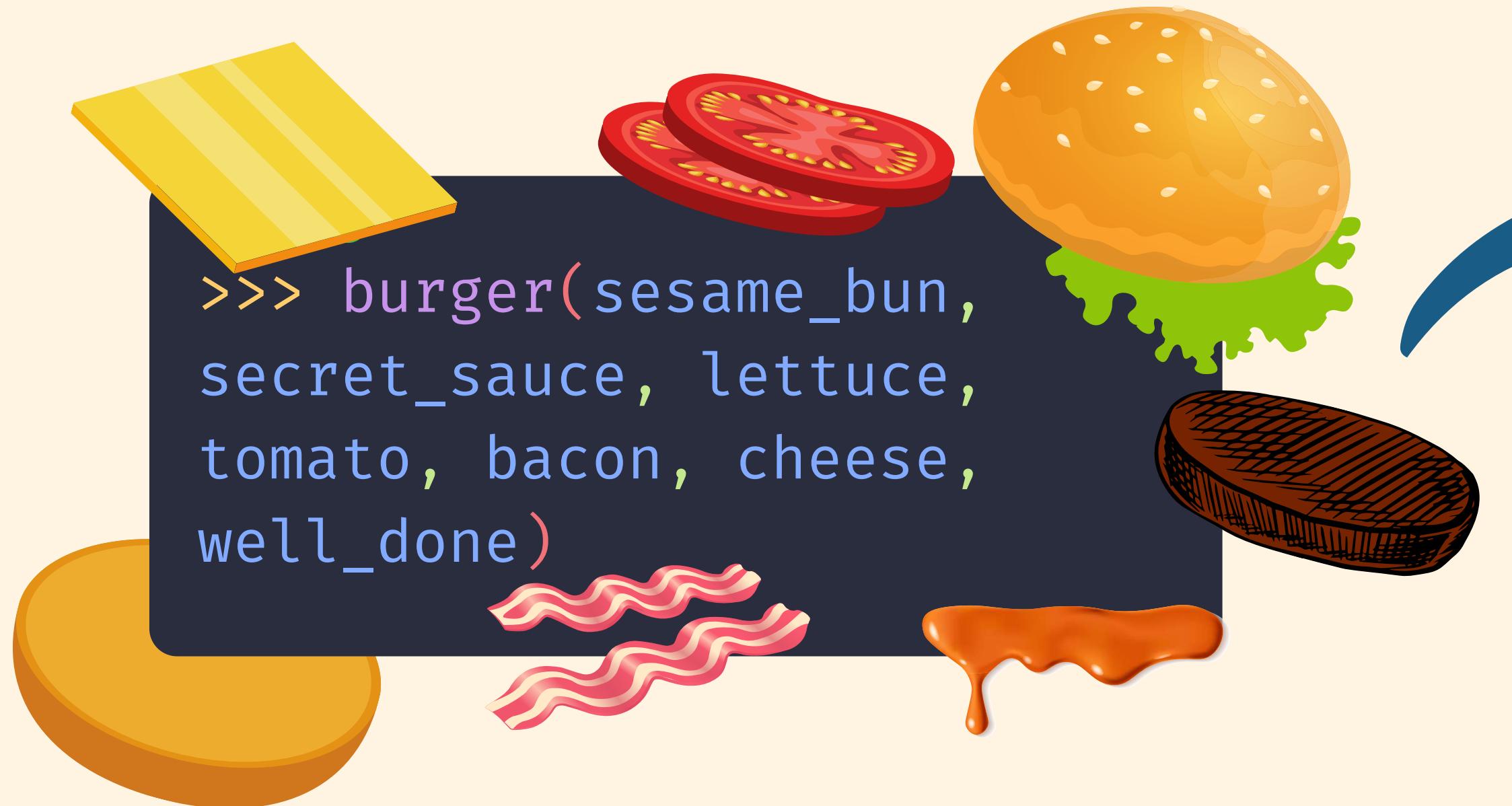
(Fancy word for inputs)

```
>>> burger(bun, tomato,  
bacon, cheese)
```



==

```
>>> burger(sesame_bun,  
secret_sauce, lettuce,  
tomato, bacon, cheese,  
well_done)
```



==

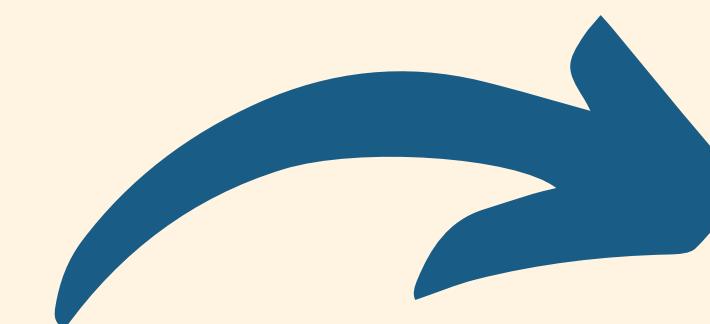
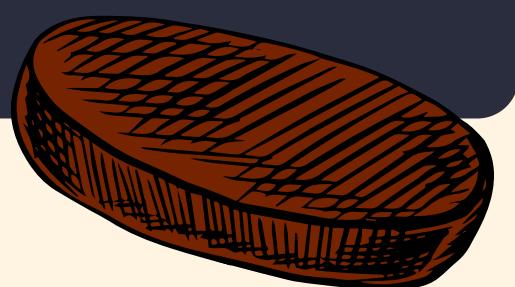
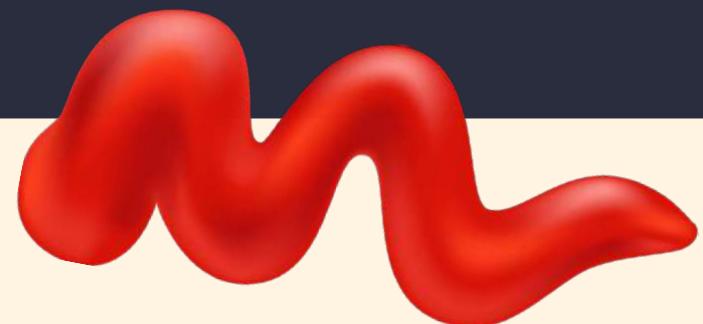
```
>>> burger(sesame_seed_bun,  
tomato, bacon, cheese,  
well_done)
```



==



```
>>> burger(lettuce_wrap,  
ketchup, tomato,  
swiss_cheese, well_done)
```



classmethod()	isinstance()	property()
delattr()	issubclass()	range()
dict()	iter()	reversed()
dir()	len()	round()
filter()	locals()	setattr()
getattr()	map()	slice()
globals()	max()	sorted()
help()	object()	sum()
id()	open()	type()
input()	print()	zip()

classmethod()
delattr()
dict()
dir()
filter()
getattr()
globals()
help()
id()
input()
isinstance()
issubclass()
iter()
len()
locals()
map()
max()
object()
open()
print()
property()
range()
reversed()
round()
setattr()
slice()
sorted()
sum()
type()
zip()

Length

The `len()` function will return the length of whatever item we pass to it. So far Strings are the only sequence we've seen, but soon we will see others!

```
● ● ●  
>>> word = "Chicken"  
>>> len(word)  
7
```

Input

The `input()` function will prompts a user to enter some input, converts it into a string, and then returns it. We can use it to gather user input in our programs

```
...> >>> age = input("how  
old are you?")
```

Type

The `type()` function accepts an input object and will return the type of that object

```
>>> type("hi")
<class 'str'>

>>> type(55)
<class 'int'>
```



Casting Types!

```
...>>> int("12")
12

>>> float("3.3")
3.3

>>> str(44.5)
'44.5'
```



Print

The `print()` function prints any arguments we pass to it to "standard output". It does not return anything.

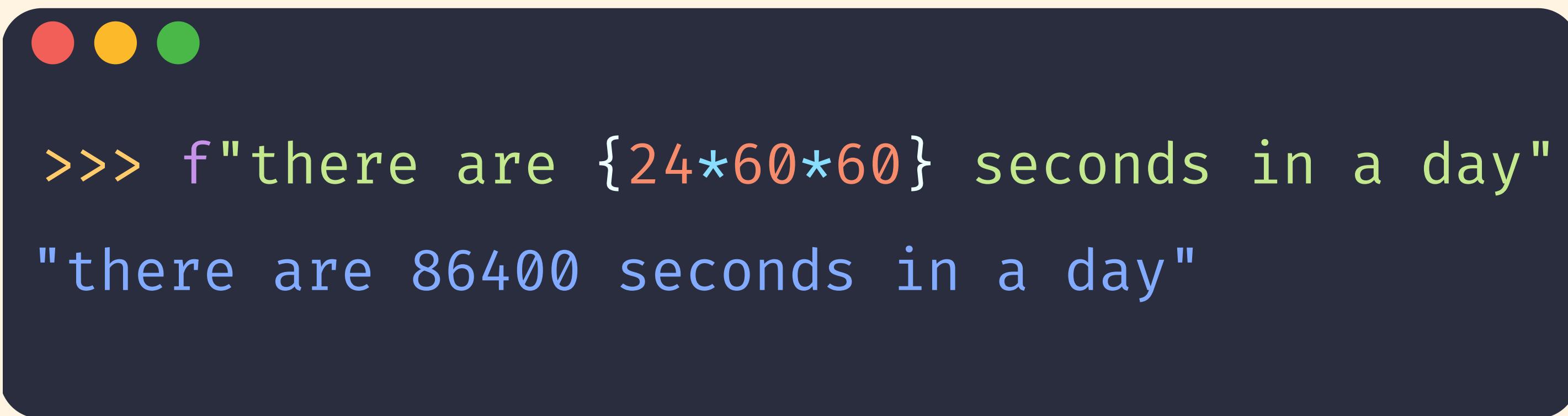


```
>>> print("hello")
```



f strings

f-strings are an easy way to generate strings that contain interpolated expressions. Any code between curly braces {} will be evaluated and then the result will be turned into a string and inserted into the overall string.

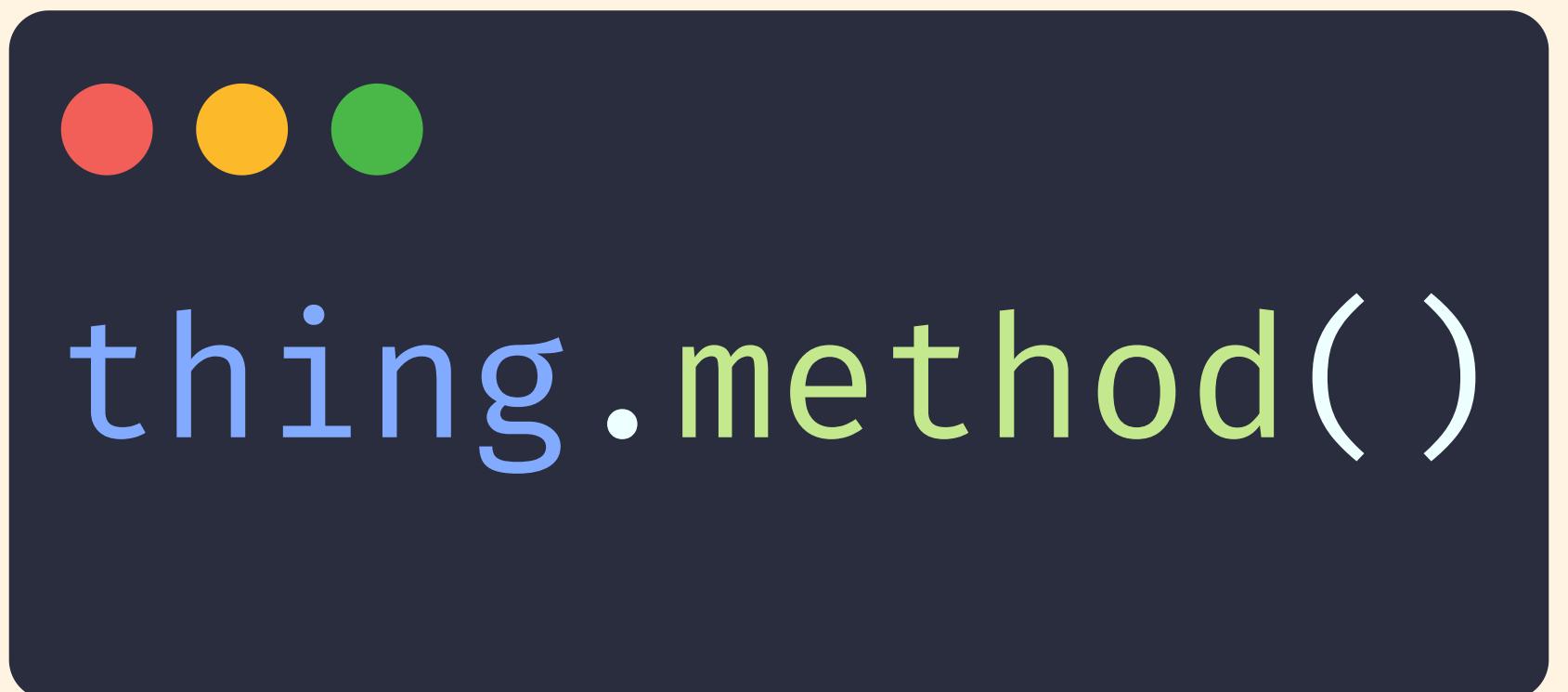


```
>>> f"there are {24*60*60} seconds in a day"  
"there are 86400 seconds in a day"
```

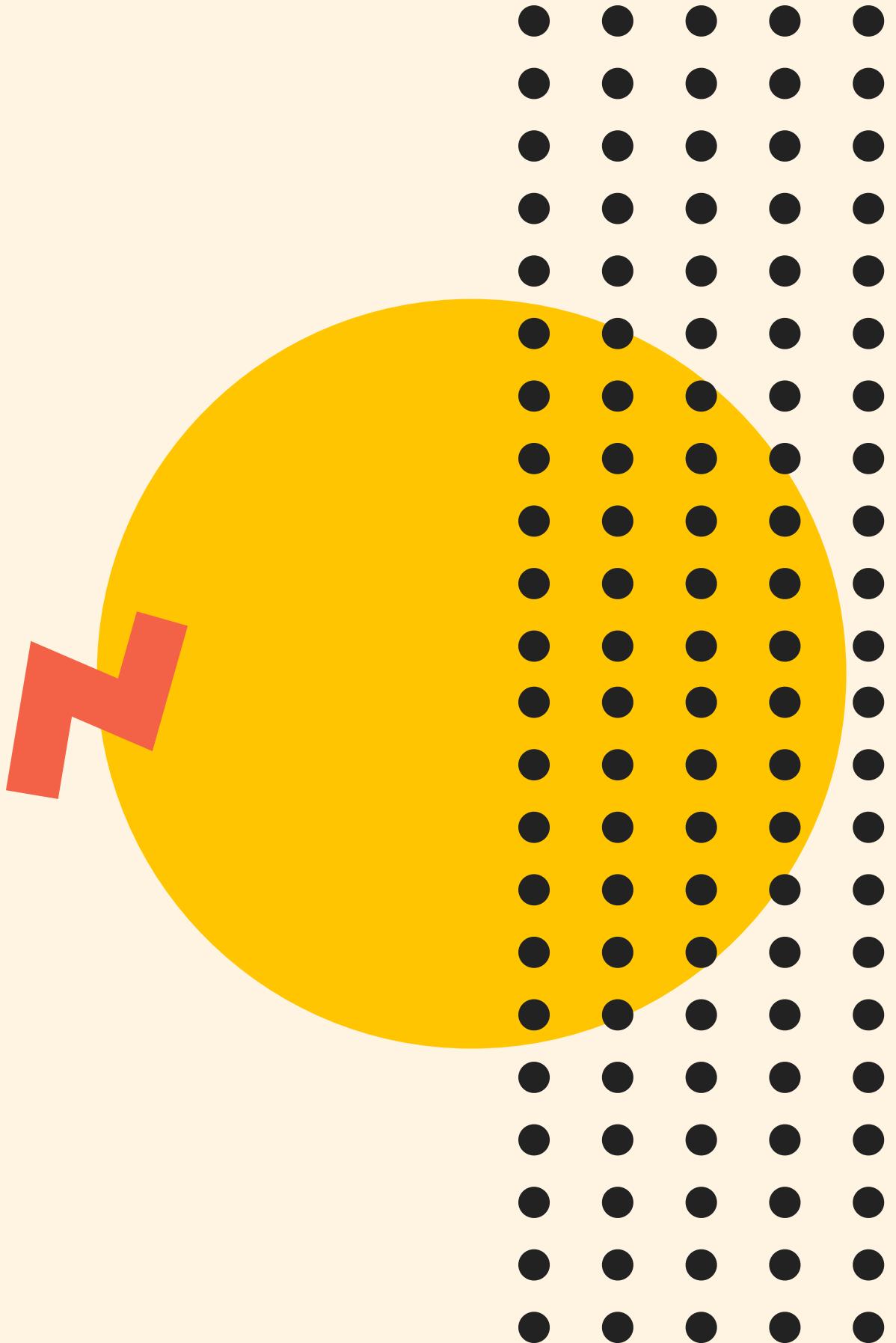


Methods

METHODS ARE FUNCTIONS
THAT "LIVE" ON OBJECTS



METHODS AUTOMATICALLY HAVE ACCESS TO THE
OBJECT THEY ARE CALLED ON.



thing.method()

String Methods

`str.capitalize()`

`str.endswith()`

`str.find()`

`str.join()`

`str.lower()`

`str.lstrip()`

`str.removeprefix()`

`str.removesuffix()`

`str.replace()`

`str.rfind()`

`str.rindex()`

`str.rjust()`

`str.split()`

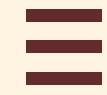
`str.startswith()`

`str.strip()`

`str.swapcase()`

`str.title()`

`str.upper()`



Capitalization Methods

```
>>> msg = "Hello world"  
>>> msg.capitalize()  
Hello world  
>>> msg.upper()  
HELLO WORLD  
>>> msg.lower()  
hello world
```



`str.upper()`



**accepts no
arguments!**

```
str.strip([chars])
```



chars is optional

strip()

Strips space characters

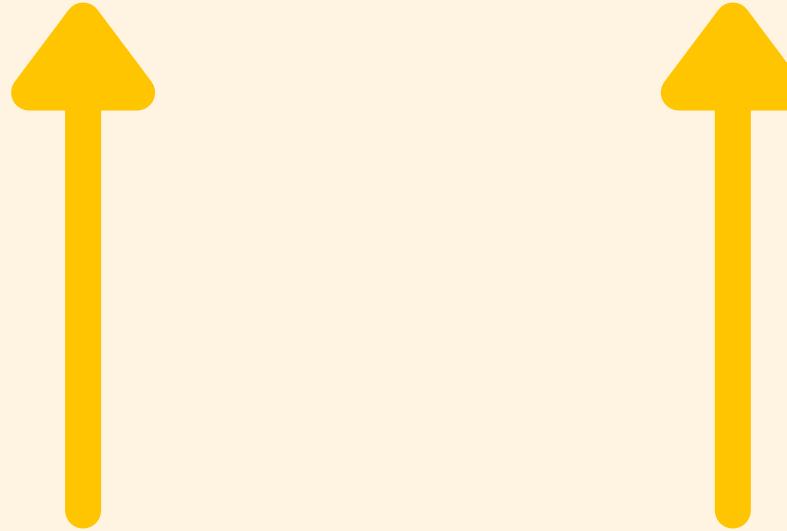
strip(' - ')

Strips '-' characters

strip('=-')

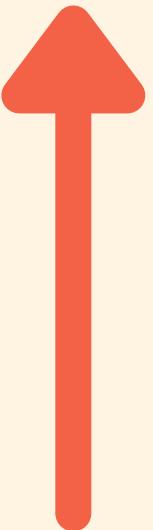
Strips '=' and '-' characters

`str.replace(old, new, [count])`



old and new are required

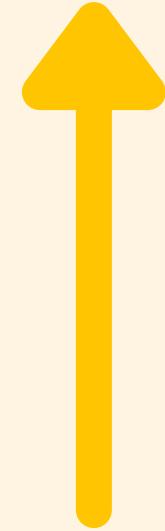
```
str.replace(old, new, [count])
```



count is optional

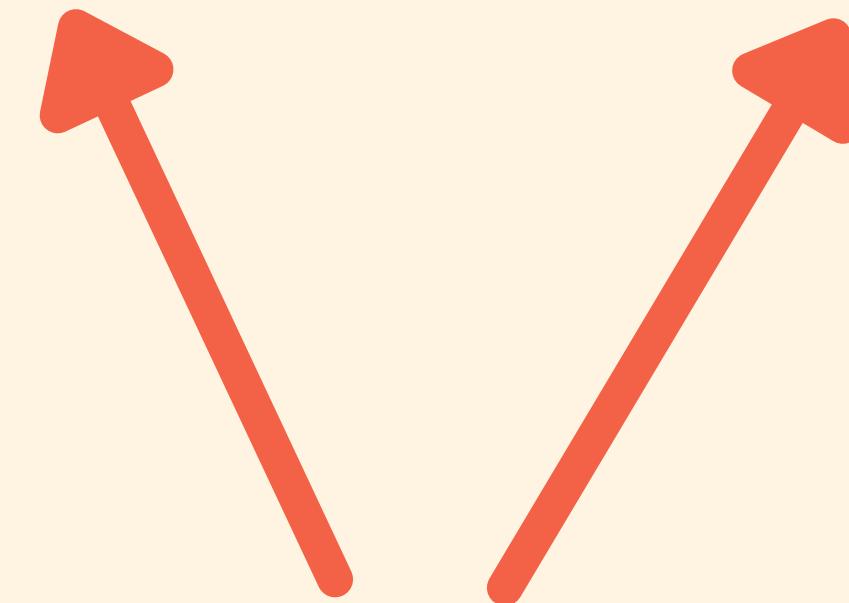
`find(sub[, start[, end]]) -> int`

```
find(sub[, start[, end]]) -> int
```



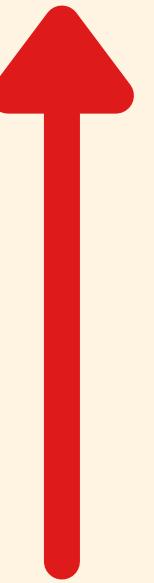
sub is a required argument

`find(sub[, start[, end]]) -> int`



anything in [] is optional

`find(sub[, start[, end]]) -> int`



find returns an integer

`find("c")` returns index where "c" is first found

`find("c", 5)` returns index where "c" is first found, after index 5

`find("c", 5)` returns index where "c" is first found, after index 5

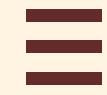
method chaining



Strip Methods

```
...  
>>> msg = "...end..."  
>>> msg.strip('.')  
end  
>>> msg.lstrip('.')  
end...  
>>> msg.rstrip('.')  
...end
```

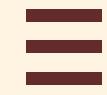




Find & Index

```
● ● ●  
>>> msg = "Cat in a hat"  
>>> msg.find('a')  
1  
>>> msg.rfind('a')  
10  
>>> msg.index('a')  
1
```





Replace & Count



```
>>> msg = "Hot dog"  
>>> msg.replace('o', 'u')  
Hut dug  
>>> msg.replace('o', 'u', 1)  
Hut dog  
>>> msg.count('o')  
2
```



BOOLEANS + COMPARISONS



Basic Data Types

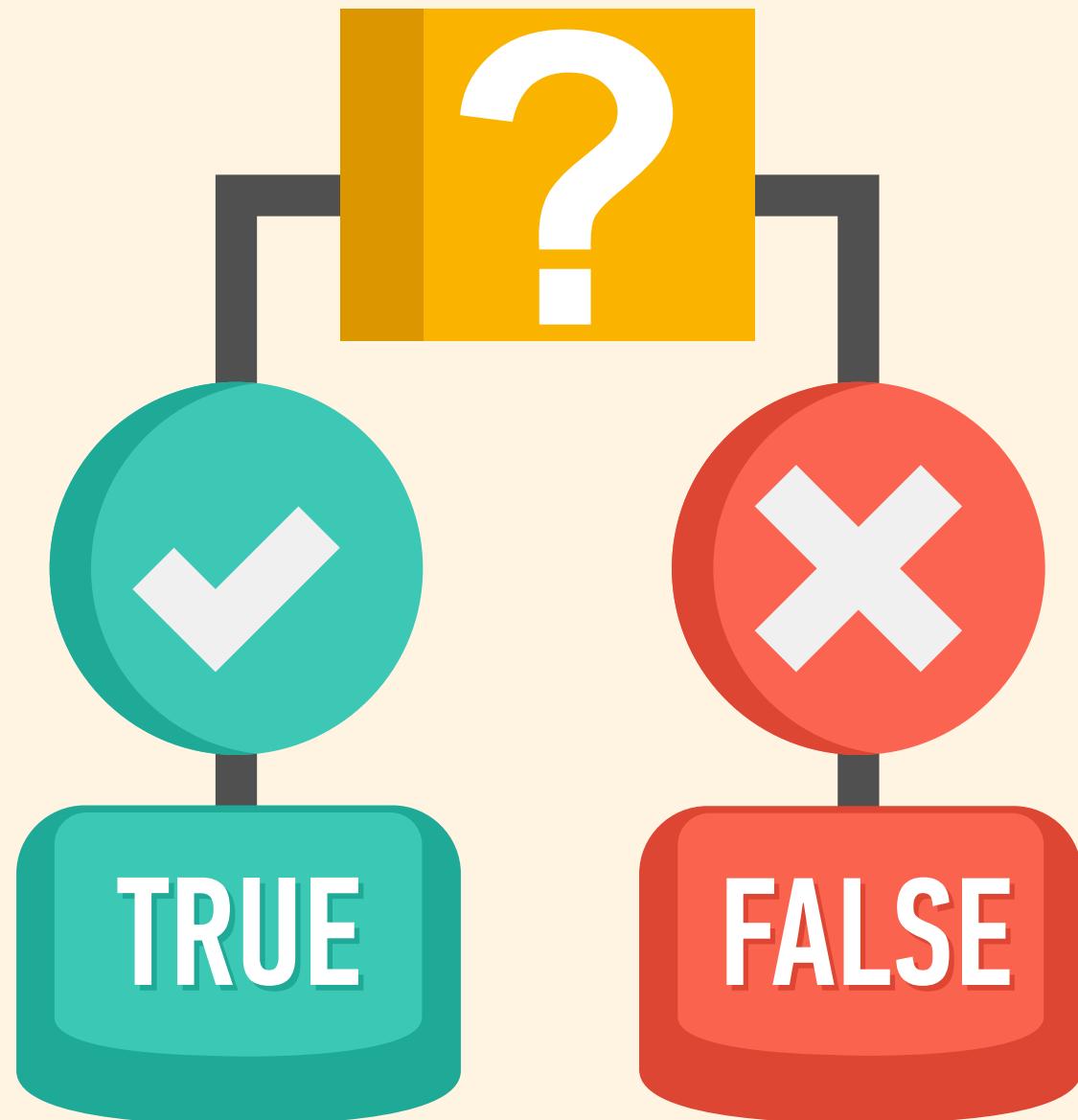
Strings

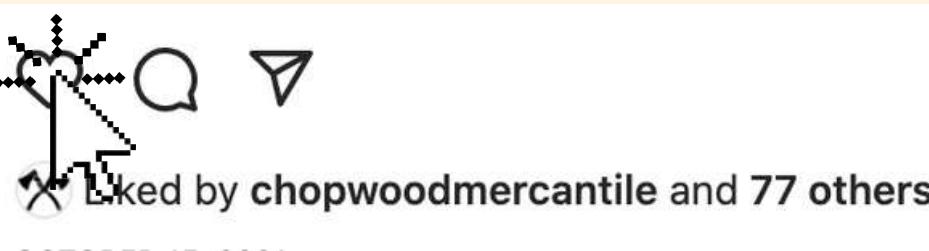
Integers

Booleans

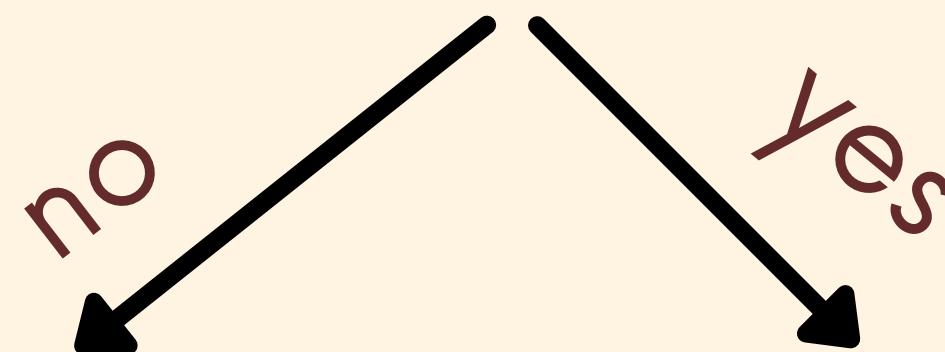
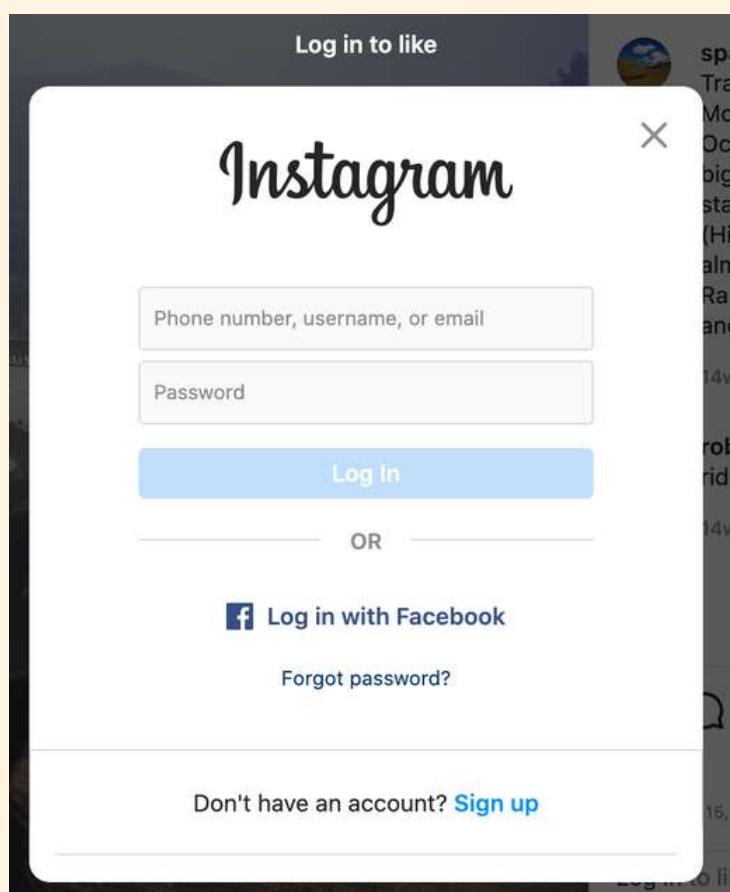
FLOATS

Decision Making



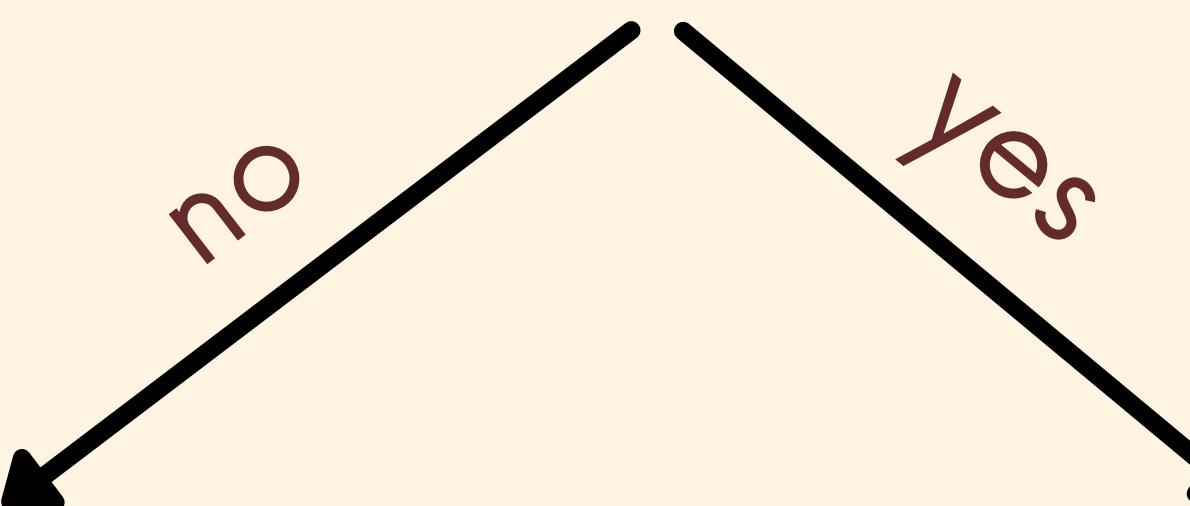


Is the user logged in?

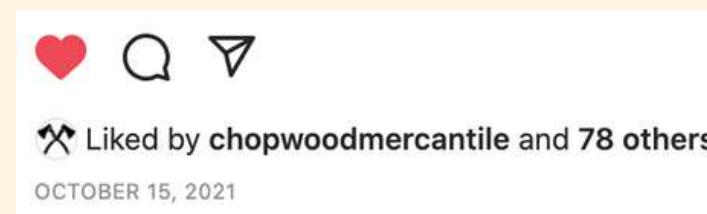


yes

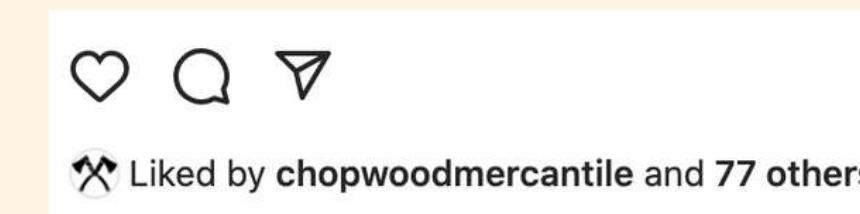
Has the user already liked the photo?



no



Like the photo



un-like the photo



Potential Decisions

Is the game over?

Does the user have any guesses left?

Is 'S' in the target word?

Is 'O' in the target word?

Is 'O' in the correct location?

Is 'U' in the target word?

Is 'N' in the target word?

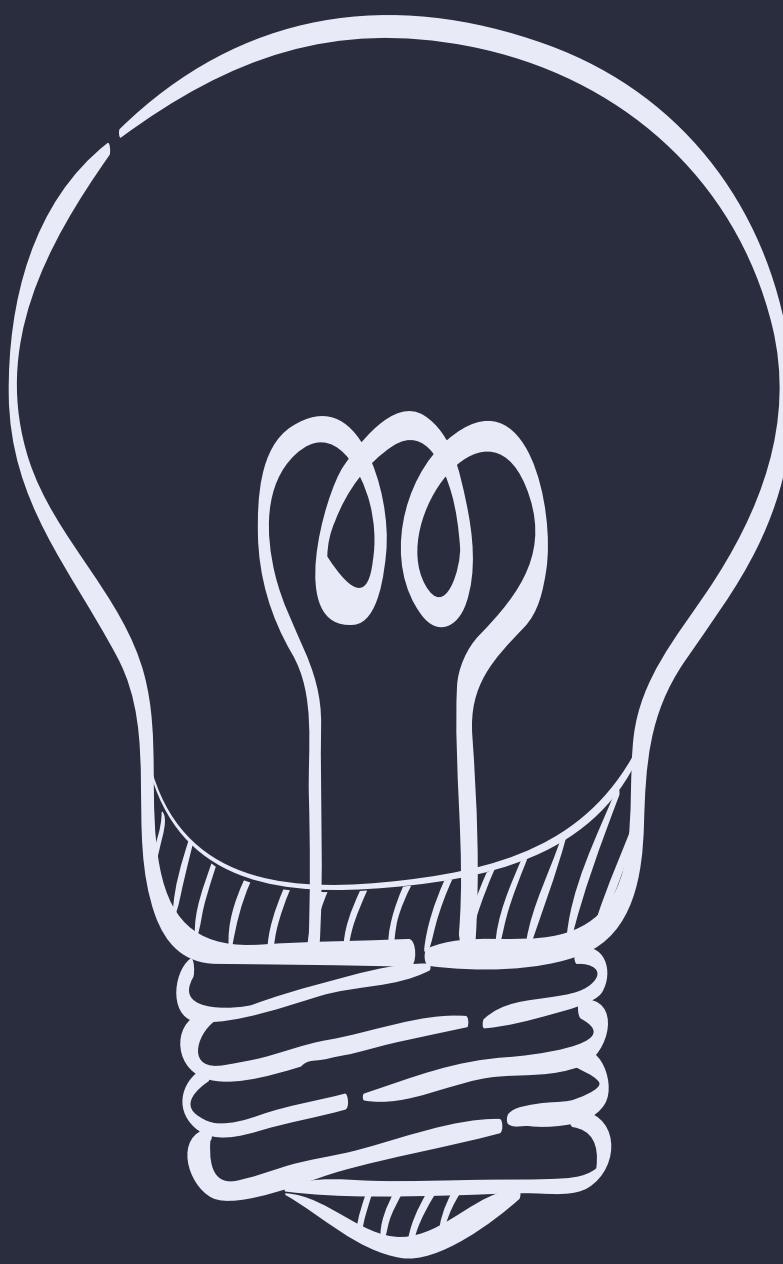
Is 'N' in the correct location?

Is 'D' in the target word?

ON



OFF





Booleans

Booleans are another basic Python type. There are only two possible values: **True** and **False**.

Notice the capitalization!!



```
...>>> True  
...>>> False
```





Booleans



```
>>> isAlive = True
```





Booleans

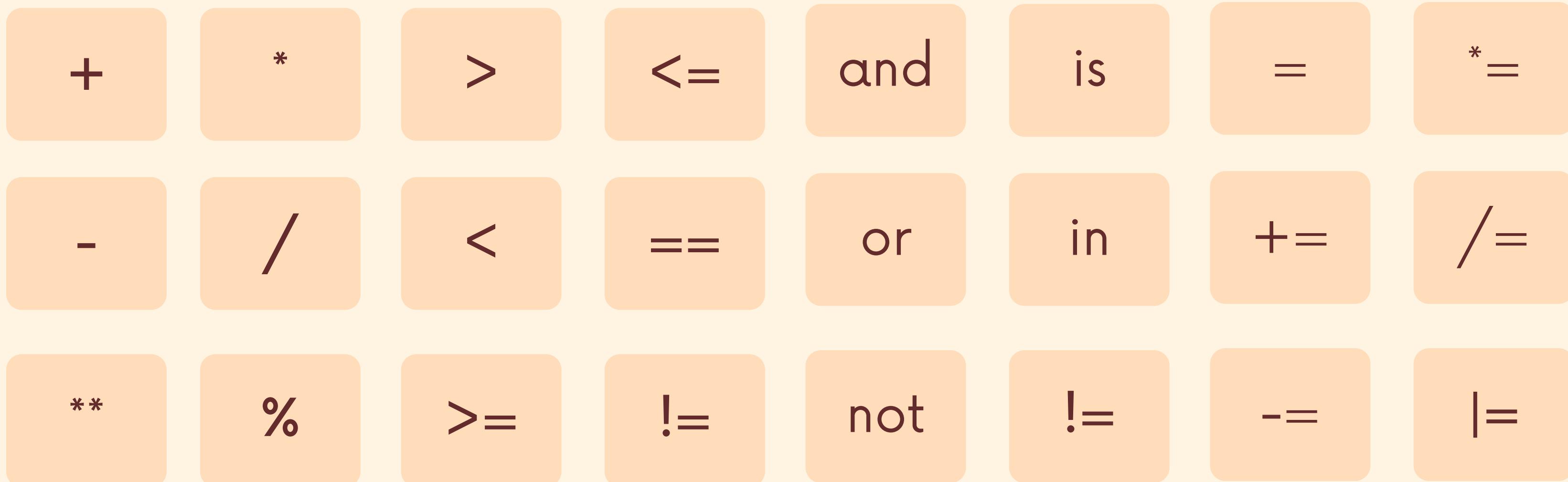


```
>>> isAlive = False
```



Operators

Operators are special characters in Python that perform operations on value(s). Below are some of the most common:



Comparisons

>

Greater Than

<

Less Than

>=

Greater Than Or Equal To

<=

Less Than Or Equal To

$a > b$

Truthy if a is greater than b

$a < b$

Truthy if a is less than b

$a \geq b$

Truthy if a is greater than or equal to b

$a \leq b$

Truthy if a is less than or equal to b



```
>>> age = 21
```



```
>>> age > 18  
True
```



```
>>> age > 35  
False
```



```
>>> age >= 21  
True
```

Comparisons

`==`

Equal To

`!=`

Not Equal To



```
>>> age = 21
```



```
age == 21  
True
```



```
age == 25  
False
```



```
age != 29  
True
```

Identity

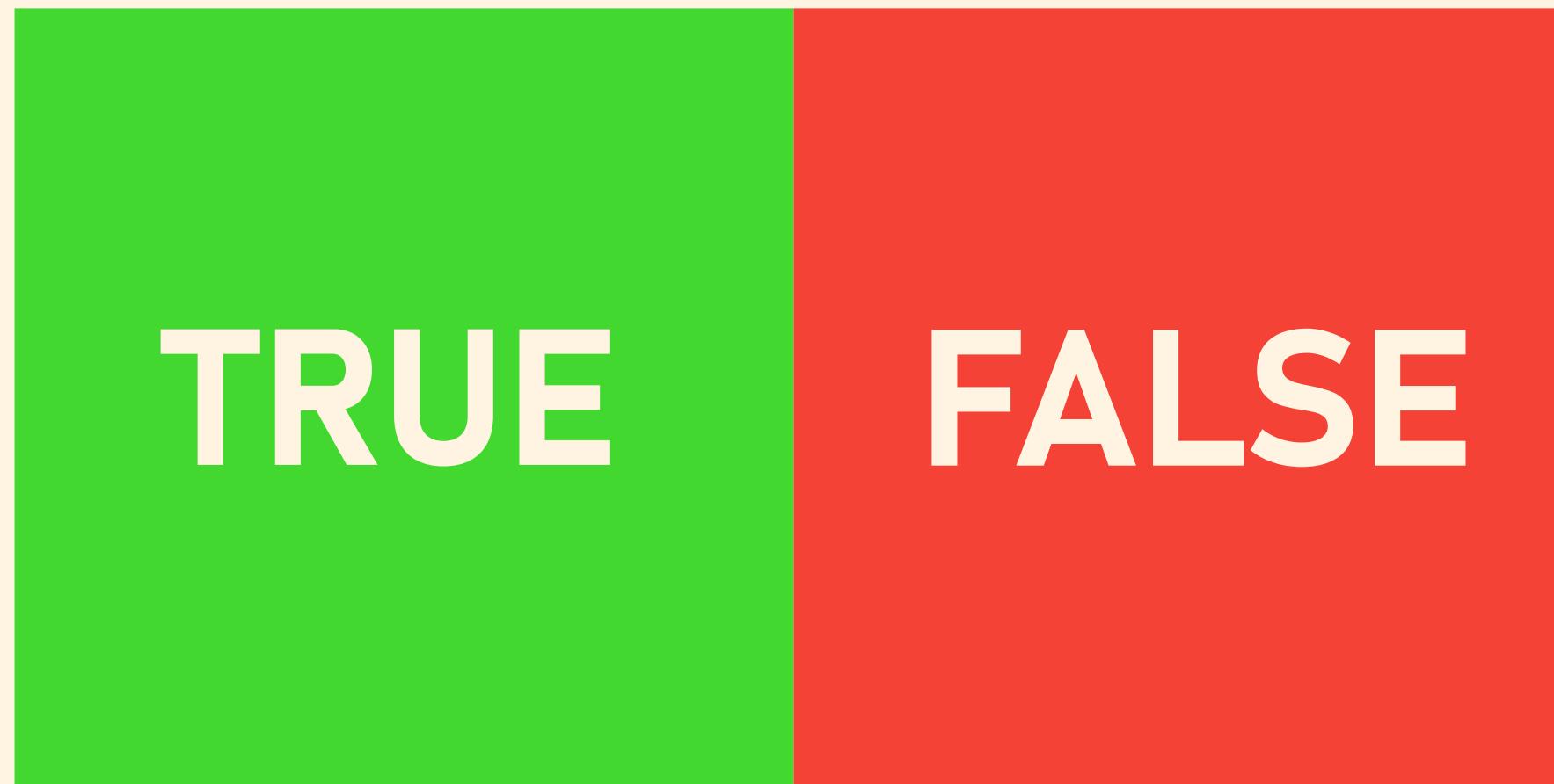
is

Evaluates to True if a and b both refer to the same object in memory

is not

Evaluates to True if a and b do NOT refer to the same object in memory

**Every value is inherently
Truth-y or False-y in Python**



False-y

False

0.0

0

None

range(0)

Empty Strings:

"

""

""""

""""""

Empty Data Structures:

[]

{}

{}{}}

set()

Truthy

Everything Else!



bool()

Just as we can use `int()`, `float()`, and `str()` to cast values, we can use `bool()` to cast a value to a Boolean.

This is one way to determine whether Python considers a value to be Truth-y or False-y

```
>>> bool()
```



String Comparison



```
>>> str1='ABC'  
>>> str2='AbC'  
>>> str3='ABC'
```

Name Object

str1

0	A	1	B	2	C
---	---	---	---	---	---

str2

0	A	1	b	2	C
---	---	---	---	---	---

str3

0	A	1	B	2	c
---	---	---	---	---	---

String Comparison

```
•••  
->>> ord('A')
```

```
65  
->>> ord('B')
```

```
66  
->>> ord('C')
```

```
67  
->>> ord('b')
```

```
98  
->>> ord('c')
```

99

Name

str1

ord()

0	A	1	B	2	C
	65		66		67

str2

ord()

0	A	1	b	2	C
	65		98		67

str3

ord()

0	A	1	B	2	c
	65		66		99

Object

String Comparison



```
>>> str1 > str2
```

False

```
>>> str2 > str3
```

True

```
>>> str1 > str3
```

False

Name

str1

0	A	1	B	2	C
	65		66		67

str2

0	A	1	b	2	c
	65		98		67

str3

0	A	1	B	2	c
	65		66		99

Object

logical and

The **and** operator will evaluate to True only if both the left and right sides evaluate to True.

```
'a' == 'a' and 1 < 5
```

```
True
```

logical and

The **and** operator will evaluate to True only if both the left and right sides evaluate to True.

```
'a' == 'a' and 1 < 5
```

```
True
```

logical and

The **and** operator will evaluate to True only if both the left and right sides evaluate to True.

```
'a' == 'a' and 1 < 5
```

True



```
>>> age = 18
>>> age > 10 and age < 21
True
```

left and **right** → **False**

left and **right** → **False**

left and **right** → **False**

left and **right** → **True**

logical or

The **or** operator will evaluate to True if one or both the left or right sides evaluate to True.

```
'a' == 'b' or 1 < 5
```

```
True
```

logical or

The **or** operator will evaluate to True if one or both the left or right sides evaluate to True.

```
'a' == 'b' or 1 < 5
```

True

left or right → **False**

left or right → **True**

left or right → **True**

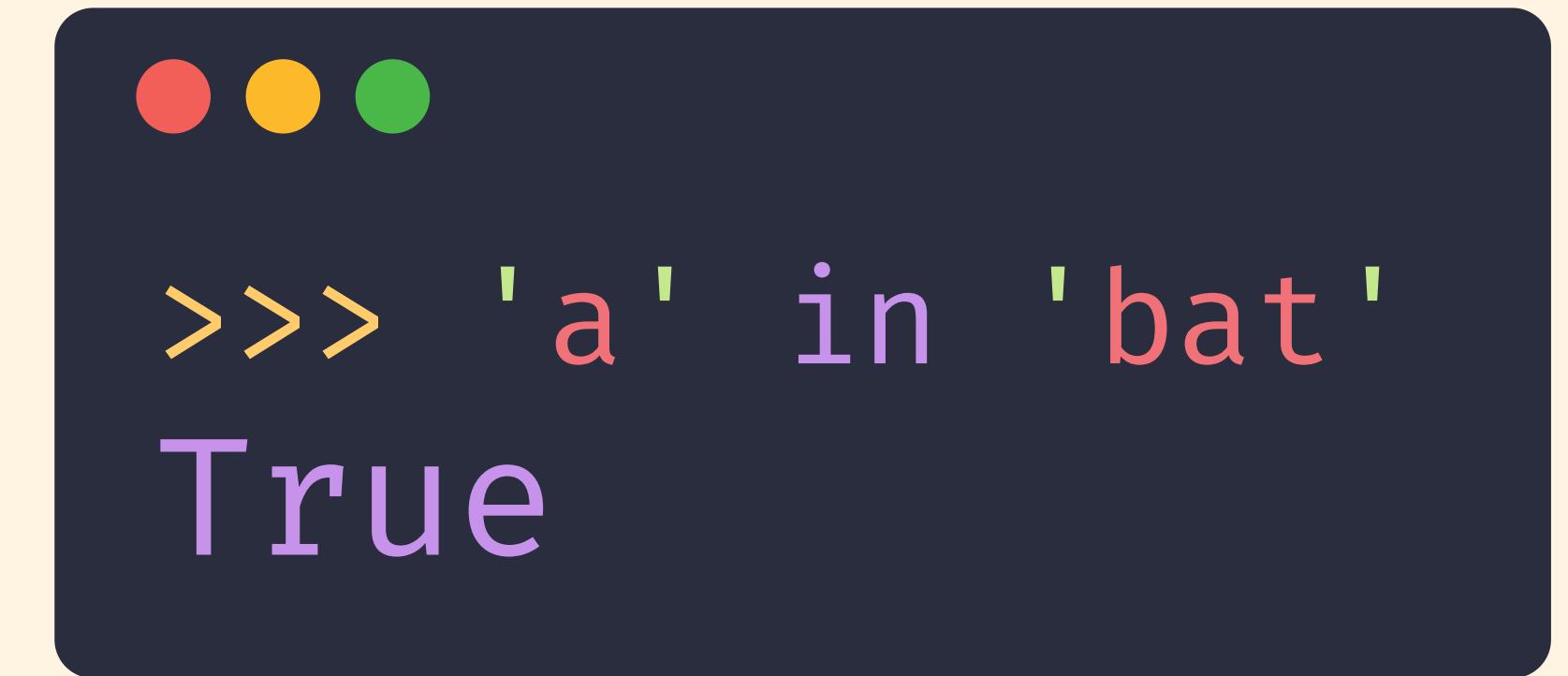
left or right → **True**



in

The "in" operator looks to see if an item is a member of a sequence.

Soon we'll see other sequences types!



A dark-themed terminal window with three colored dots (red, yellow, green) at the top. It displays the Python code `>>> 'a' in 'bat'` followed by the output `True`.

```
>>> 'a' in 'bat'  
True
```



logical not

The **not** operator changes True to False and False to True. It negates expressions.

```
1 < 5
```

True

```
not 1 < 5
```

False

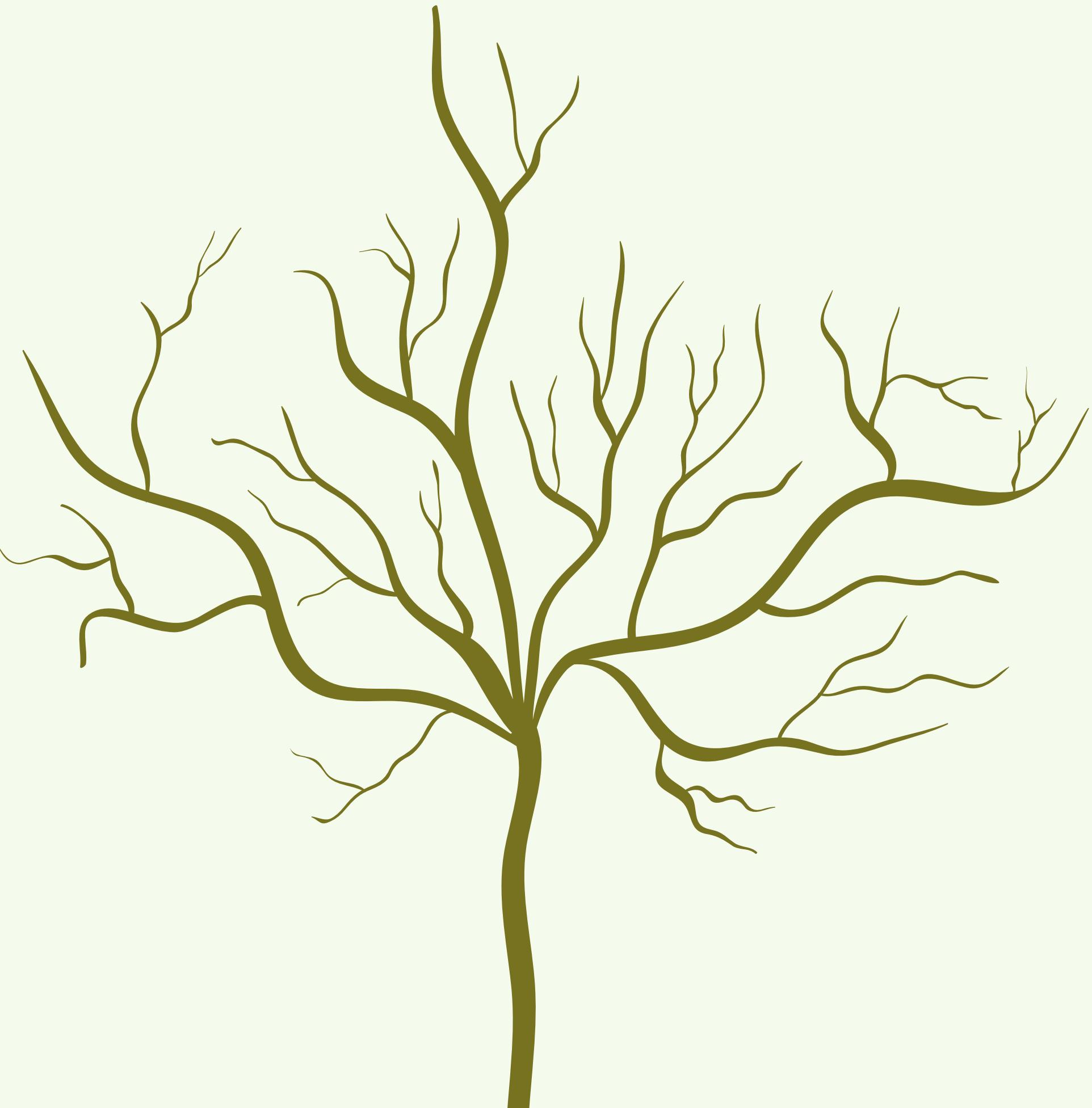
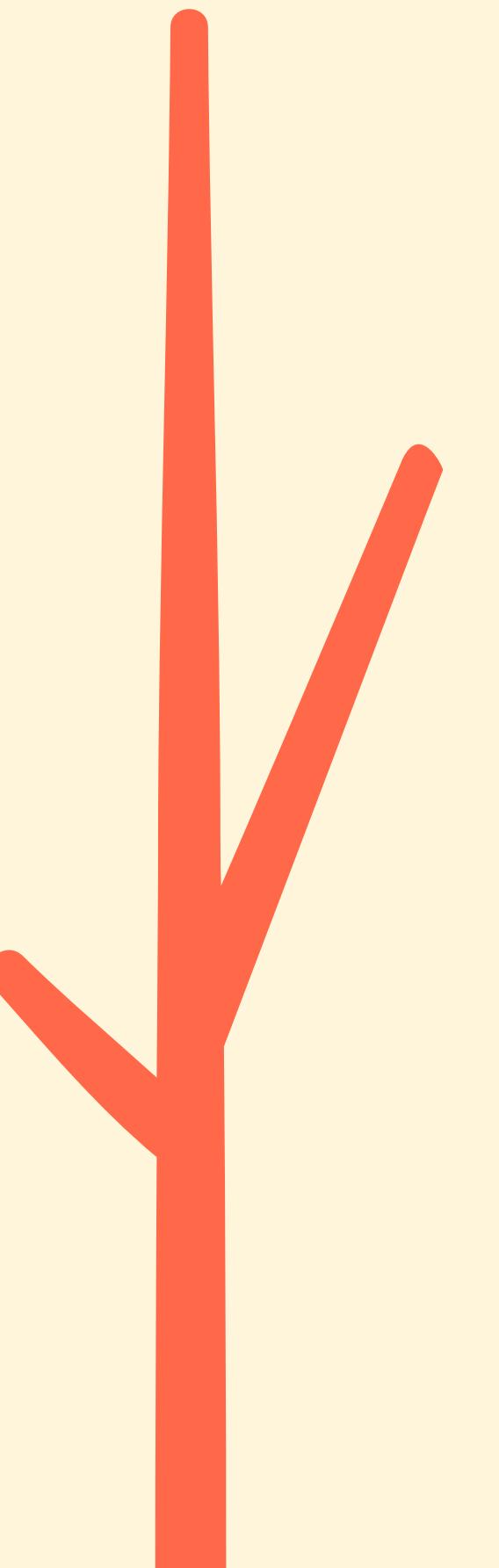
Conditionals making decisions



Conditionals

How do we make decisions with code
and have different outcomes??









Decisions

Are You Over 21?

yes

Come on in!

no

Go home kid.





age \geq 21

Are You Over 21?

yes

Come on in!

no

Go home kid.



Are You Over 21?

yes

Come on in!

no

Go home kid.



```
age >= 21
```

Are You Over 21?

yes

Come on in!

no

Go home kid.



```
print("Come on in!")
```



```
age >= 21
```

Are You Over 21?

yes



Come on in!

no



Go home kid.



```
print("Come on in!")
```



```
print("Go home kid.")
```

```
● ● ●  
age >= 21
```

Are You Over 21?

yes

HOW???

no

Come on in!

Go home kid.

```
● ● ●
```

```
print("Come on in!")
```

```
● ● ●
```

```
print("Go home kid.")
```

Conditionals

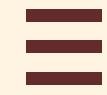
```
...  
if age >= 21:  
    print("Come on in!")  
else:  
    print("Go home kid")
```

≡

if Statement

```
if
```

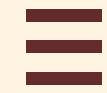




if Statement

```
if condition
```

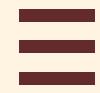




if Statement

```
if condition:
```

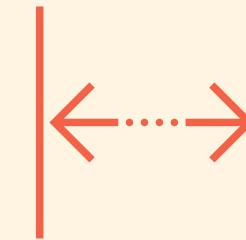




if Statement

```
if condition:
```

Indent - 4 Spaces



*4 spaces is standard, but you can use
fewer as long as you are consistent

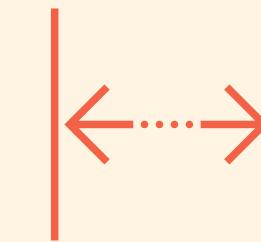




if Statement

```
if condition:
```

Indent - 4 Spaces



```
code that runs if  
condition is True
```

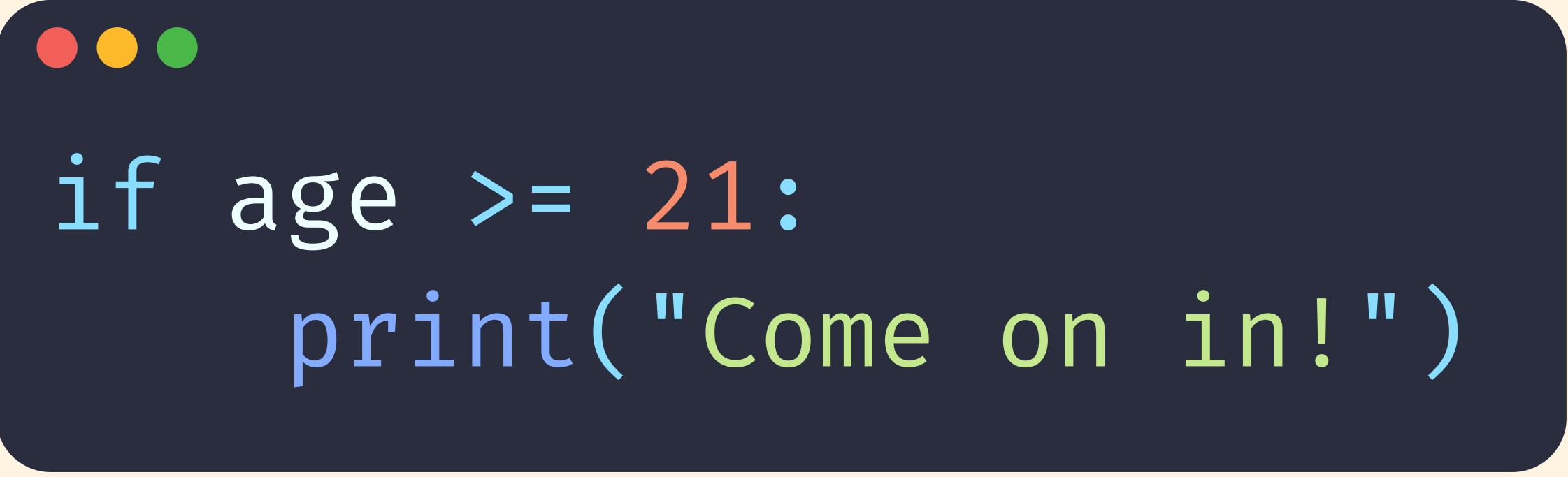




```
if age >= 21:  
    print("Come on in!")
```



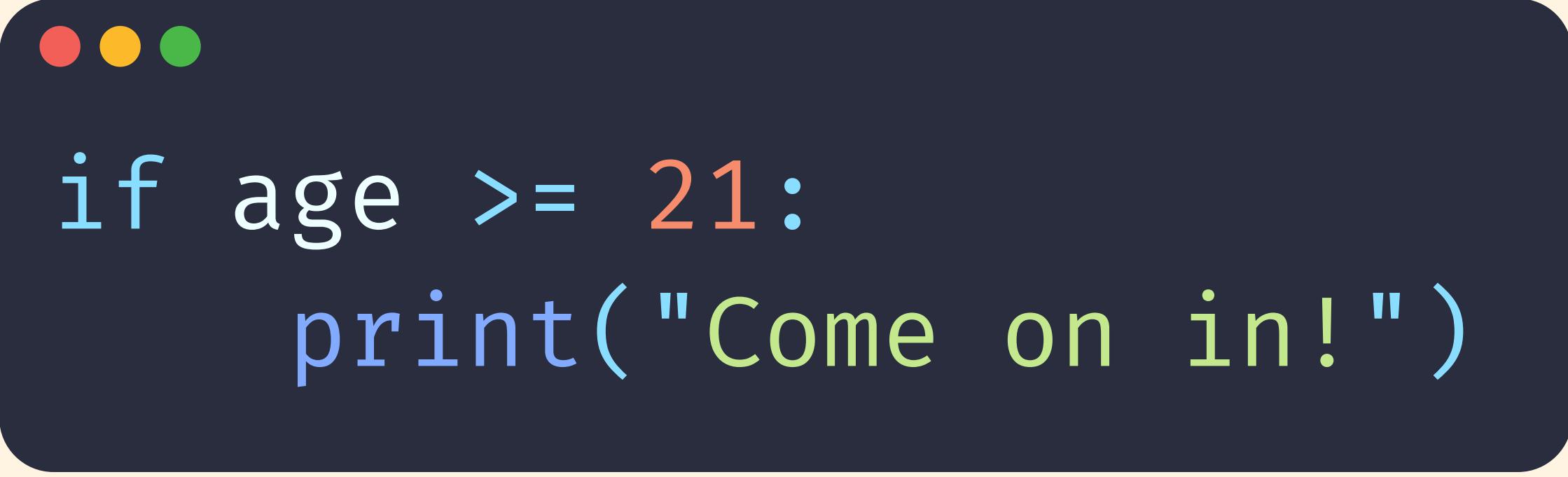
```
age = 62
```



```
if age >= 21:  
    print("Come on in!")
```



```
age = 62
```



```
if age >= 21:  
    print("Come on in!")
```

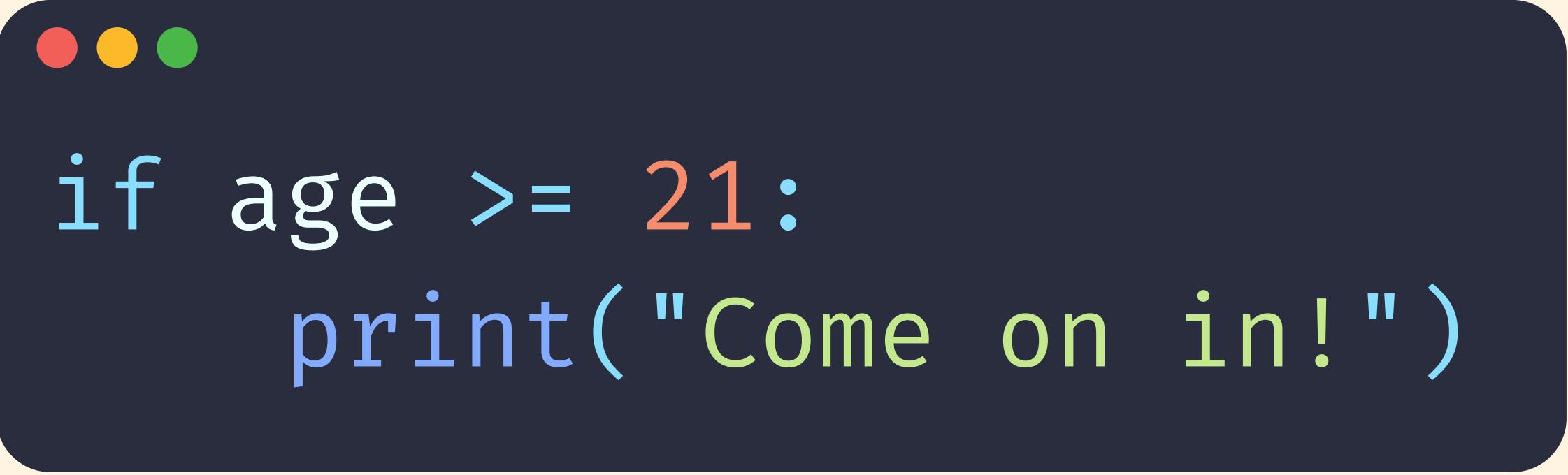
"Come on in!"



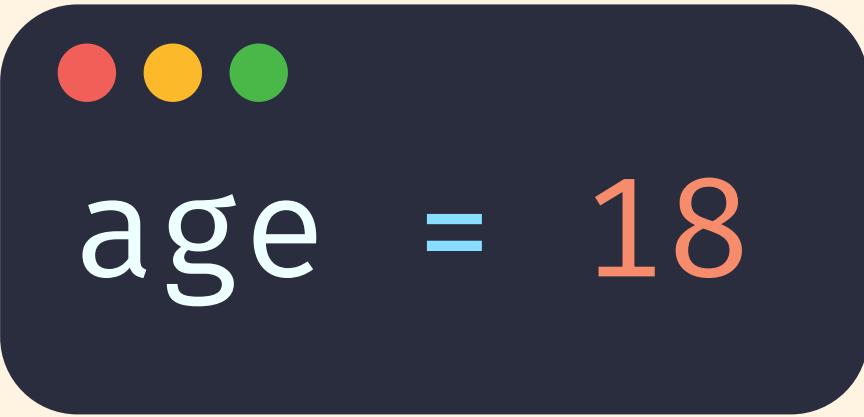
```
if age >= 21:  
    print("Come on in!")
```



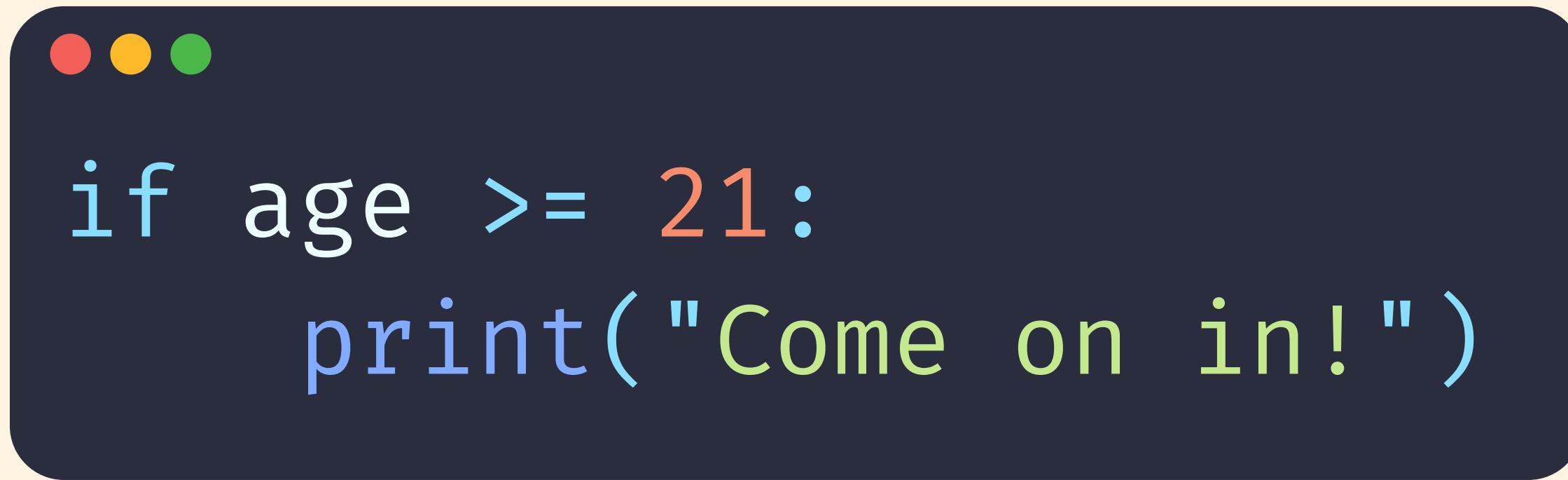
```
age = 18
```



```
if age >= 21:  
    print("Come on in!")
```

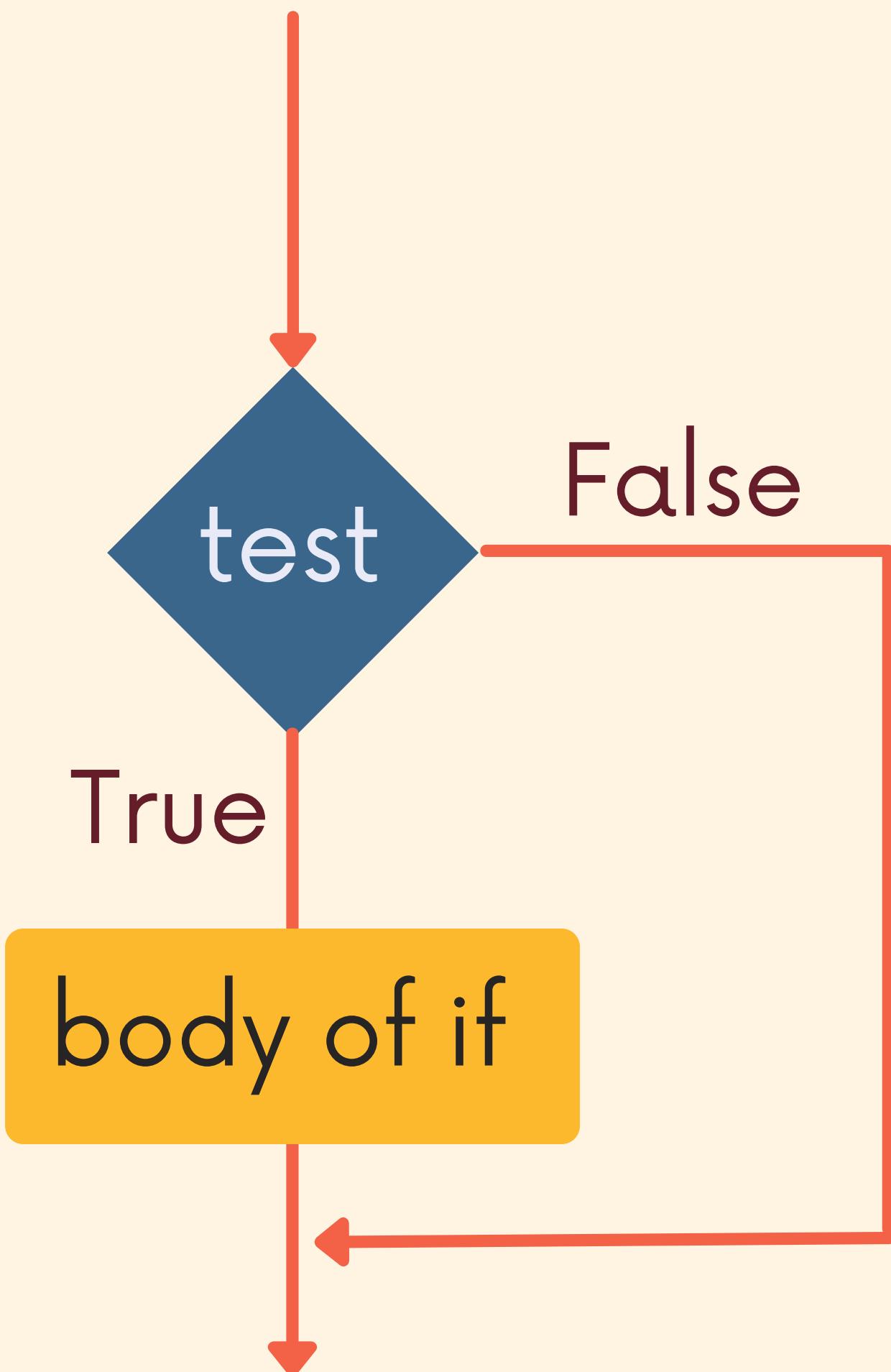


age = 18



```
if age >= 21:  
    print("Come on in!")
```

Nothing Printed



Else if

If not the first thing,
maybe try this instead?

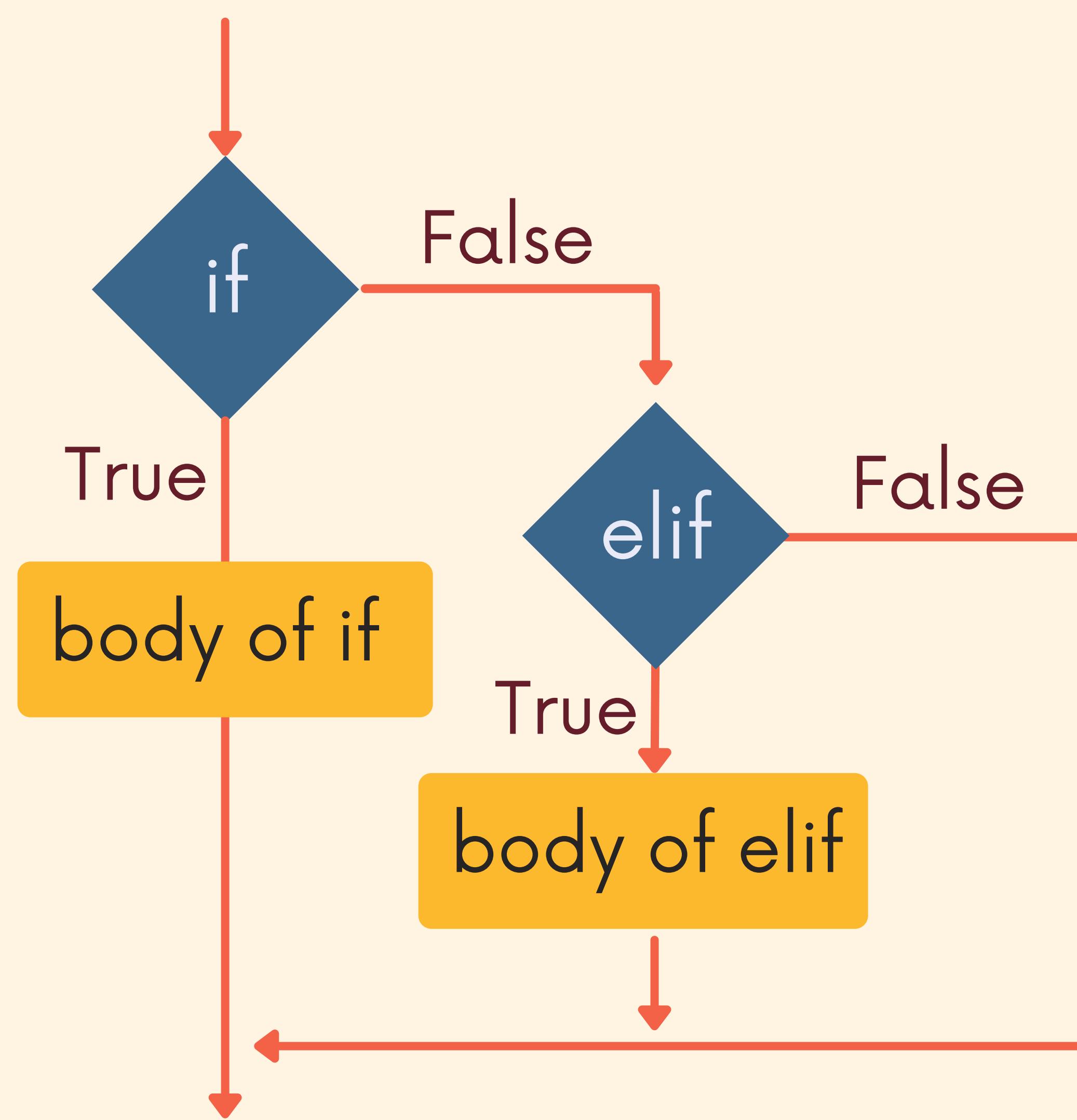
Else If

```
if test:
```

```
    code if True
```

```
elif test 2:
```

```
    code if True
```





```
if color == "green":  
    print("Beginner Ski Run")  
elif color == "blue":  
    print("Intermediate Ski Run")
```

```
color = "blue"
```

```
if color == "green":  
    print("Beginner Ski Run")  
elif color == "blue":  
    print("Intermediate Ski Run")
```

```
color = "blue"
```

```
if color = "green":  
    print("Beginner Ski Run")  
elif color = "blue":  
    print("Intermediate Ski Run")
```

"Intermediate Ski Run"



```
if color == "green":  
    print("Beginner Ski Run")  
elif color == "blue":  
    print("Intermediate Ski Run")
```

```
color = "green"
```

```
if color == "green":  
    print("Beginner Ski Run")  
elif color == "blue":  
    print("Intermediate Ski Run")
```

```
color = "green"
```

```
if color == "green":  
    print("Beginner Ski Run")  
elif color == "blue":  
    print("Intermediate Ski Run")
```

"Beginner Ski Run"



```
if color == "green":  
    print("Beginner Ski Run")  
elif color == "blue":  
    print("Intermediate Ski Run")  
elif color == "black":  
    print("Expert Ski Run")
```

```
color = "black"
```

```
if color == "green":  
    print("Beginner Ski Run")  
elif color == "blue":  
    print("Intermediate Ski Run")  
elif color == "black":  
    print("Expert Ski Run")
```

```
color = "black"
```

```
if color = "green":  
    print("Beginner Ski Run")  
elif color = "blue":  
    print("Intermediate Ski Run")  
elif color = "black":  
    print("Expert Ski Run")
```

"Expert Ski Run"

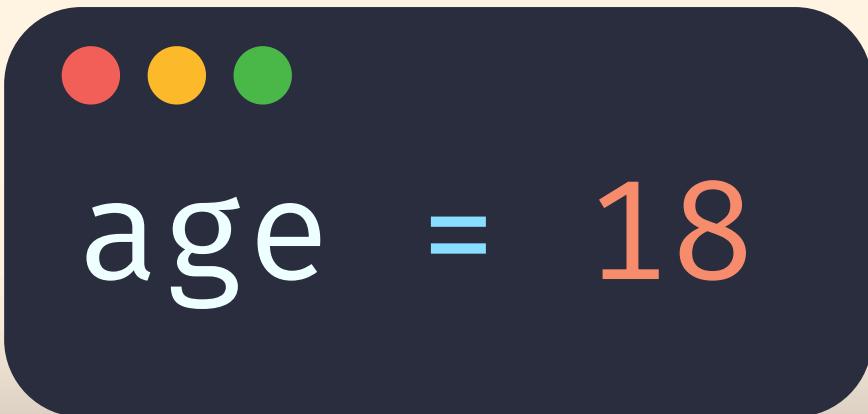
Else

If none of the above
were True, do this...

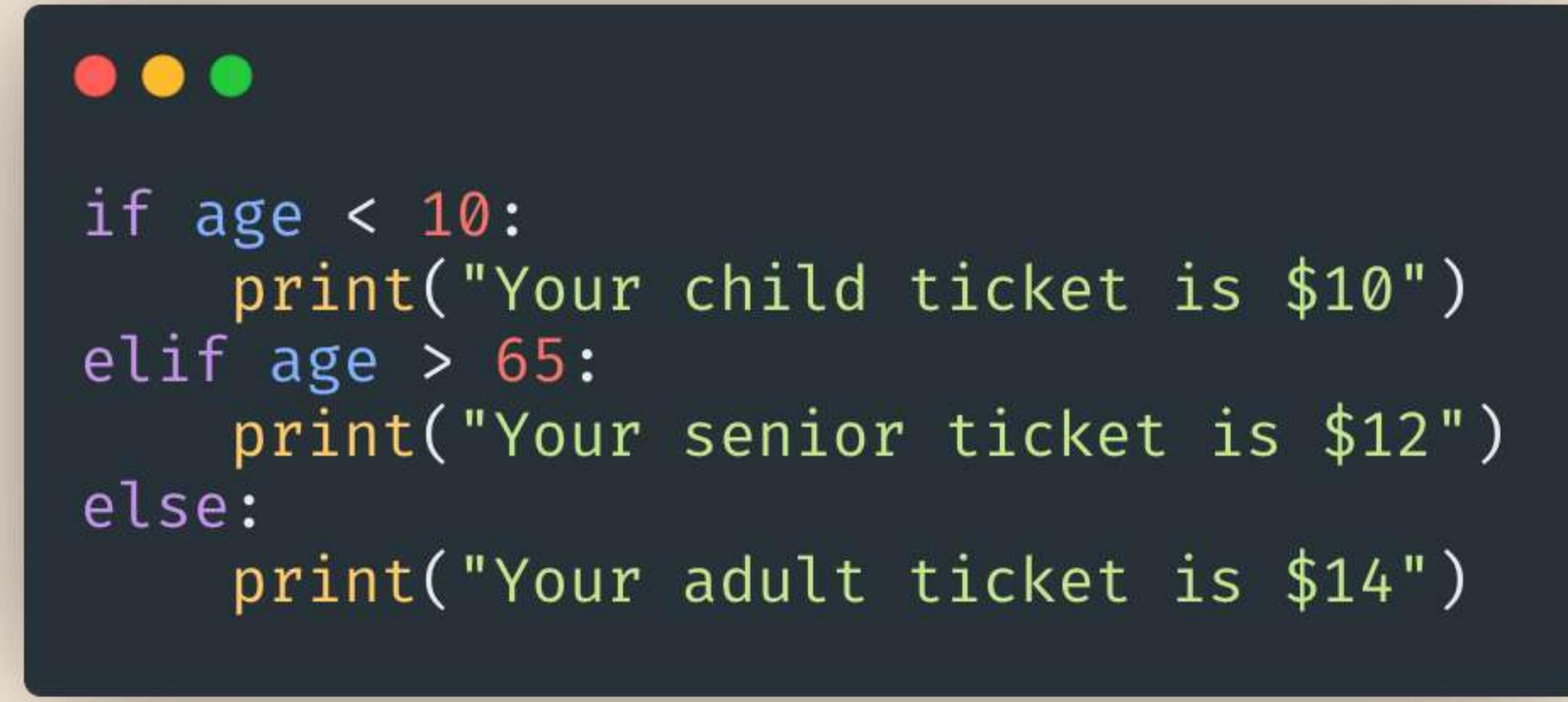
```
if age >= 21:  
    print("Come on in!")  
else:  
    print("Go home kid")
```



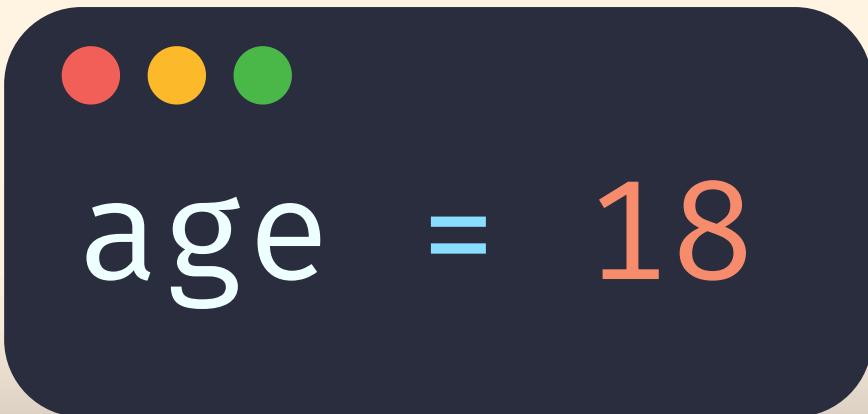
```
if age < 10:  
    print("Your child ticket is $10")  
elif age > 65:  
    print("Your senior ticket is $12")  
else:  
    print("Your adult ticket is $14")
```



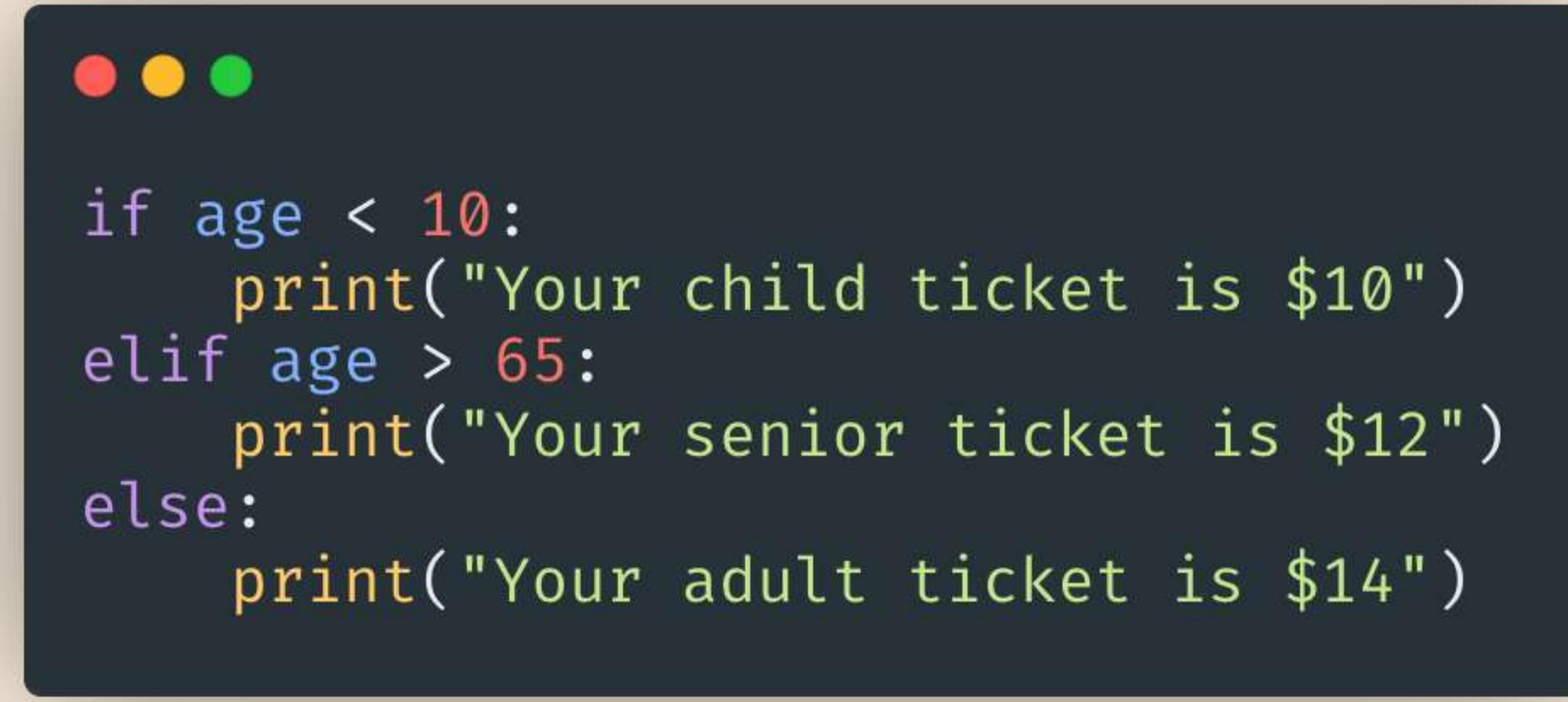
age = 18



```
if age < 10:  
    print("Your child ticket is $10")  
elif age > 65:  
    print("Your senior ticket is $12")  
else:  
    print("Your adult ticket is $14")
```

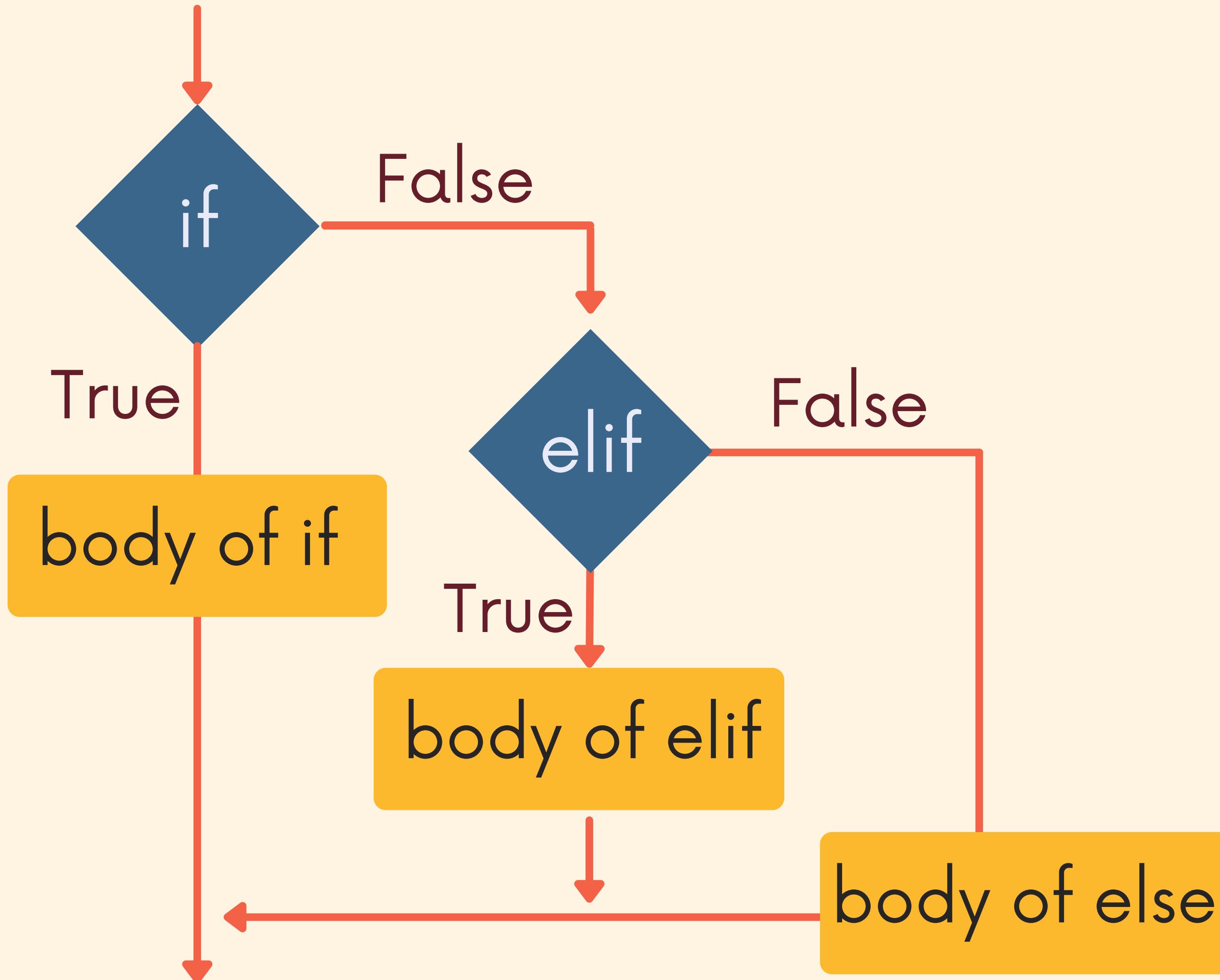


age = 18



```
if age < 10:  
    print("Your child ticket is $10")  
elif age > 65:  
    print("Your senior ticket is $12")  
else:  
    print("Your adult ticket is $14")
```

"Your adult ticket is \$14"



Nested Conditionals





```
if unit == 'fahrenheit':  
    if temp <= 32:  
        print("It's freezing out!")  
    else:  
        print("It's not freezing ")  
else:  
    print("I'm really bad with celsius!")
```

```
● ● ●  
temp = 30  
unit = 'fahrenheit'
```

```
● ● ●  
  
if unit == 'fahrenheit':  
    if temp <= 32:  
        print("It's freezing out!")  
    else:  
        print("It's not freezing ")  
else:  
    print("I'm really bad with celsius!")
```

```
● ● ●  
temp = 30  
unit = 'fahrenheit'
```

```
● ● ●  
  
if unit == 'fahrenheit':  
    if temp <= 32:  
        print("It's freezing out!")  
    else:  
        print("It's not freezing ")  
else:  
    print("I'm really bad with celsius!")
```

It's freezing out!

```
● ● ●  
temp = 39  
unit = 'fahrenheit'
```

```
● ● ●  
  
if unit == 'fahrenheit':  
    if temp <= 32:  
        print("It's freezing out!")  
    else:  
        print("It's not freezing ")  
else:  
    print("I'm really bad with celsius!")
```

It's not freezing

```
● ● ●  
temp = 39  
unit = 'celsius'
```

```
● ● ●  
  
if unit == 'fahrenheit':  
    if temp <= 32:  
        print("It's freezing out!")  
    else:  
        print("It's not freezing ")  
else:  
    print("I'm really bad with celsius!")
```

I'm really bad with celsius!

Generating Random Numbers



```
import random  
random.randint(1,6)
```

6

```
random.randint(1,6)
```

2

Loops

Repeat Stuff



"Repeat this code exactly 7 times"

"Print out every name in the contacts list"

"Keep playing the game until a user quits"

While



```
while expression:  
    statement
```

For



```
for item in iterable:  
    statement
```

While

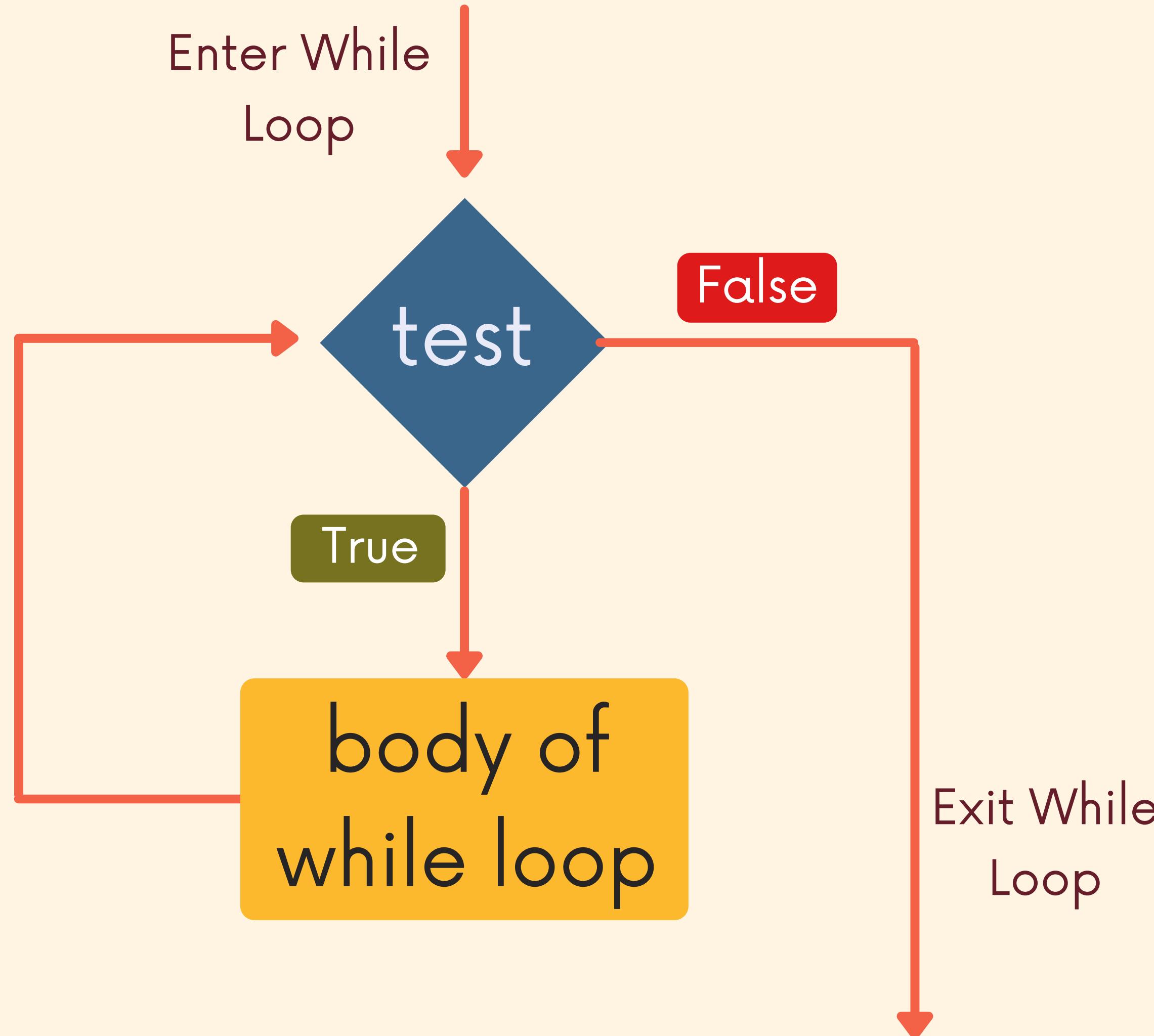
```
while expression:  
    statement
```

Loop repeats as long as expression is True.



```
answer = input("Please say hi ")  
while answer != "hi":  
    answer = input("Rude. Say hi... ")  
print("Thank you. Hi to you too!")
```

Please say hi: no
Rude. Say hi... ugh
Rude. Say hi... stfu
Rude. Say hi... hi
Thank you. Hi to you too!





while Loop Constructs



```
while True:  
    if condition:  
        break
```



```
count = 1  
while count<5:  
    count +=1
```



Name

Object

Loop Count: 0

```
count = 1
while count <=2:
    count +=1
```

Name

Object

Loop Count: 0

```
count = 1
while count <=2:
    count +=1
```

Loop Count: 0



```
count = 1  
while count <=2:  
    count +=1
```



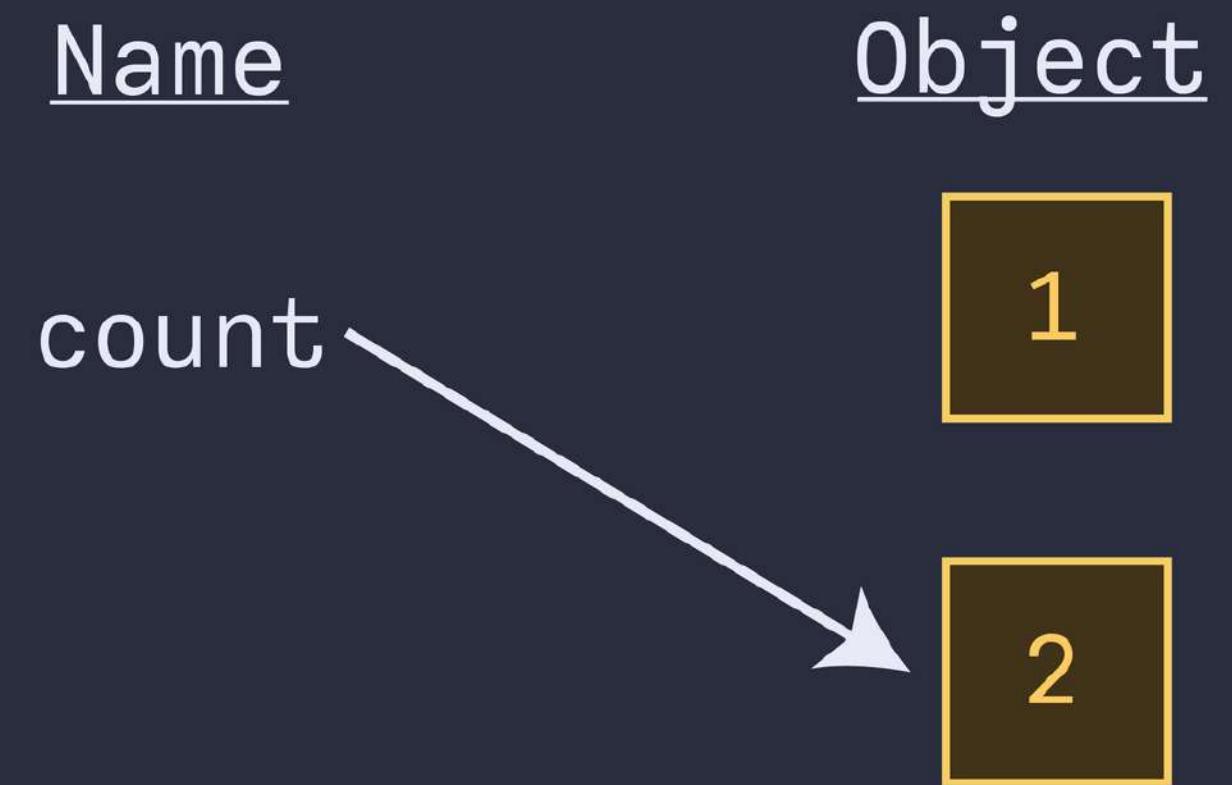
True!
Loop Count: 1



```
count = 1
while count <=2: ←
    count +=1
```

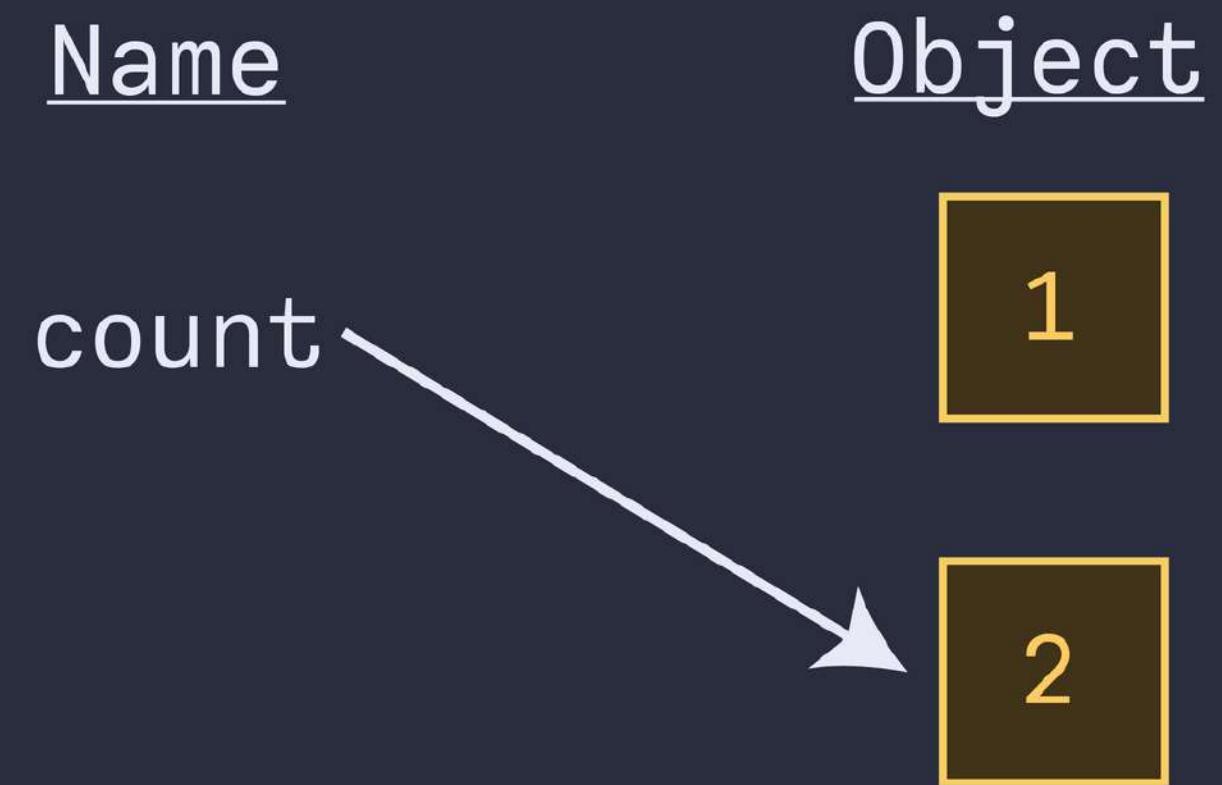
Loop Count: 1

```
count = 1
while count <=2:
    count +=1
```



True!
Loop Count: 2

```
count = 1  
while count <=2:  
    count +=1
```



Loop Count: 2

```
count = 1
while count <=2:
    count +=1
```

Name Object

count

1

2

3



False

Loop Count: 2

count = 1

while count <=2:

 count +=1

Name

count

Object

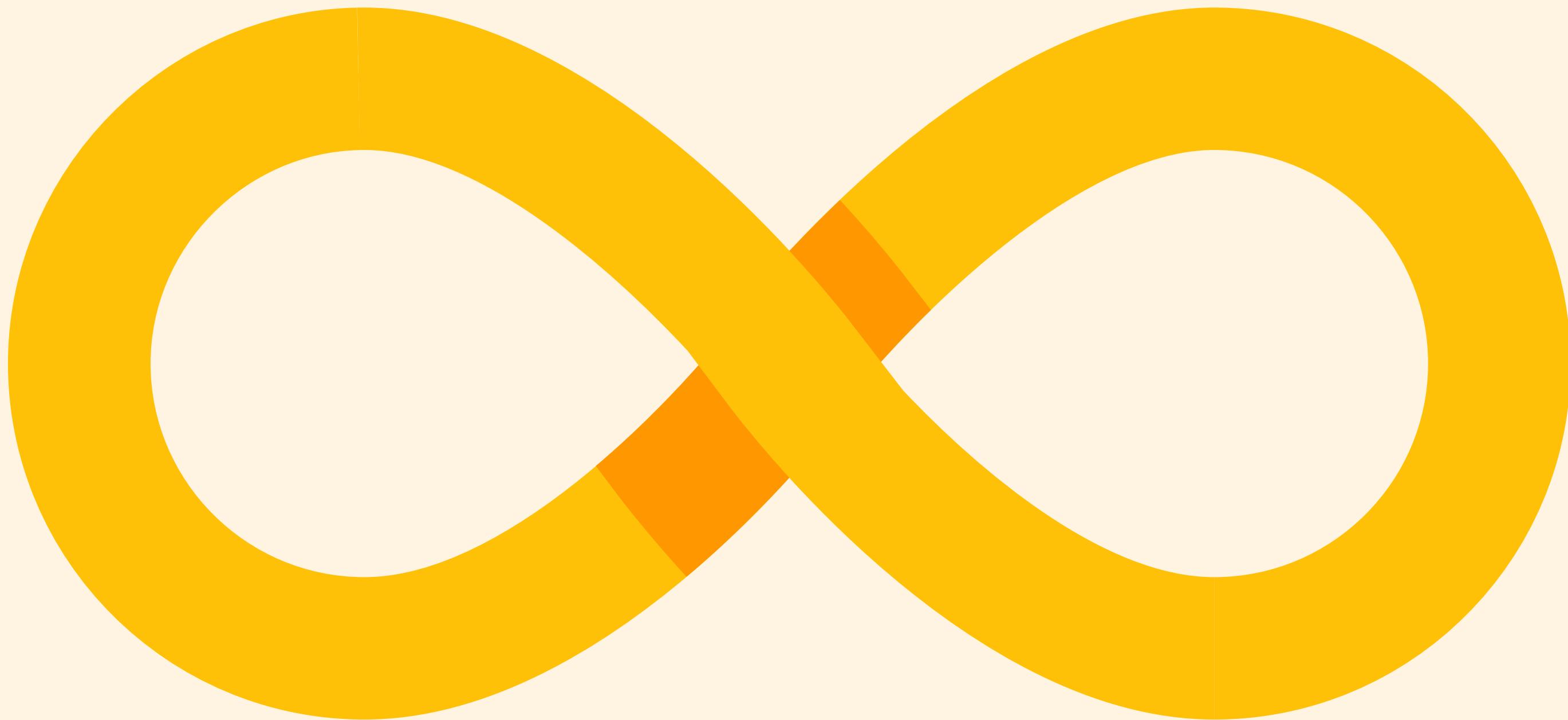
1

2

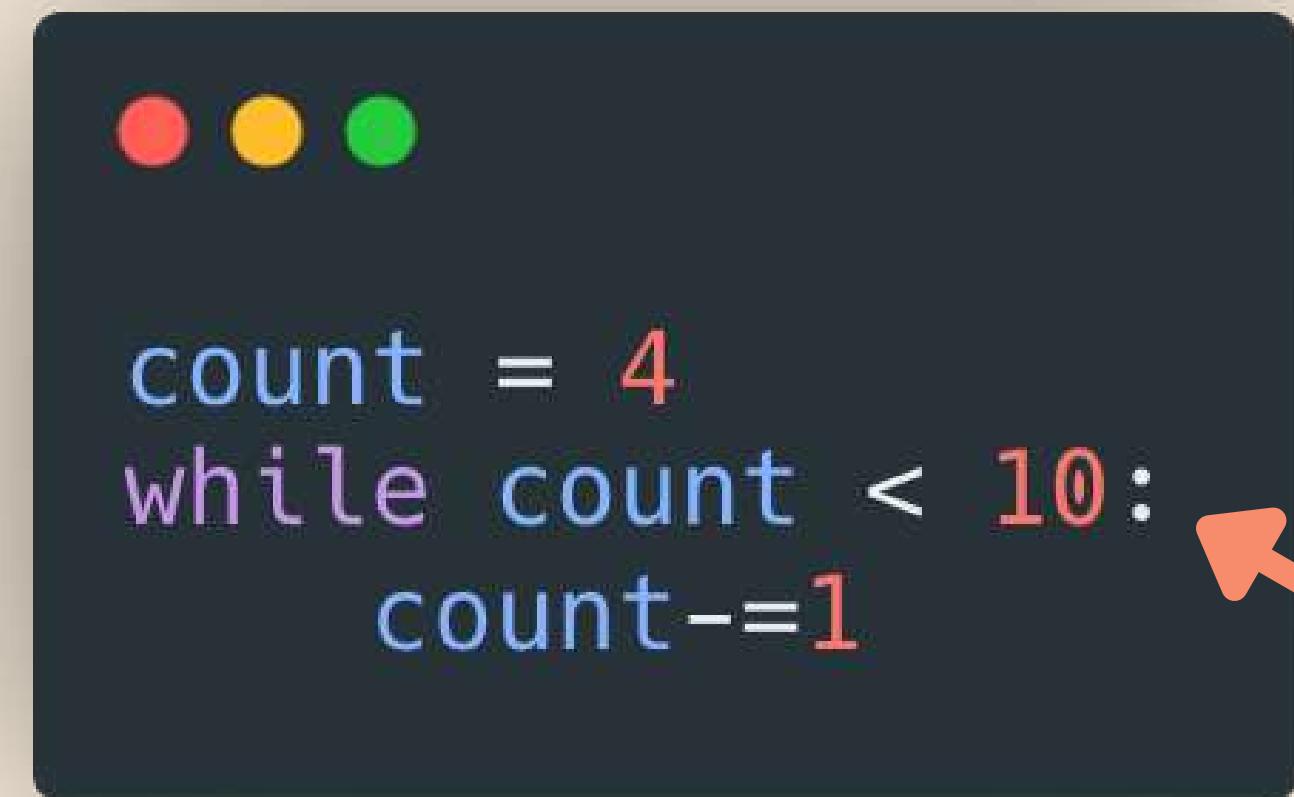
3



Avoid Infinite Loops!



Avoid Infinite Loops!



```
count = 4
while count < 10:
    count-=1
```



This will
ALWAYS
be True

For



```
for item in iterable:  
    statement
```



```
word = "Hello"  
for char in word:  
    print(char)
```



Loop Count: 0

```
word = "Hello"  
for char in word:  
    print(char)
```

Name

Object

Output

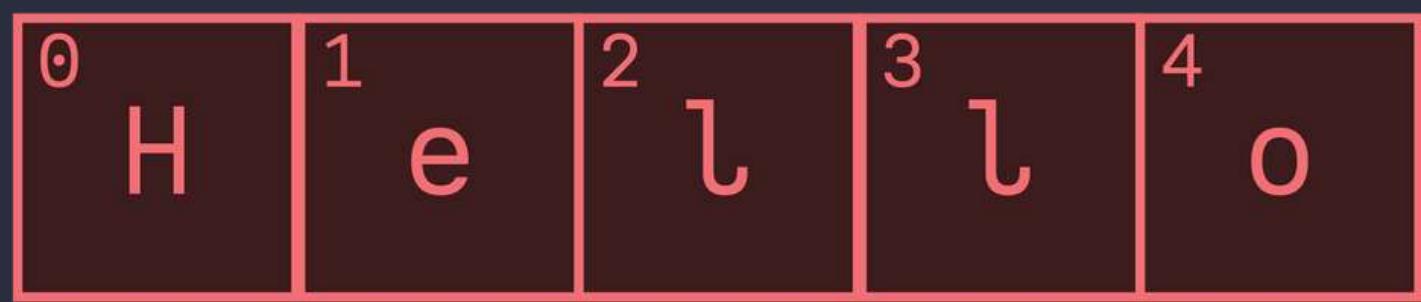
Loop Count: 0

```
word = "Hello"
```

```
for char in word:  
    print(char)
```

Name

Object



Output

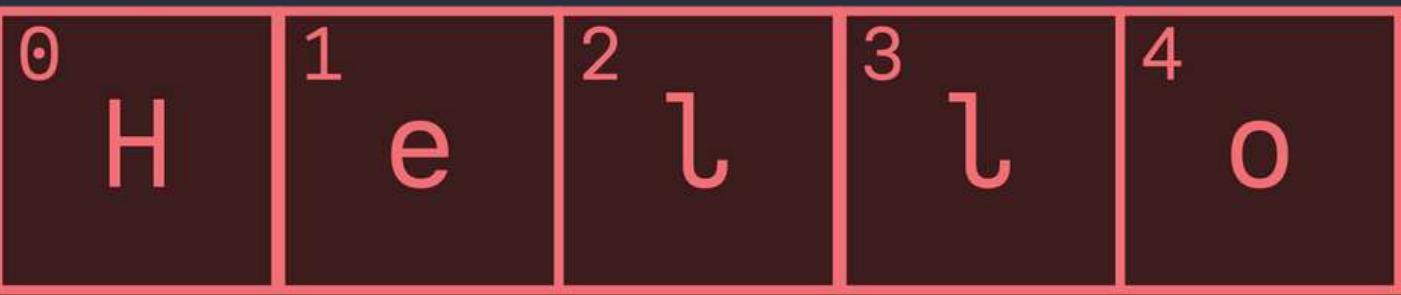
Loop Count: 1

```
word = "Hello"  
for char in word:  
    print(char)
```

Name

char

Object



Output

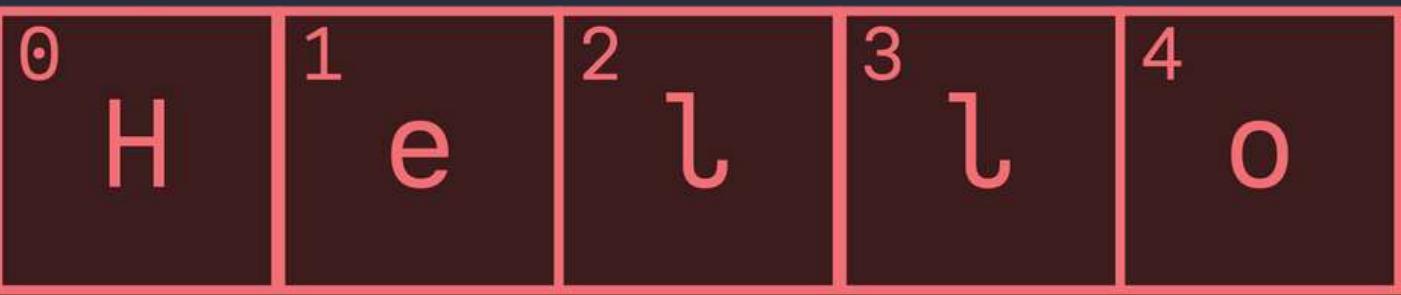
Loop Count: 1

```
word = "Hello"  
for char in word:  
    print(char)
```

Name

char

Object



Output

H

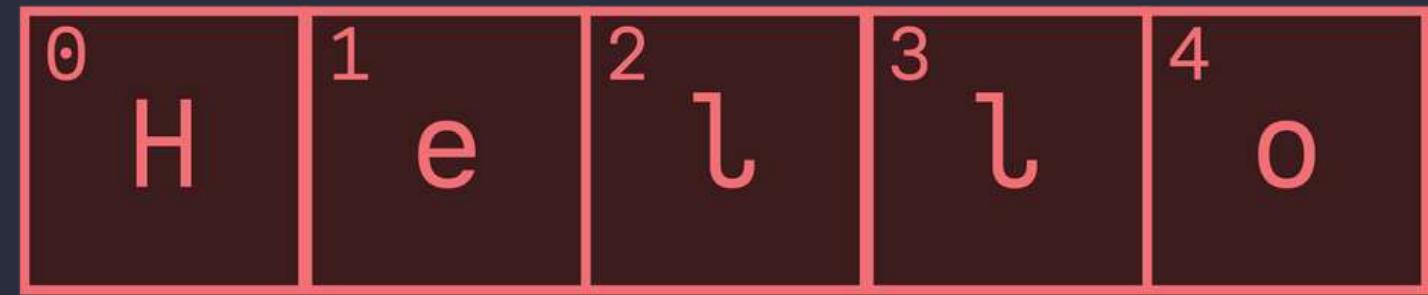
Loop Count: 2

```
word = "Hello"  
for char in word:  
    print(char)
```

Name

char

Object



Output

Loop Count: 2

```
word = "Hello"  
for char in word:  
    print(char)
```

Name

char

Object



Output

e

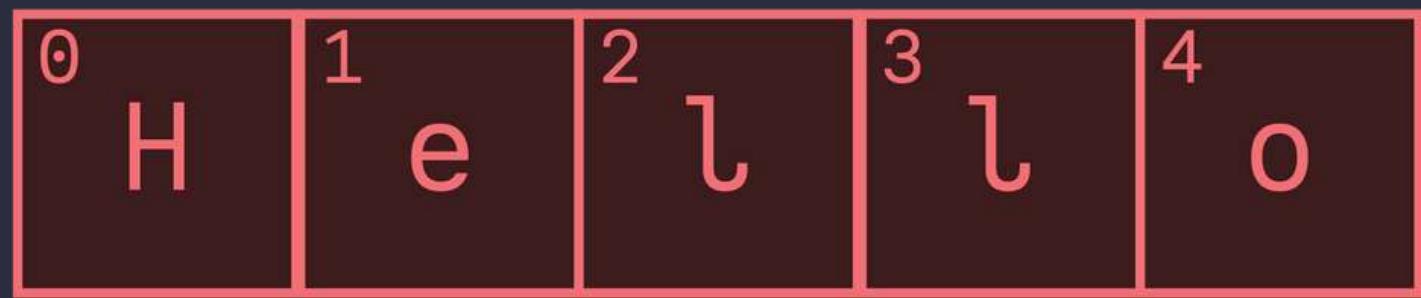
Loop Count: 3

```
word = "Hello"  
for char in word:  
    print(char)
```

Name

char

Object



Output

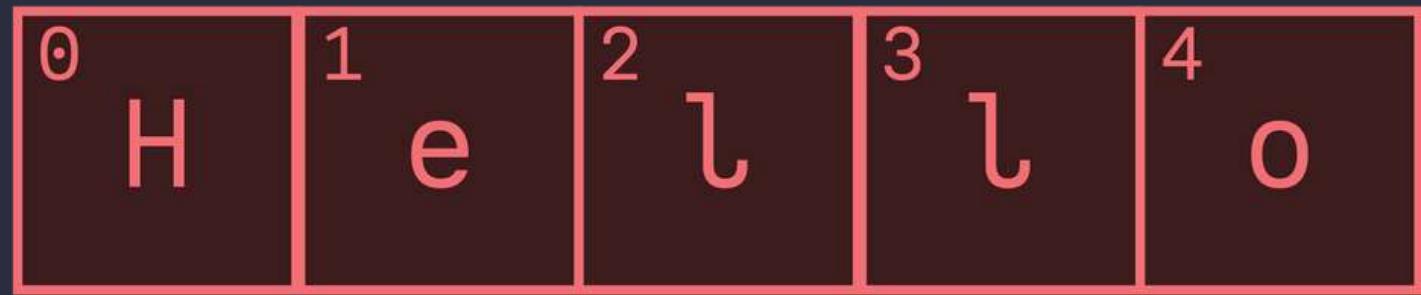
Loop Count: 3

```
word = "Hello"  
for char in word:  
    print(char)
```

Name

char

Object



Output

l

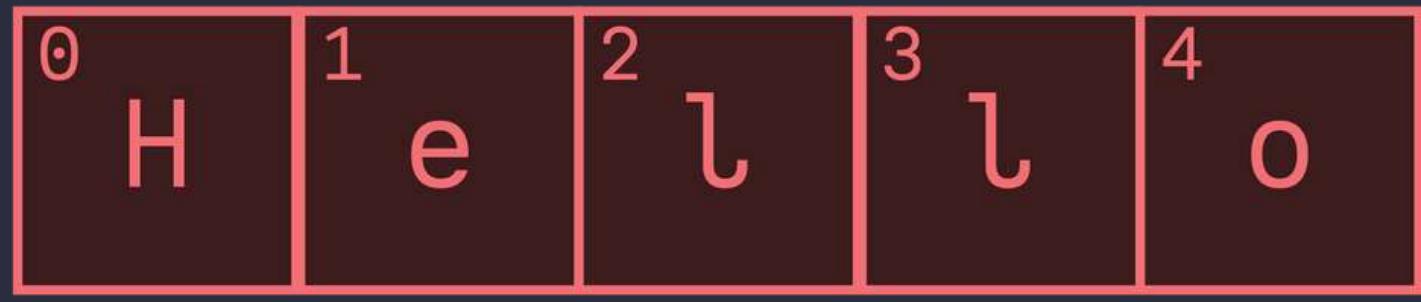
Loop Count: 4

```
word = "Hello"  
for char in word:  
    print(char)
```

Name

char

Object



Output

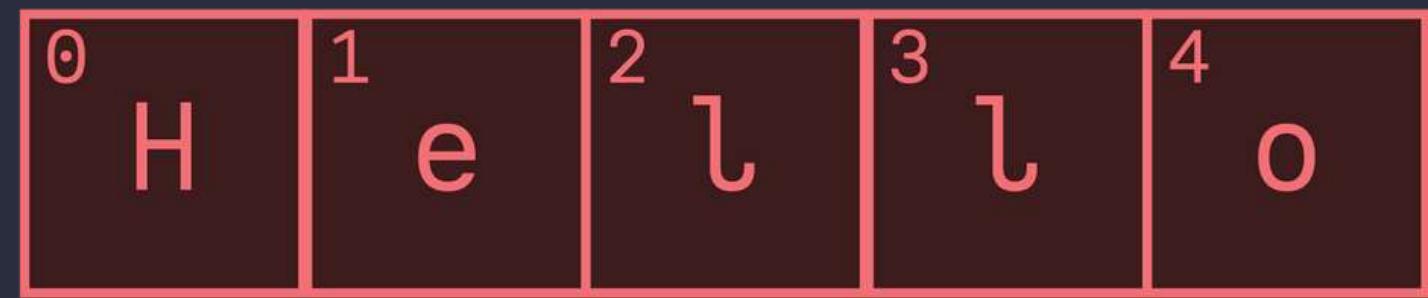
Loop Count: 4

```
word = "Hello"  
for char in word:  
    print(char)
```

Name

char

Object



Output

l

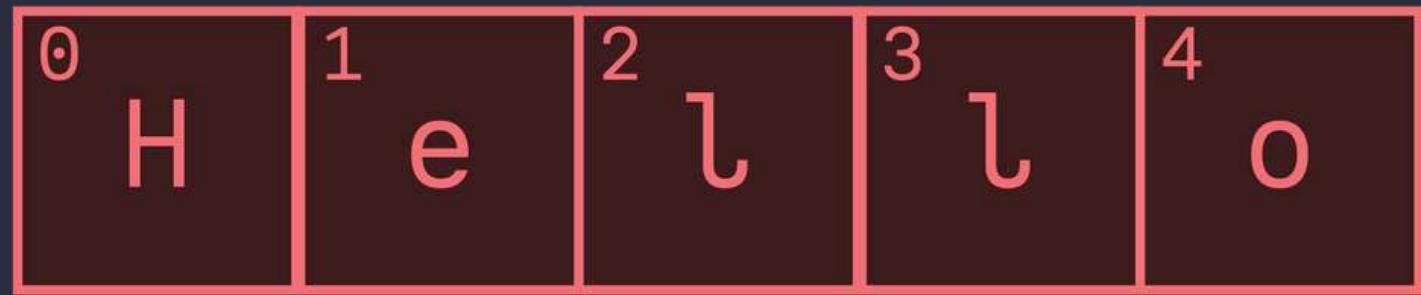
Loop Count: 5

```
word = "Hello"  
for char in word:  
    print(char)
```

Name

char

Object



Output

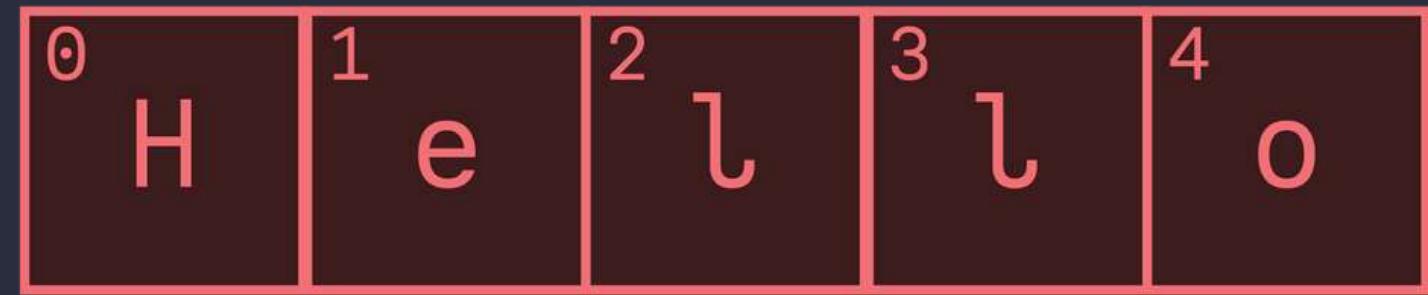
Loop Count: 5

```
word = "Hello"  
for char in word:  
    print(char)
```

Name

char

Object



Output

o

==

The diagram illustrates the parameters of the `range()` function in Python. A red arrow points from the word "Start" to the first parameter "1". A purple arrow points from the word "Stop" to the second parameter "10". A blue arrow points from the word "Step" to the third parameter "2". The code block contains three colored dots at the top left.

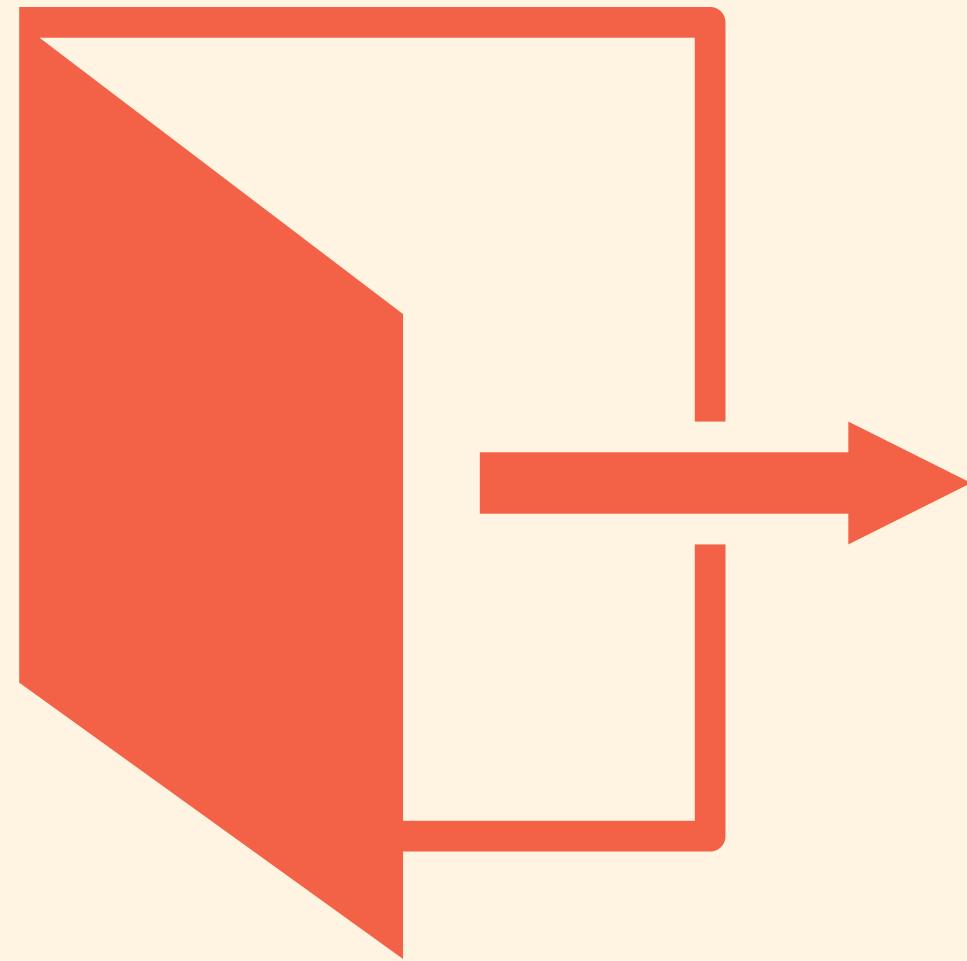
```
for num in range(1, 10, 2):
    print(num)
```





break

We can use the **break** keyword to prematurely exit a loop. Usually this is done inside of a conditional.



break

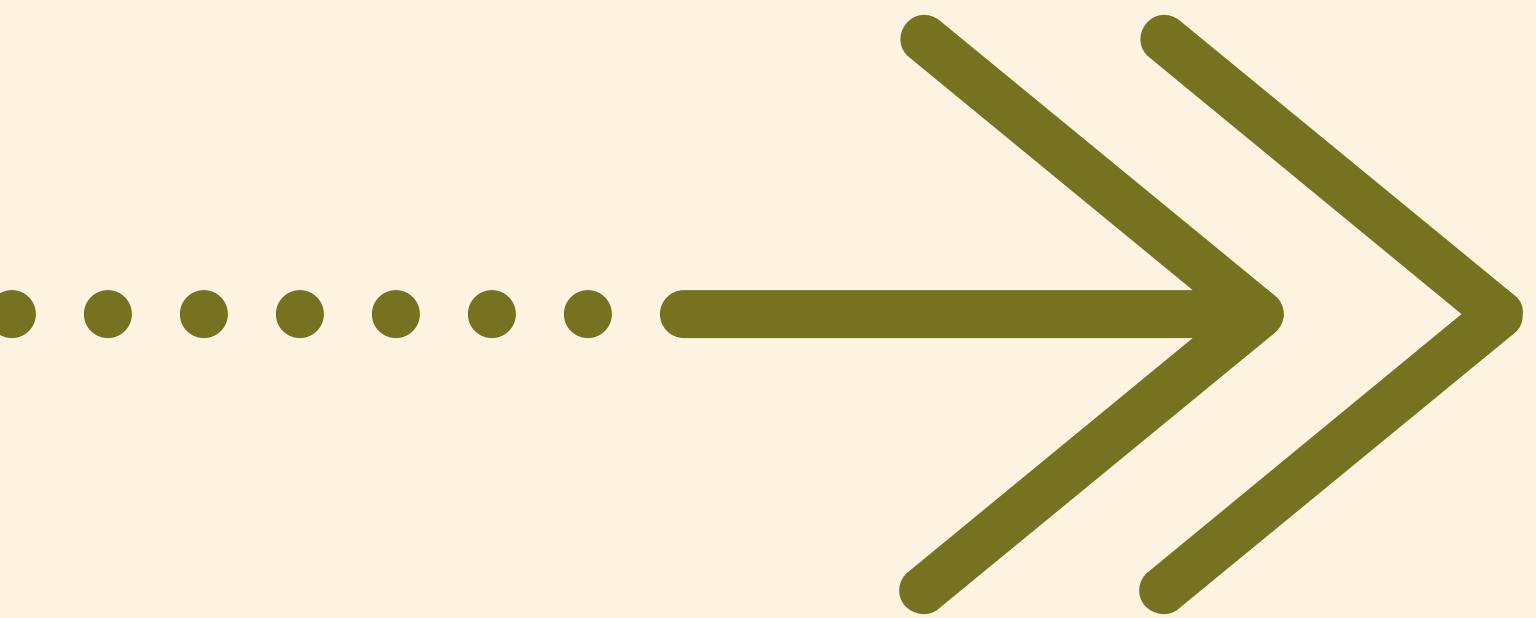
```
● ● ●  
for char in "pickleface":  
    if char == "f":  
        break  
    print(char)  
  
print("After Loop")
```

p
i
c
k
|
e
After Loop



continue

The **continue** keyword ends the current iteration of the loop, but does not break out of the loop.



continue



```
for char in "FATCAT":  
    if char == "A":  
        continue  
    print(char)  
  
print("After Loop")
```

F

T

C

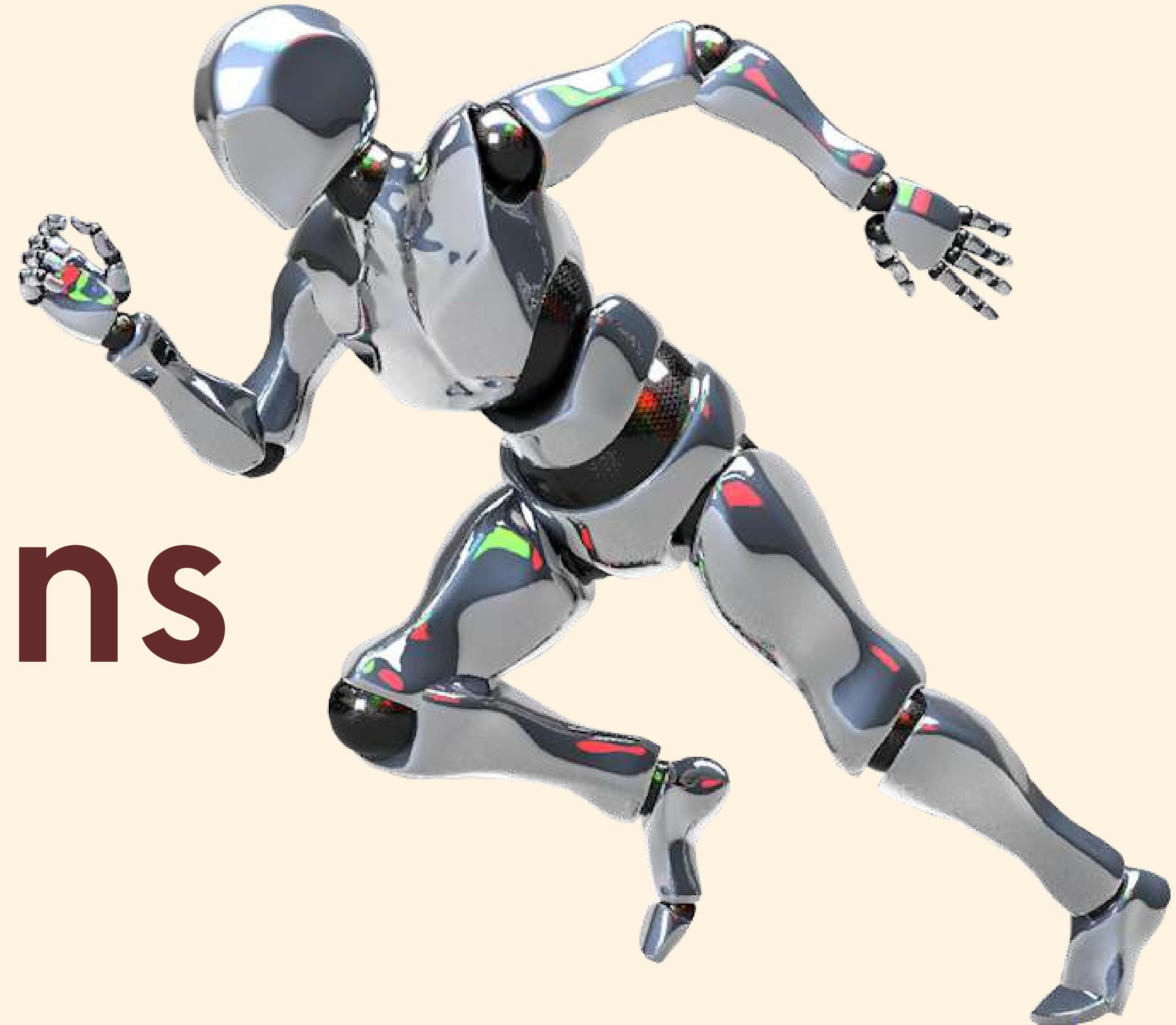
T

After Loop

Nested Loops

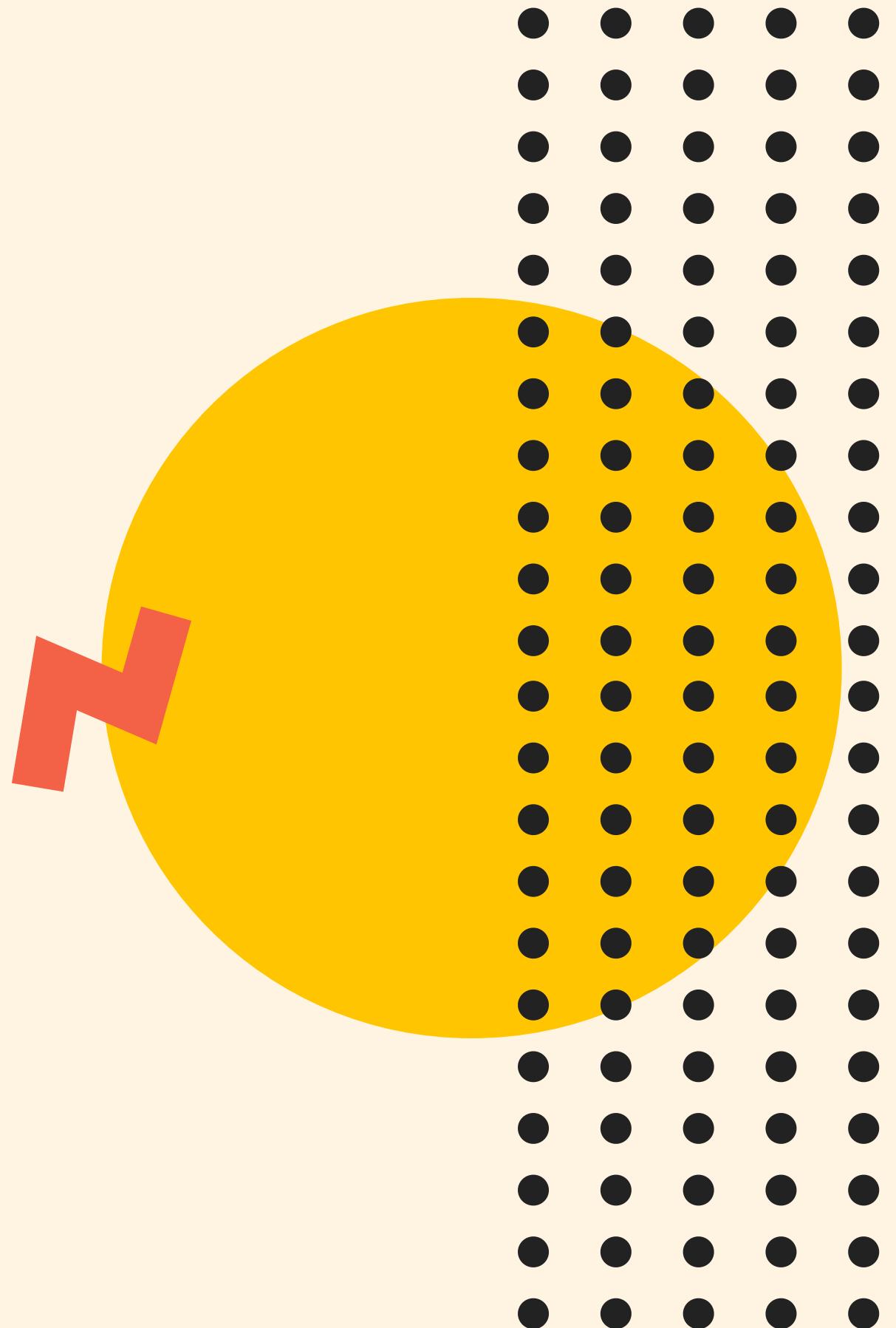
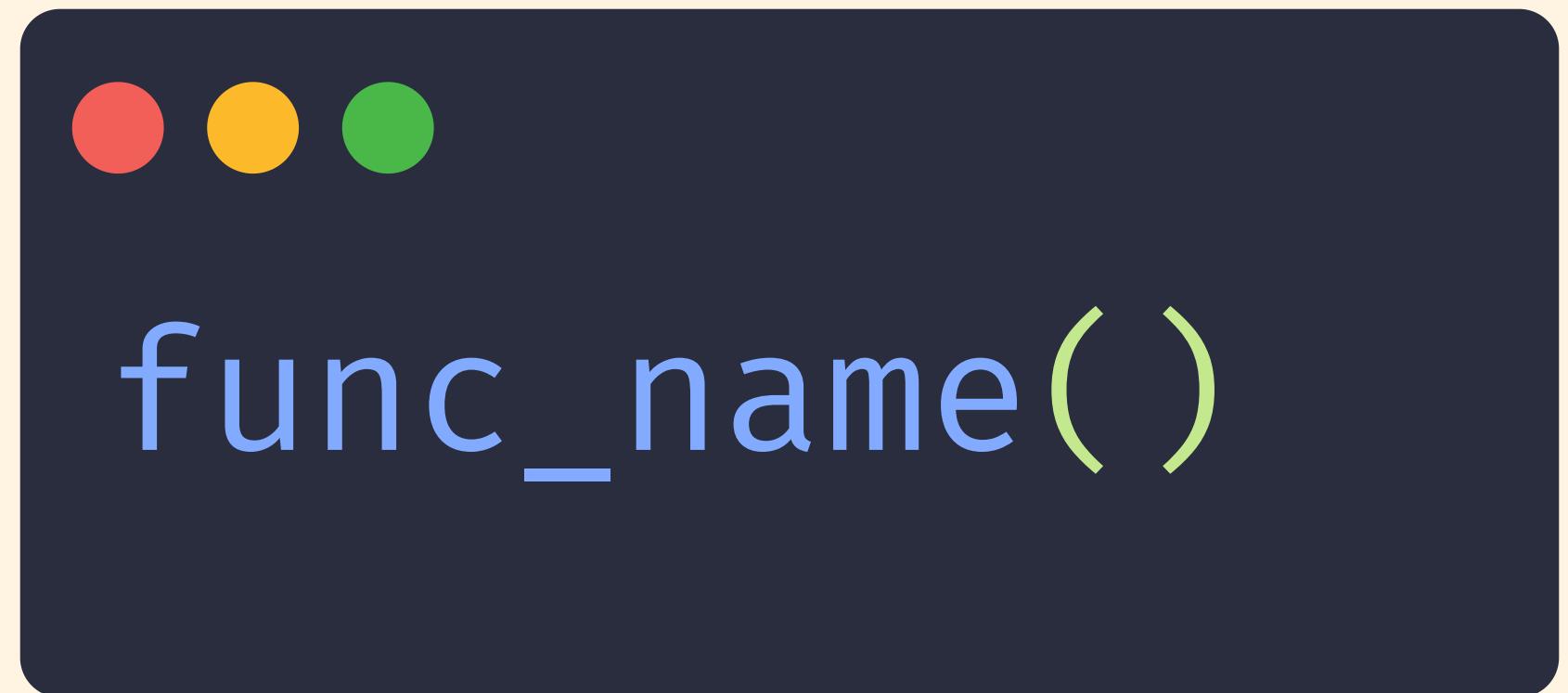


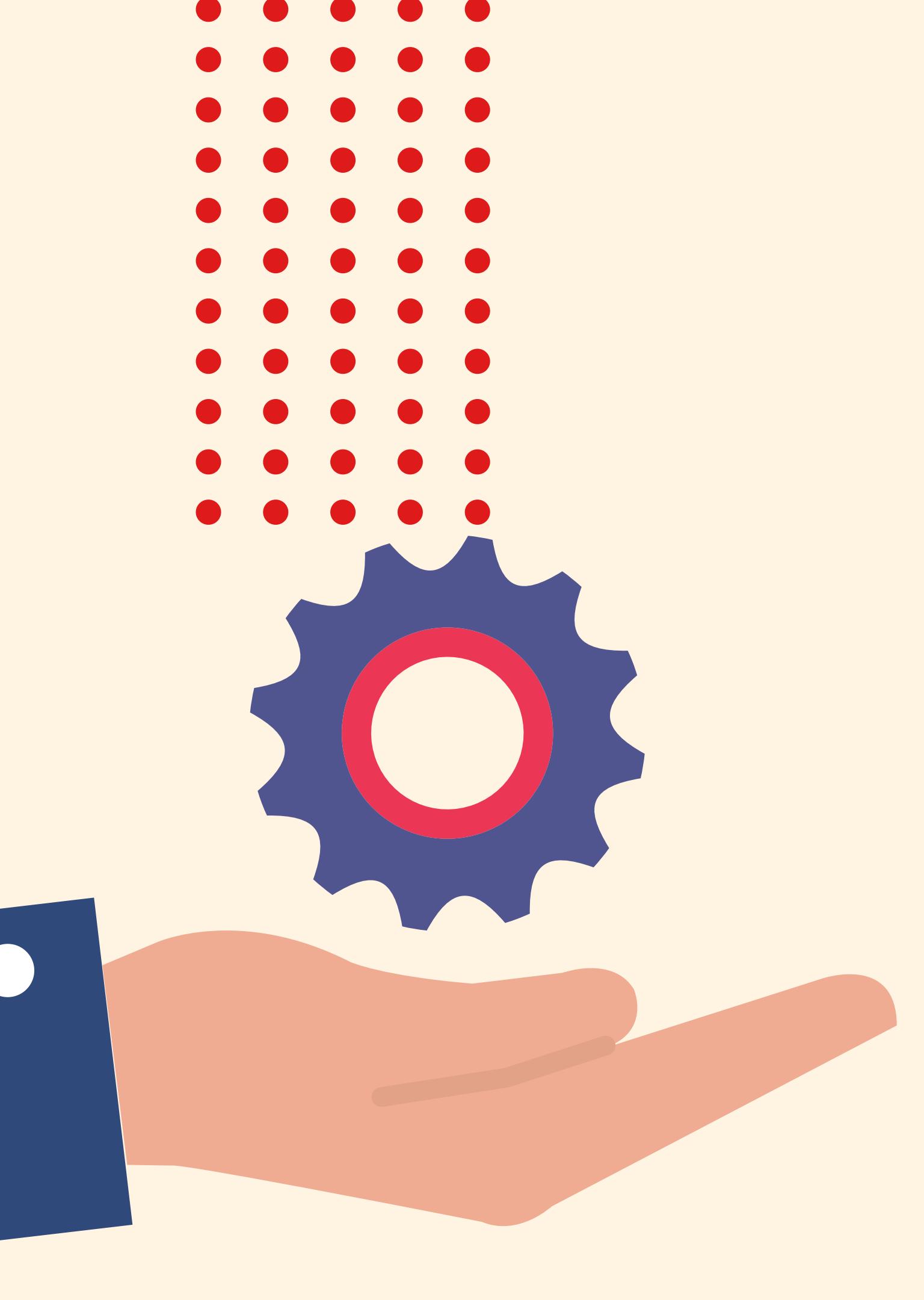
Functions



Functions

**FUNCTIONS ARE REUSABLE
ACTIONS THAT HAVE A NAME**





Functions

WHY USE THEM?

- We can use functions to prevent code duplication. Keep code DRY
- Functions help us abstract away code, breaking a complex program down into small pieces.

Define

**BEFORE WE CAN USE A
FUNCTION, WE MUST DEFINE
IT AND GIVE IT A NAME.**

Execute

**ONCE PYTHON "KNOWS"
ABOUT OUR FUNCTION, WE
CAN CALL IT ANYTIME.**

Defining A Function



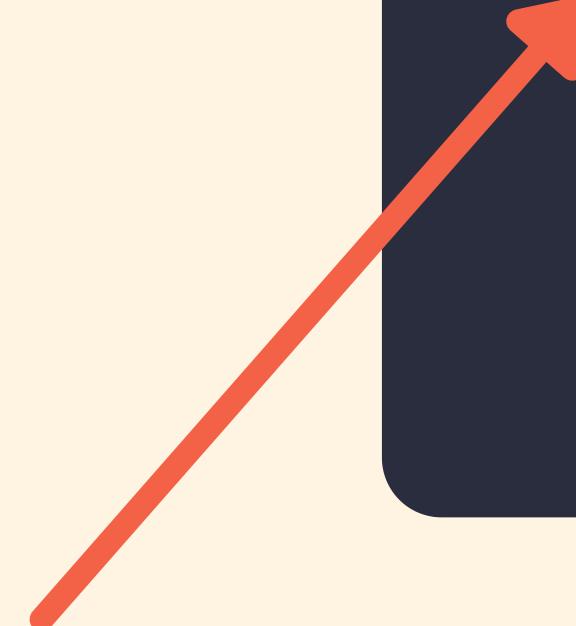
```
def laugh():
    print("HA" * 20)
```

Defining A Function



```
def laugh():
    print("HA" * 20)
```

def
keyword



Defining A Function

```
def laugh()  
    print("HA" * 20)
```

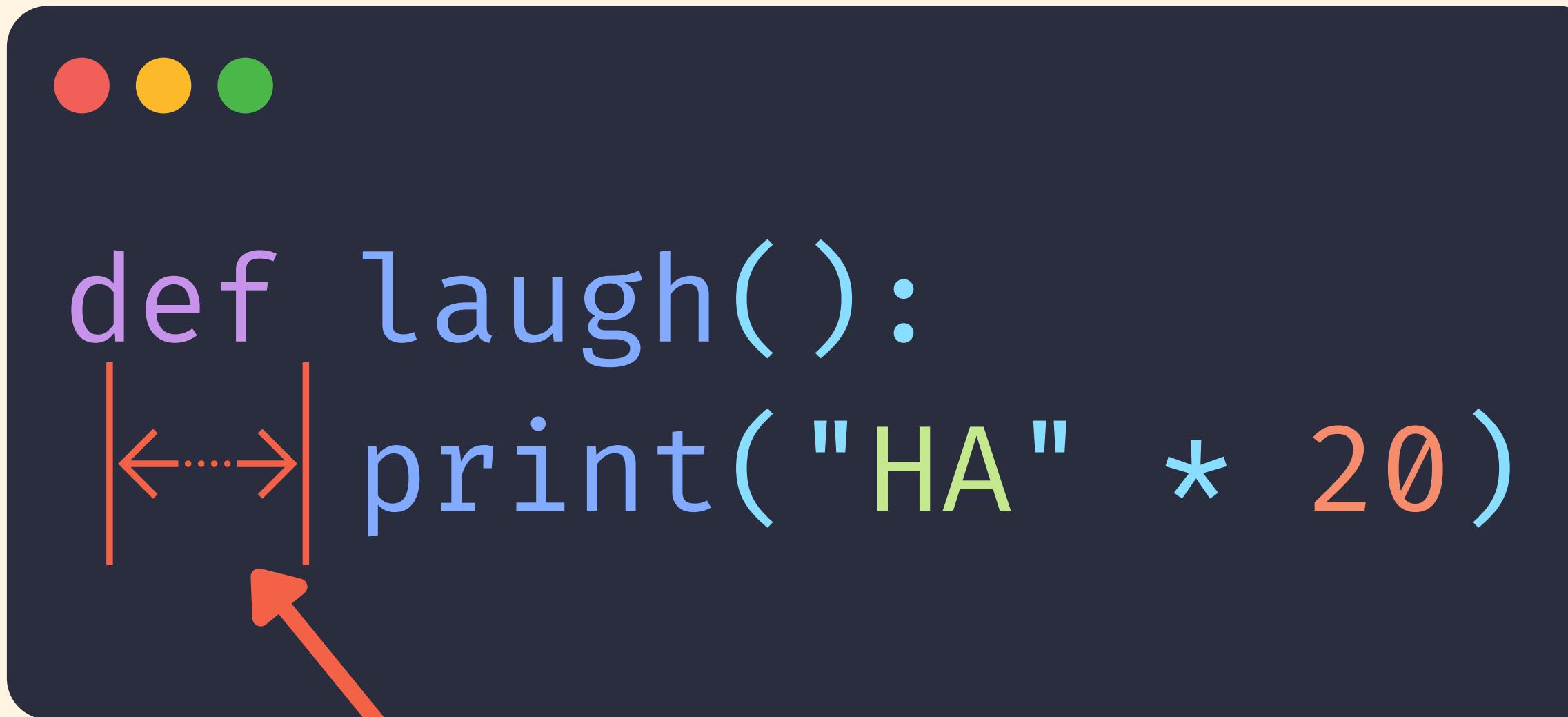
parens

Defining A Function

```
def laugh():  
    print("HA" * 20)
```

colon

Defining A Function



```
def laugh():
    print("HA" * 20)
```

indentation!

Calling A Function



laugh()

HAHAHAHAHAHAHAHAHA

HAHAHAHAHAHAHAHAHA



Function Name

Parentheses



Function Name

Argument

Parentheses

Arguments!



```
def laugh(intensity):  
    print("HA" * intensity)
```



laugh(2)

HAHA

laugh(10)

HAHAHAHAHAHAHAHAHA

Arguments!



```
def divide(x,y):  
    print(x/y)
```

Order of the arguments matters!



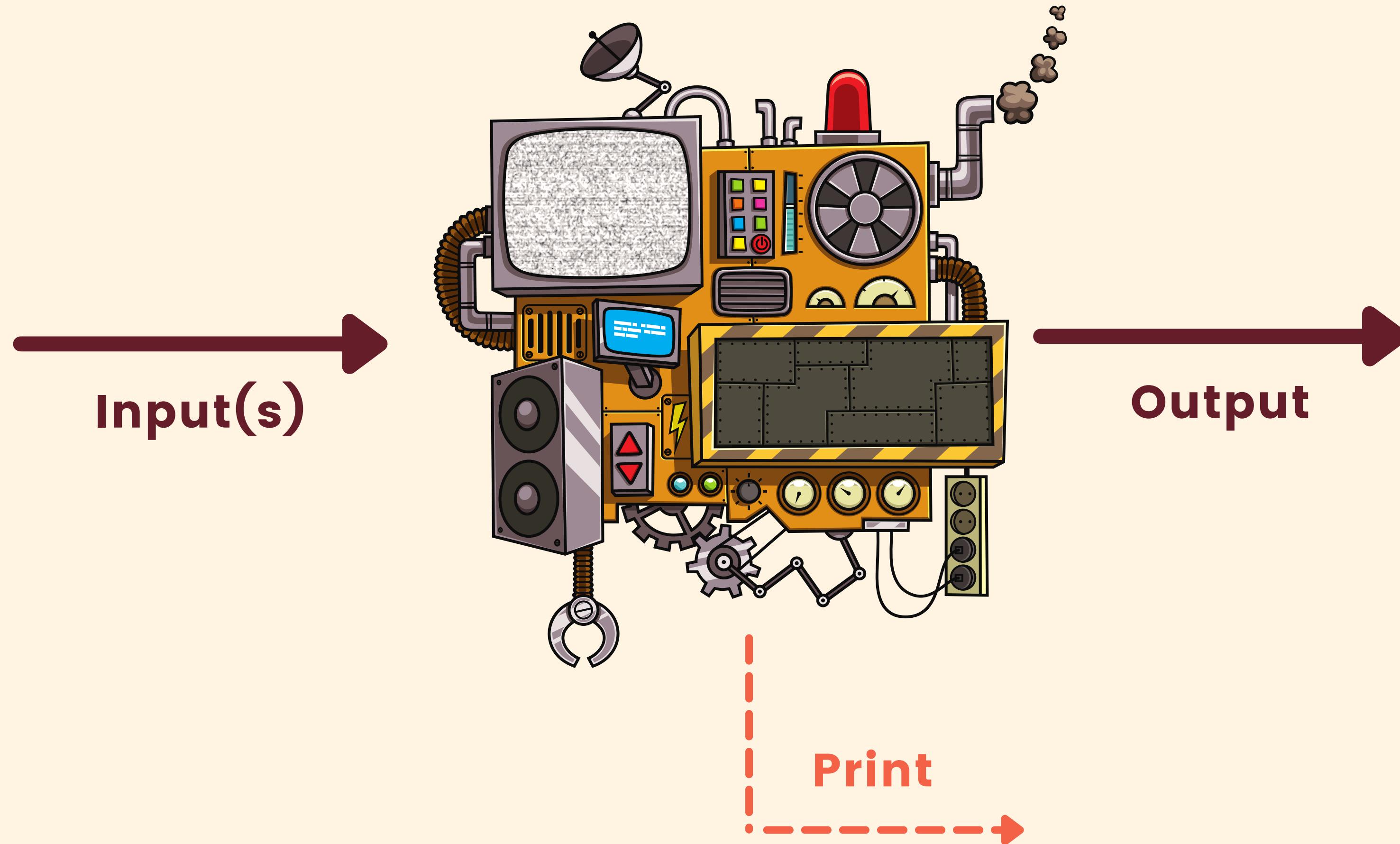
divide(12,3)

4.0

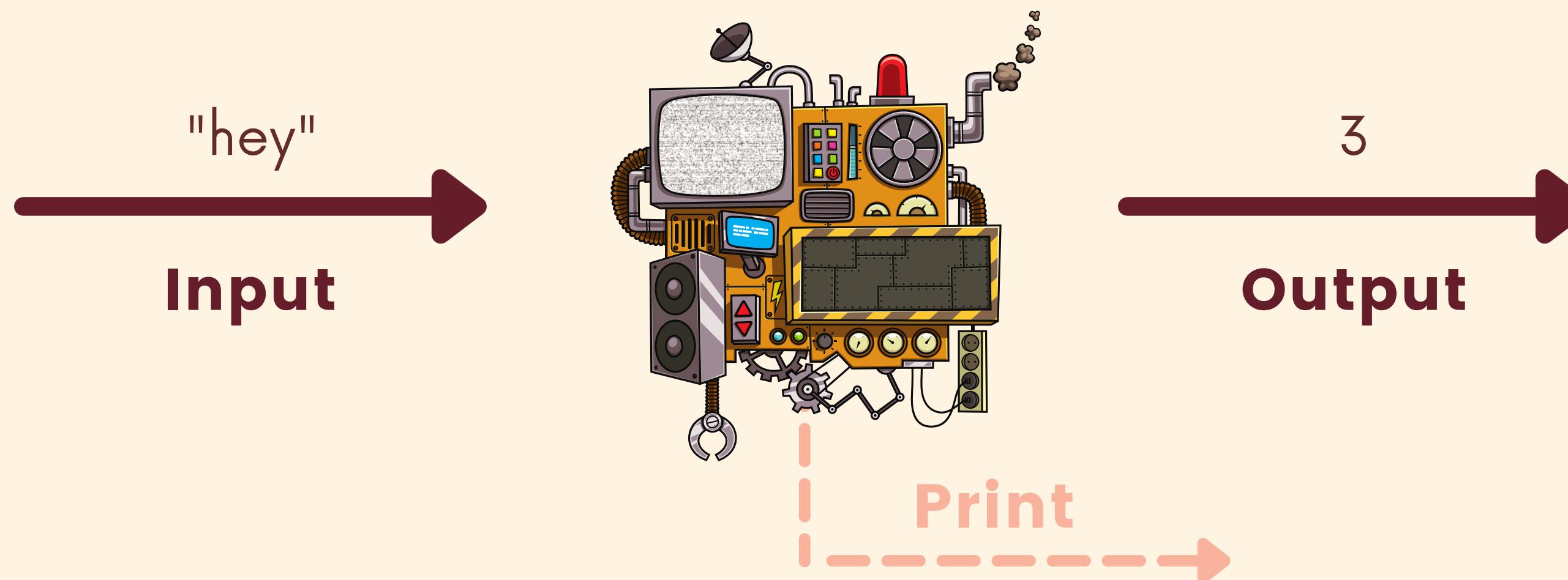


divide(3,12)

0.25



```
● ● ●  
> num = len("hey")  
> num  
3
```

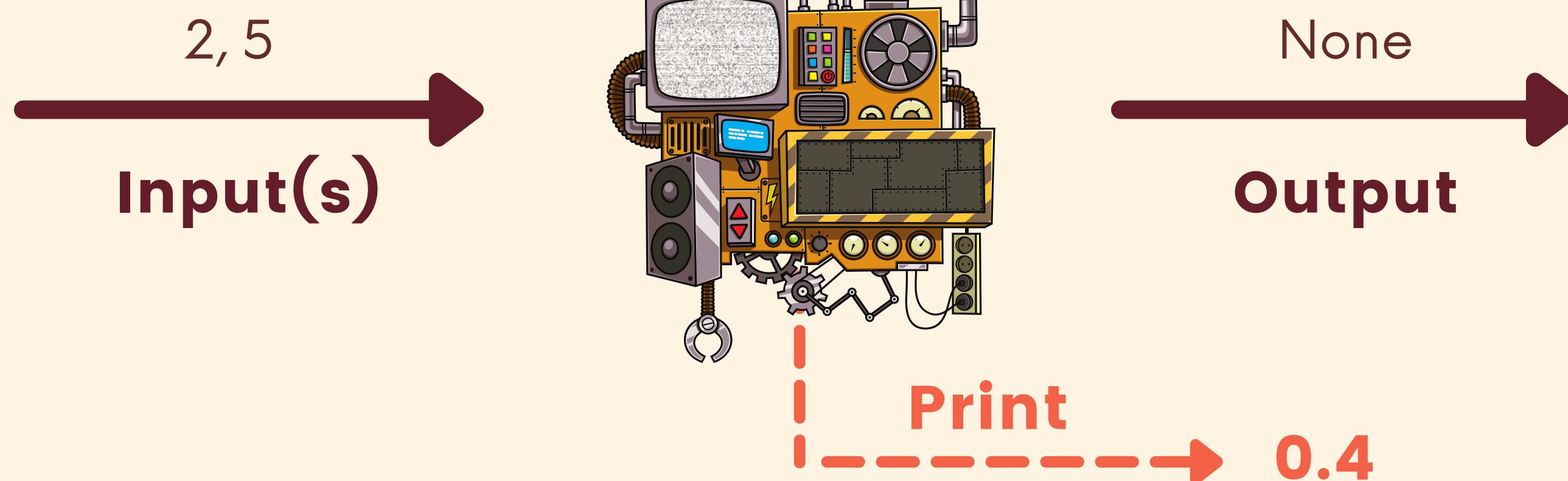


• • •

```
def divide(x,y):  
    print(x/y)
```

• • •

```
> n = divide(2,5)  
0.4  
> n  
>
```



The Return Keyword



```
def divide(x,y):  
    return x/y
```

Outputs whatever value comes after return keyword
Ends the execution of a function



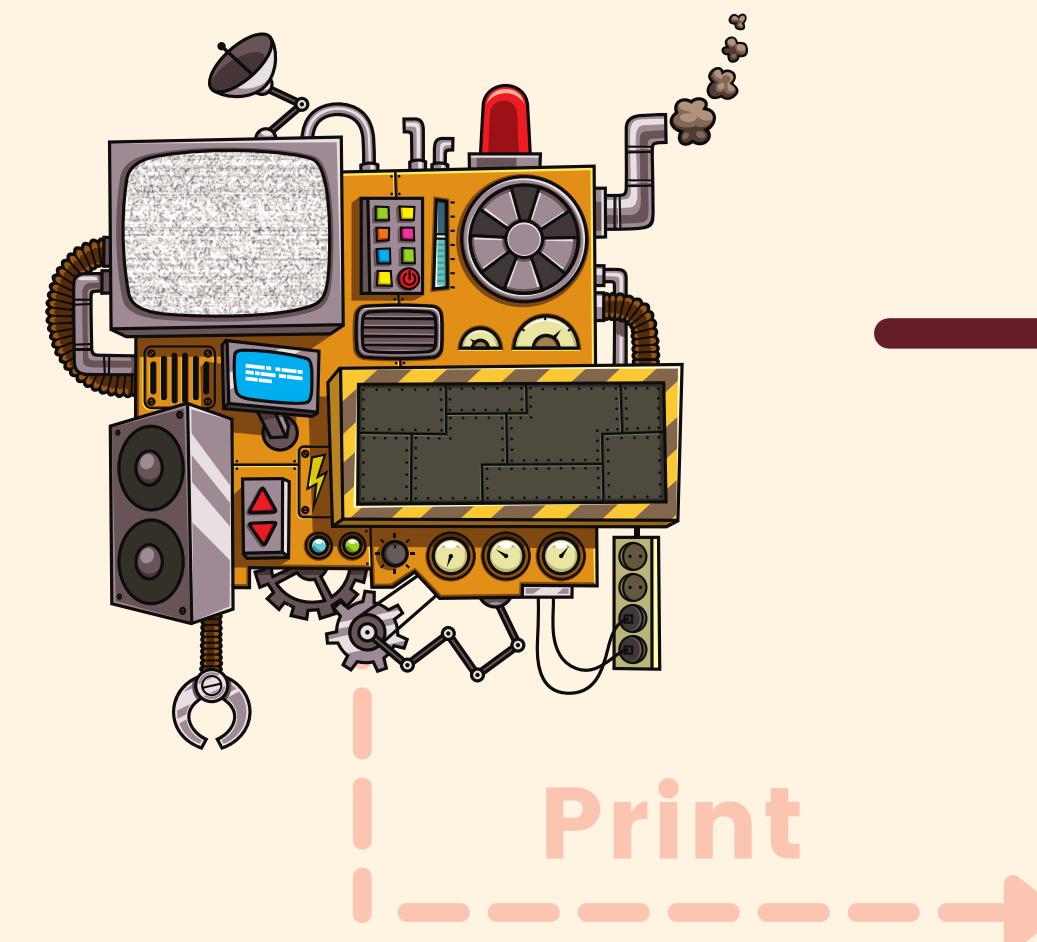
```
def divide(x,y):  
    return x/y
```



```
> n = divide(2,5)  
> n  
0.4
```

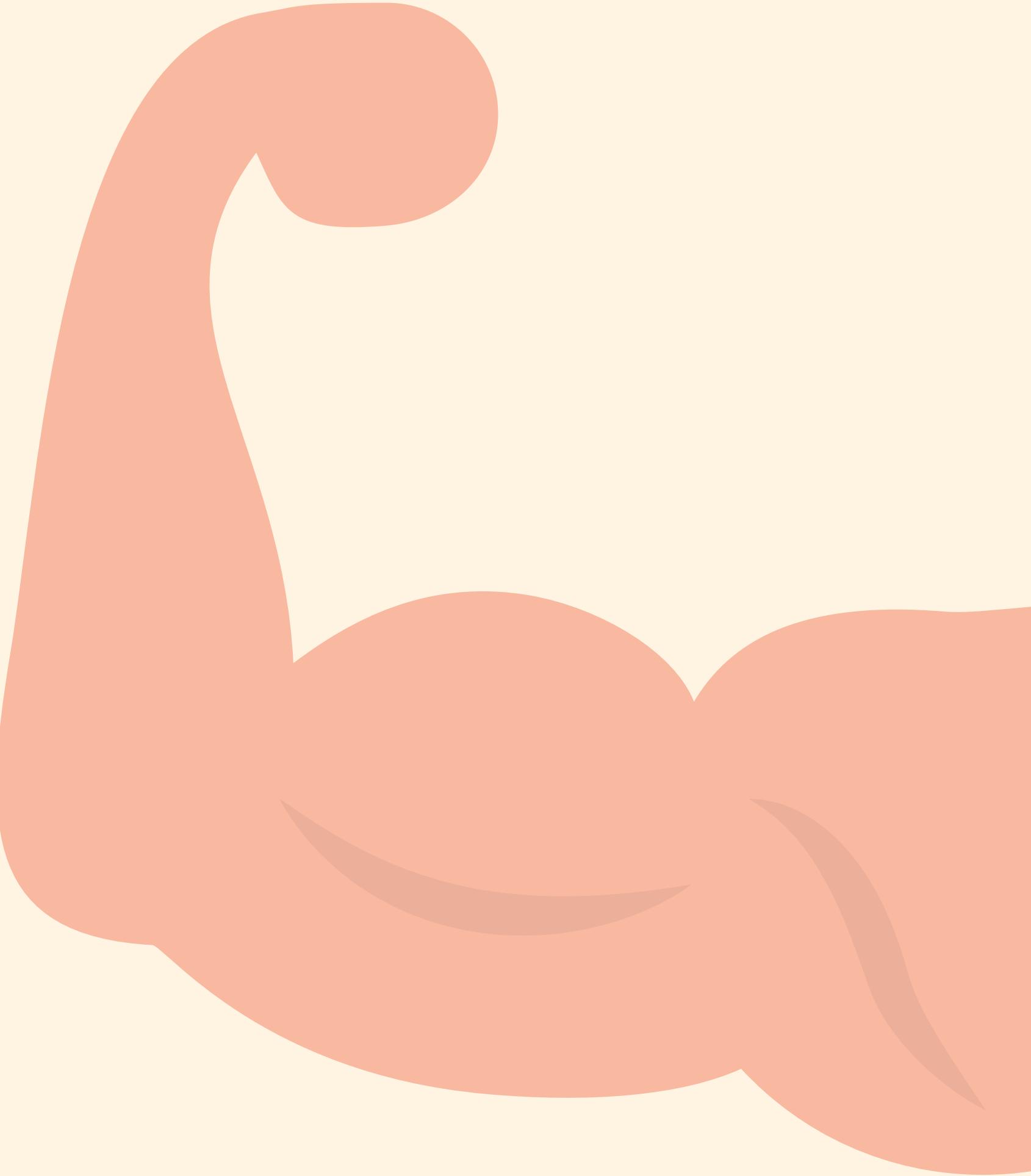
2, 5
Input(s)

0.4
Output



Print

Practice



Default Parameters



```
def laugh(intensity=10):  
    print("HA" * intensity)
```

To give a parameter a default value if no argument is provided, simply add the default using this format:

parameter=value

Scope





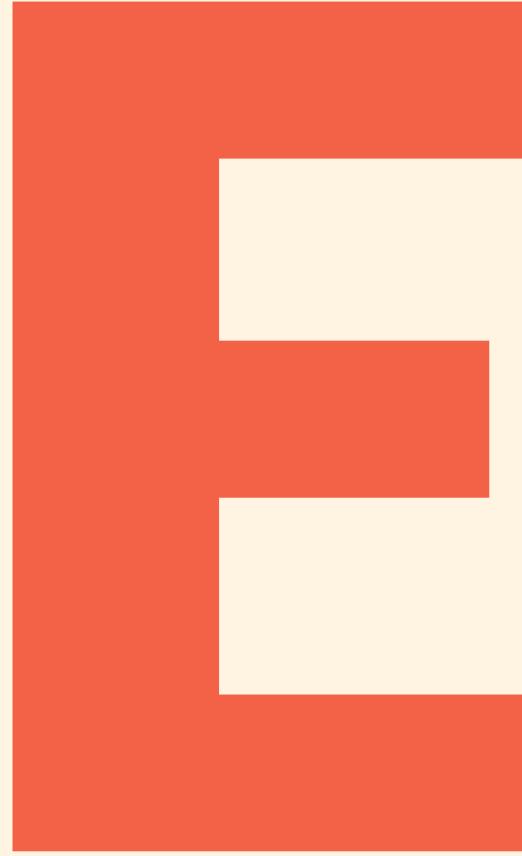
Scope

WHAT IS IT??

Every variable we work with in Python has a scope or boundary where it can be used. There are specific rules to how variables are scoped based on where they are initially defined.

A yellow L-shaped block, representing a lexical scope.

Lexical

An orange E-shaped block, representing an enclosing scope.

Enclosing

A red G-shaped block, representing a global scope.

Global

A dark red B-shaped block, representing a built-in scope.

Built-In

Global Scope

```
movie = "Amadeus"  
def review():  
    print(movie)  
def inner():  
    print(movie)
```

Variables declared outside of functions are in the global scope. All functions have access to them.

Local Scope

```
def cube(num):  
    answer = num ** 3  
    print(answer)
```

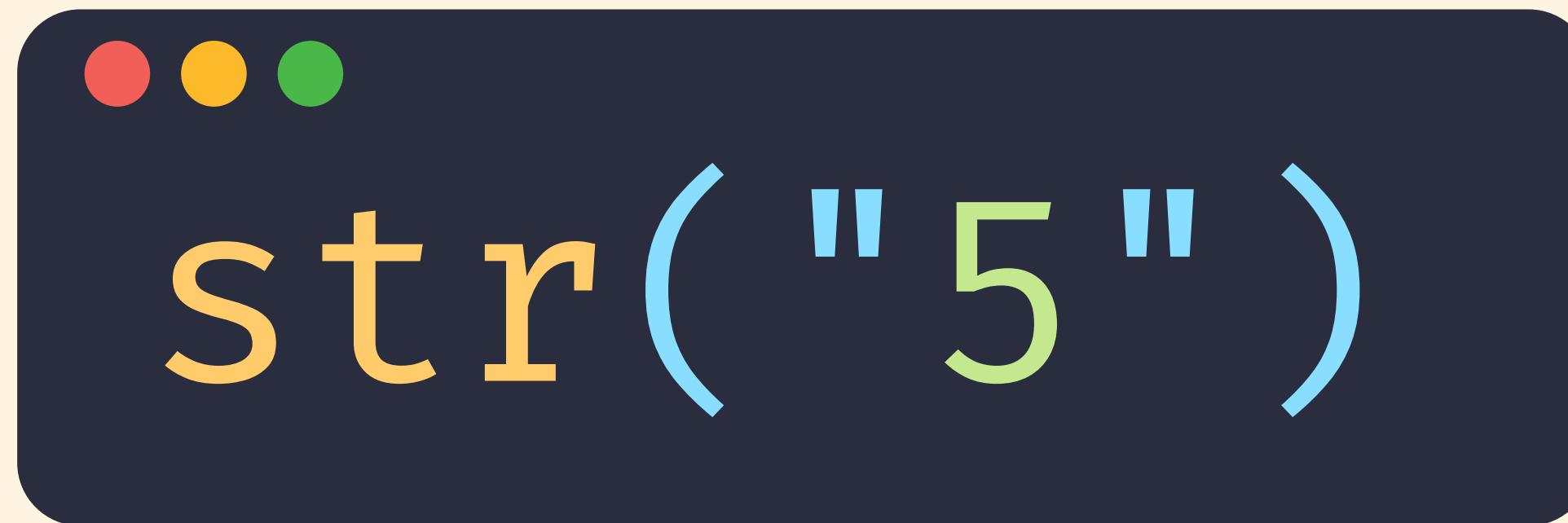
Variables defined in a function are **scoped to that function**.
They are not available outside that function!

Enclosing Scope

```
...  
def outside():  
    a = 10  
    def inside():  
        print("a is: ", a)  
    inside()
```

Nested "inner" functions have access to variables declared in outer parent functions

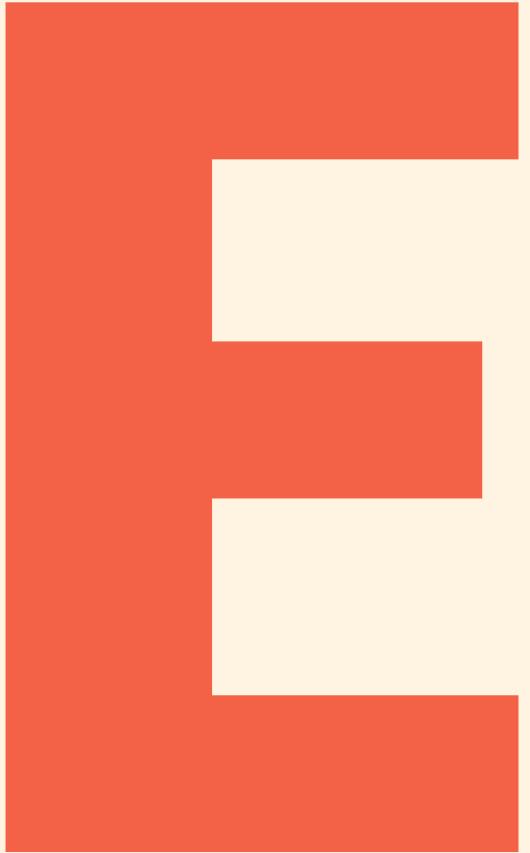
Built-In Scope



All the built-in objects in Python are in the Built-In Scope. We have access to them anywhere!

A large yellow L-shaped block, representing the Lexical scope.

Lexical

A large orange F-shaped block, representing the Enclosing scope.

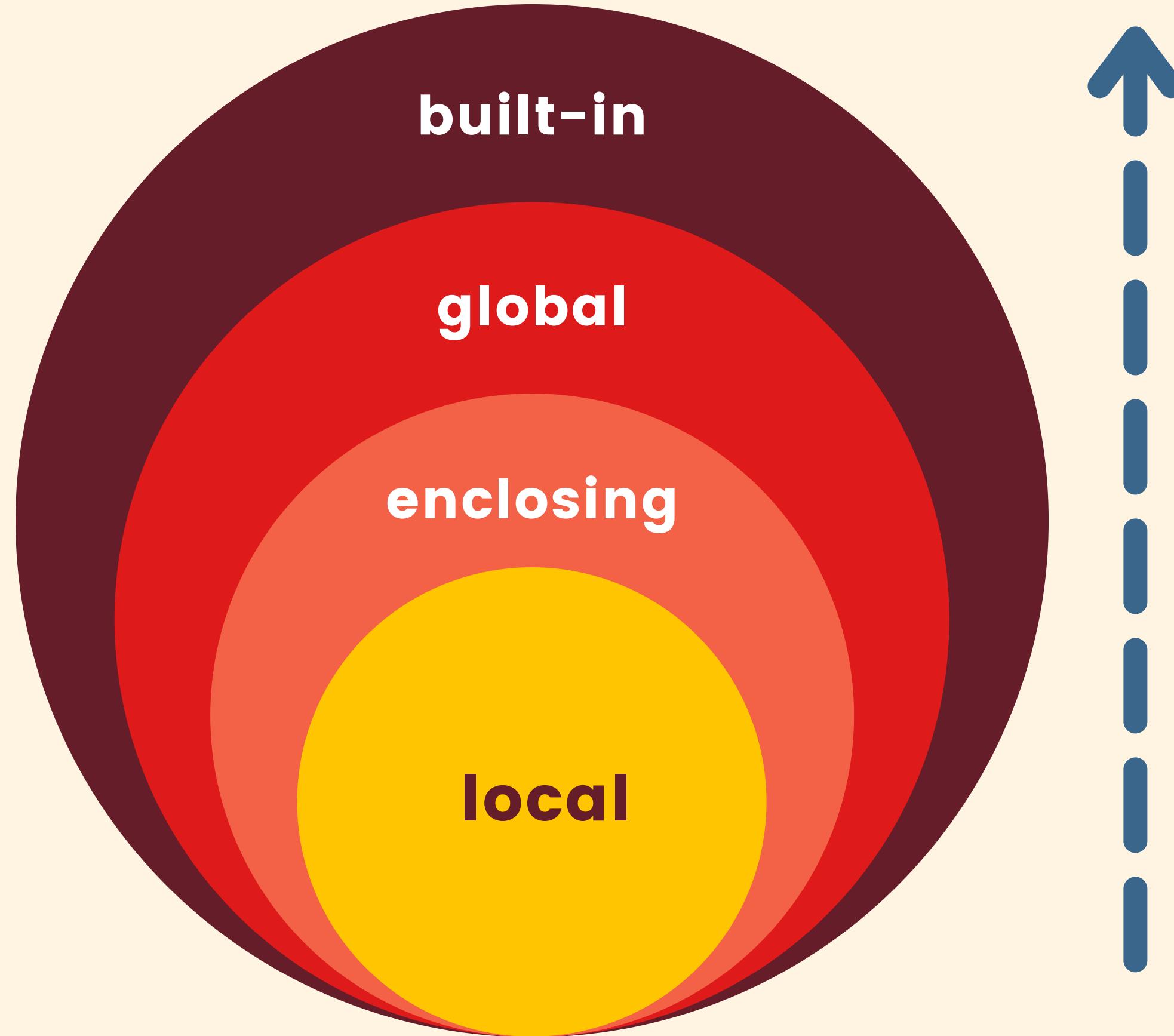
Enclosing

A large red G-shaped block, representing the Global scope.

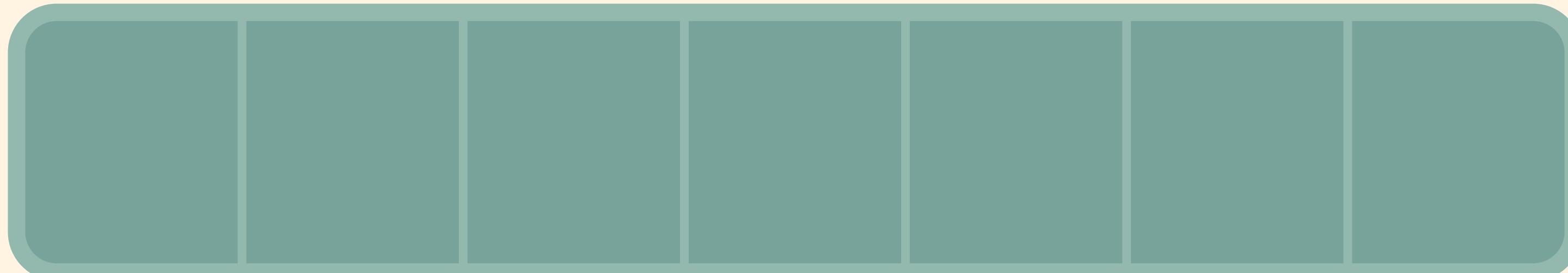
Global

A large dark red B-shaped block, representing the Built-In scope.

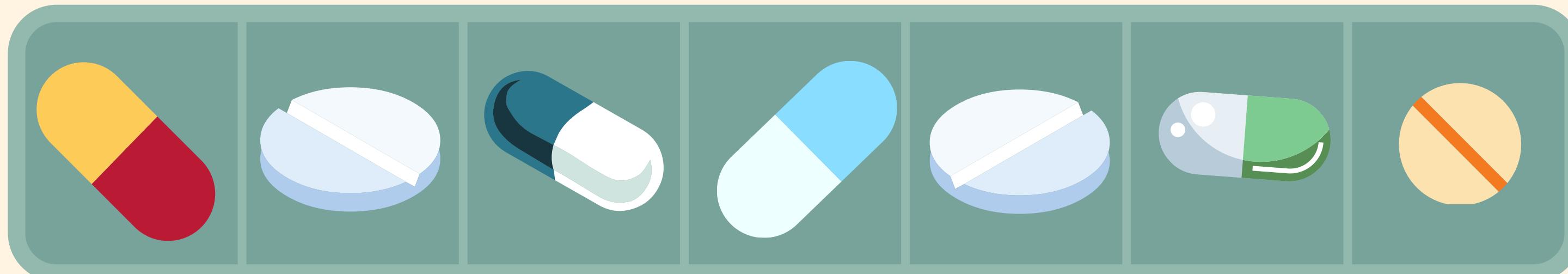
Built-In



lists



lists



Lists

Lists are **ordered collections** of data.

They can hold any of the data types we've seen.

27

'hi'

False

8

0.6

'cat'

-3



Creating Lists



```
tasks = ["Trash", "Dishes", "Laundry", "Dinner"]
```





Creating Lists

```
...  
tasks = ["Trash", "Dishes", "Laundry", "Dinner"]
```

Items





Creating Lists

```
tasks = ["Trash", "Dishes", "Laundry", "Dinner"]
```

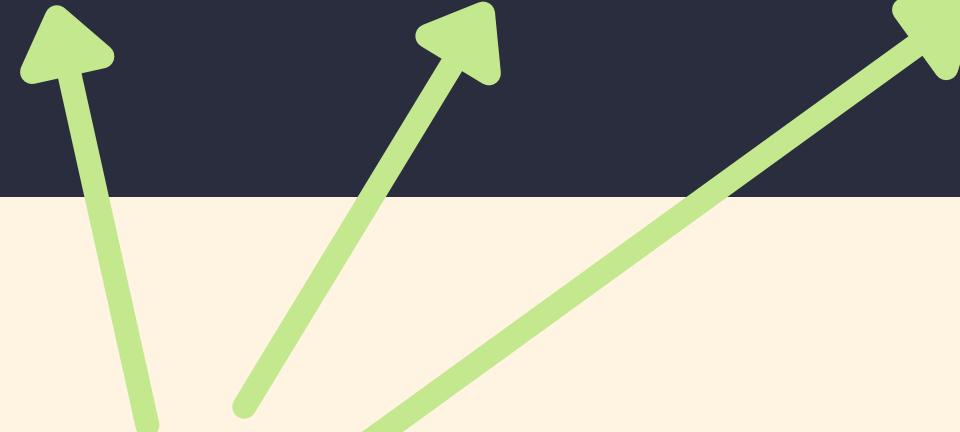
Brackets





Creating Lists

```
...  
tasks = ["Trash", "Dishes", "Laundry", "Dinner"]
```



Commas



Script

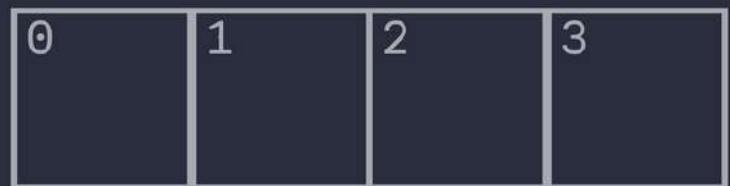
```
some_list = [1, "Hello", False, 3.5]
```

Name

some_list



Object



1

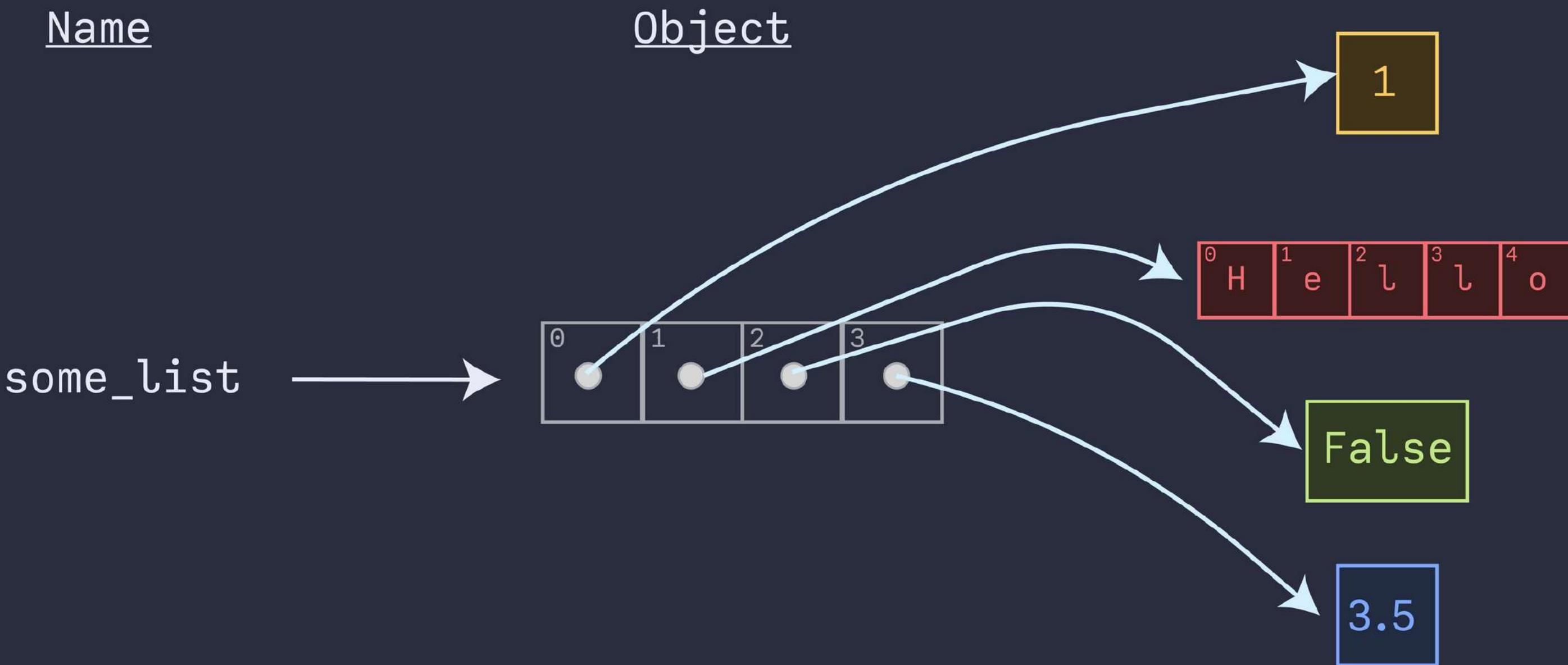
0 H | 1 e | 2 l | 3 l | 4 o

False

3.5

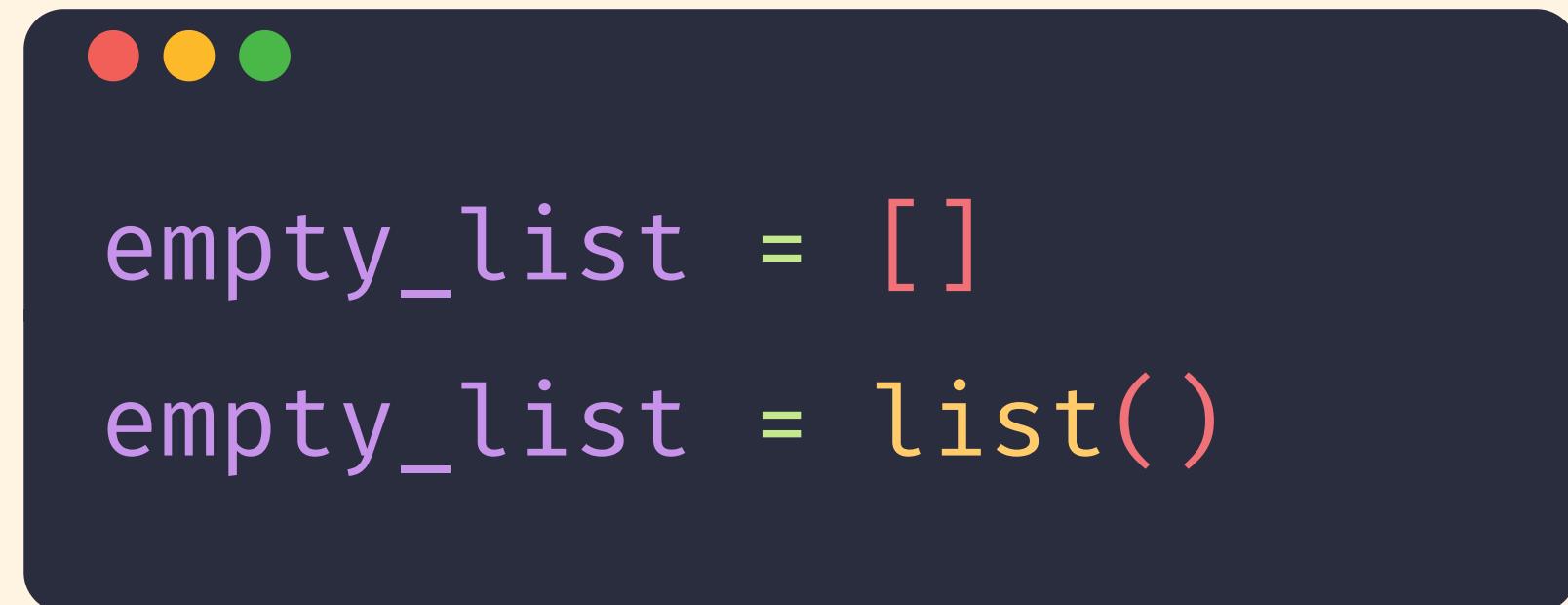
Script

```
some_list = [1, "Hello", False, 3.5]
```





Empty List



```
empty_list = []
empty_list = list()
```



```
list[index]
```

We can retrieve individual elements from a list by passing an index number inside of square brackets. Like strings, indices start at 0.

27	'hi'	False	8	0.6	'cat'	-3
----	------	-------	---	-----	-------	----

0

1

2

3

4

5

6

list[index]

```
● ● ●  
> langs = ["Python", "C", "JavaScript"]  
  
> langs[1]  
C  
  
> langs[0]  
Python
```

Updating

```
● ● ●  
> nums = [7,3,9]  
> nums[1] = 8  
> nums  
[7, 8, 9]
```

Update a specific element using its index

[start:stop:step]

```
● ● ●  
letters = ['a', 'b', 'c', 'd', 'e']  
>>> letters[1:3]  
['b', 'c']  
  
>>> letters[0:5:2]  
['a', 'c', 'e']
```



Nested Lists

```
● ● ●  
> nums = [1, 2, 3, 4, [5,6]]  
> nums[4]  
[5,6]  
> nums[4][1]  
6
```





Looping Lists

```
● ● ●  
> langs = ["Python", "C", "JavaScript", "C"]  
  
> for lang in langs:  
    print(lang)  
  
Python  
C  
JavaScript  
C
```





index()

- Returns the index number for the first instance of the value you are passing
- Will give an error if the value is not in the list



```
> langs = ["Python", "C", "JavaScript", "C"]  
> langs.index("C")  
1
```





append

```
● ● ●  
> nums = [1, 2, 3, 4]  
  
> nums.append(5)  
  
[1, 2, 3, 4, 5]
```

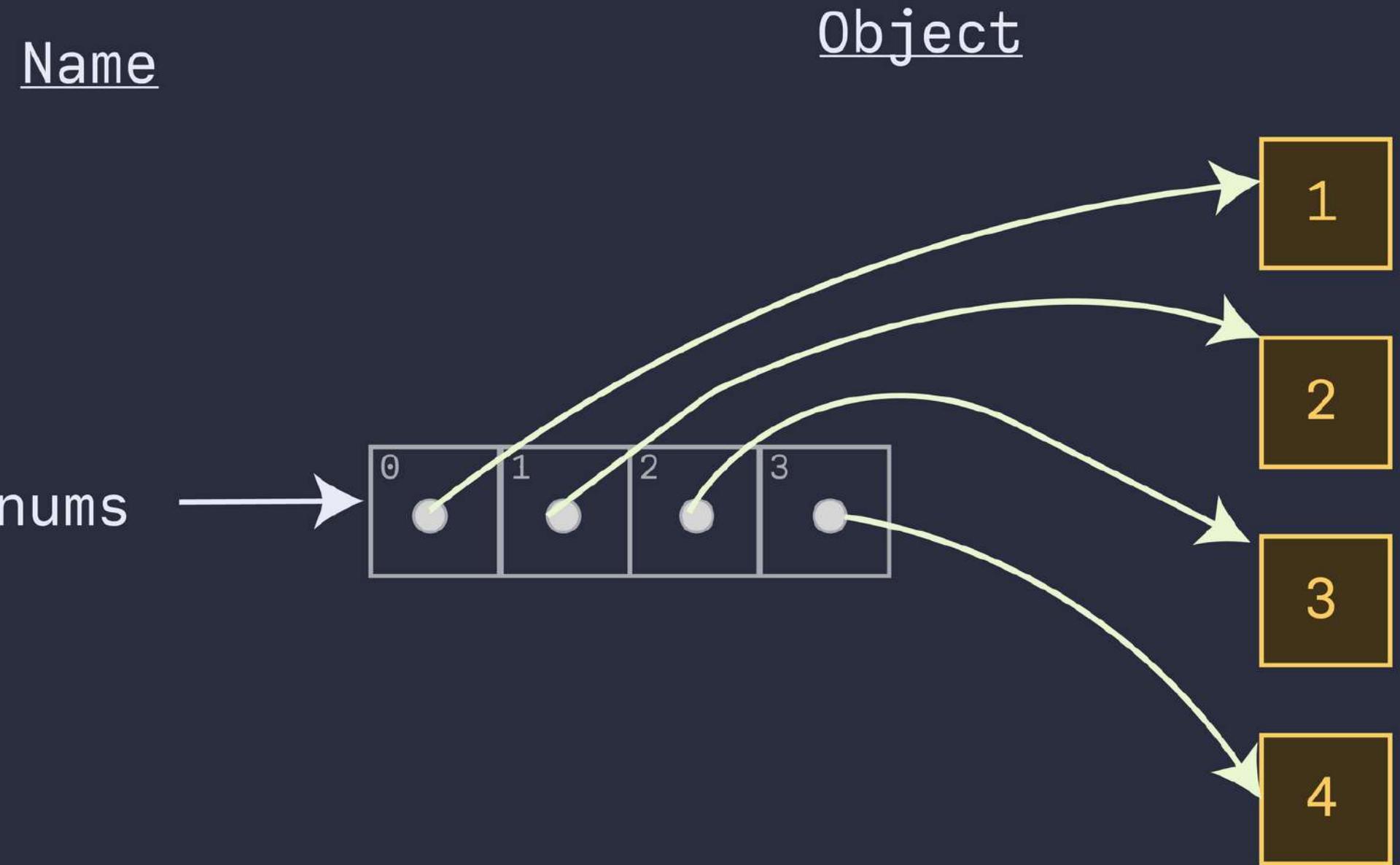
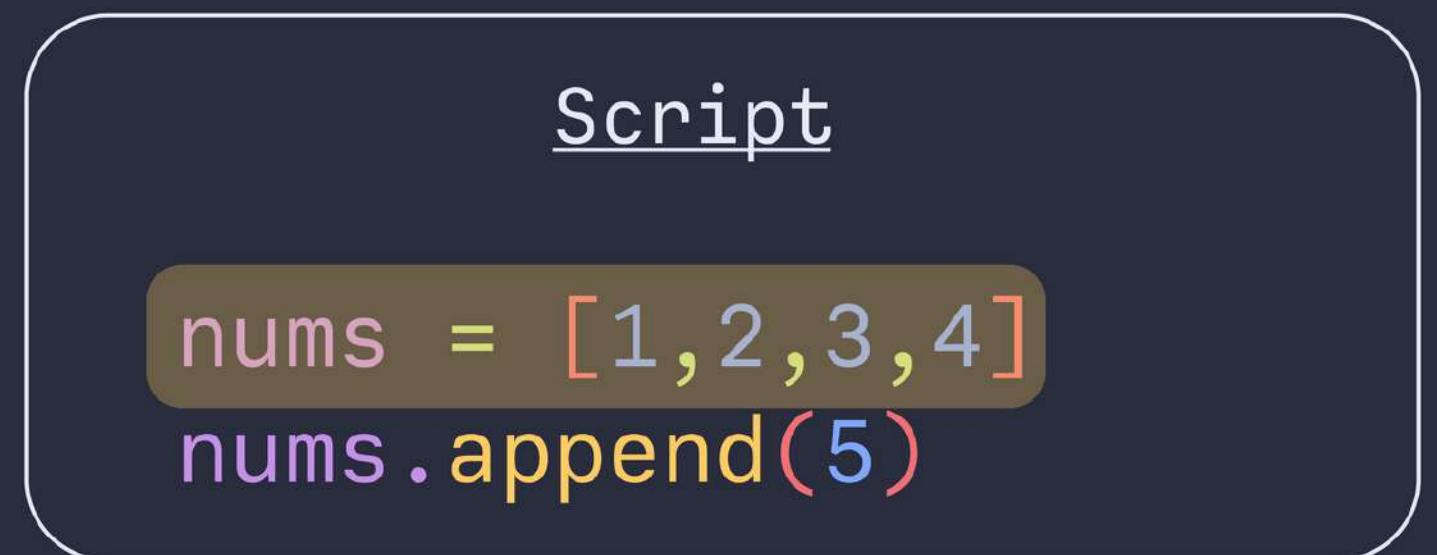
Adds a single value to the end of a list



append

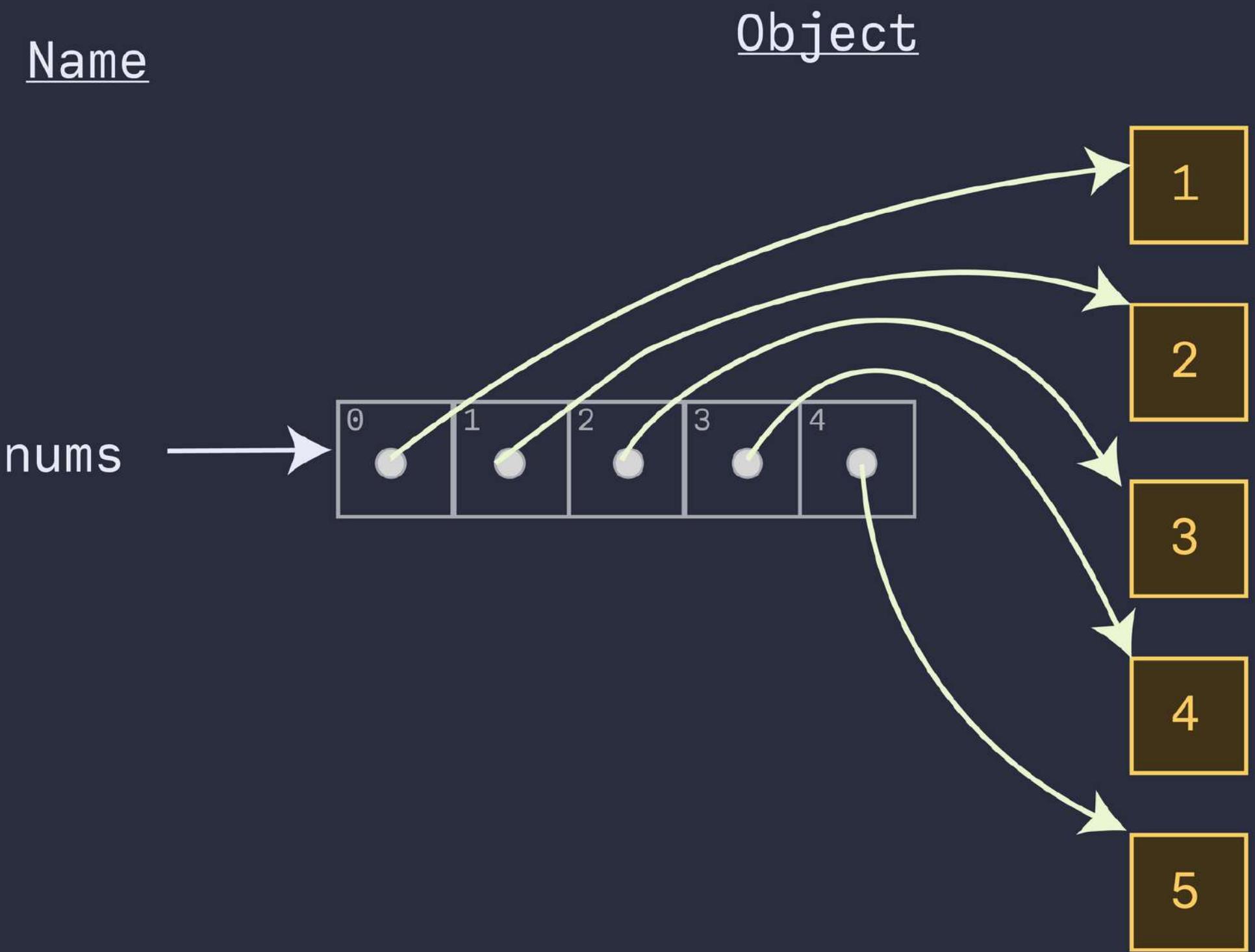
```
● ● ●  
> nums = [1, 2, 3, 4]  
> nums.append(5)  
> nums  
[1, 2, 3, 4, 5]
```

Adds a single value **to the end** of a list



Script

```
nums = [1, 2, 3, 4]
nums.append(5)
```



extend()

```
●●●  
> nums = [1,2,3]  
> nums.extend("abc")  
> nums  
[1,2,3,'a','b','c']
```

Accepts an iterable and appends each item from that iterable to the end of the list

element to be
inserted into the list

insert(index, element)

Index before
which to insert
the element



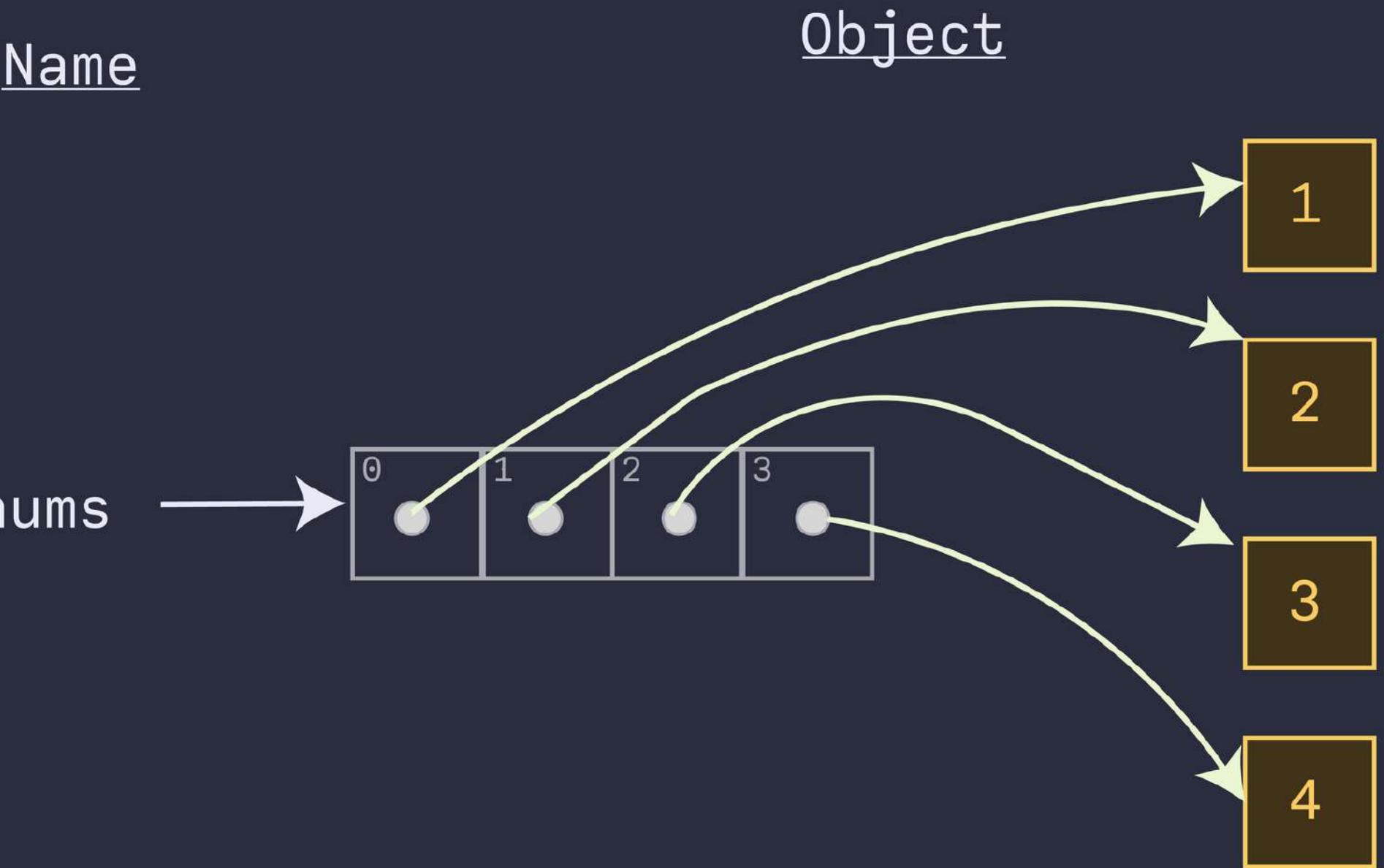
insert()

```
● ● ●  
> nums = [7,3,9]  
> nums.insert(1,"hi")  
> nums  
[7, 'hi', 3, 9]
```

Insert "hi" before the element currently located at index 1

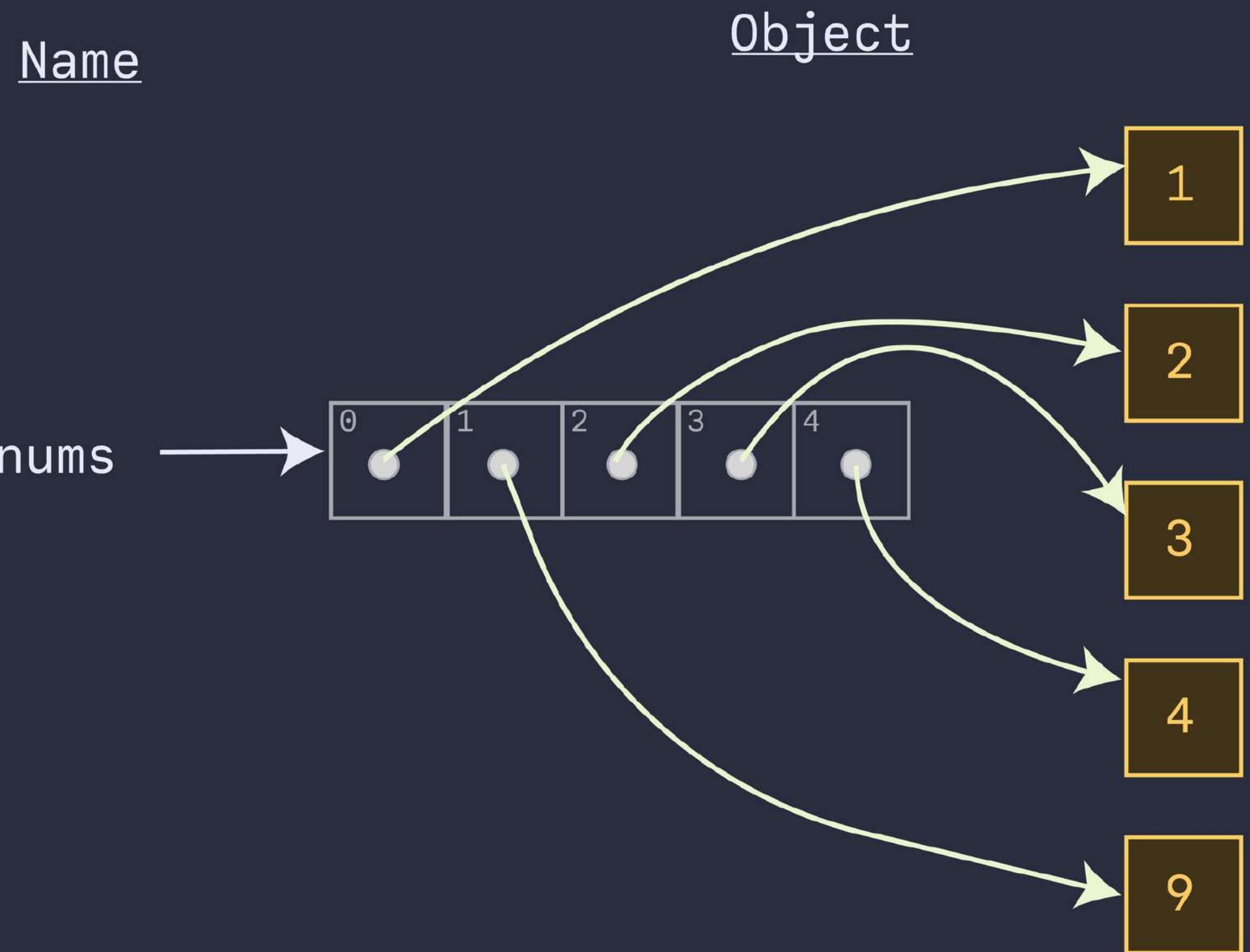
Script

```
nums = [1, 2, 3, 4]
nums.insert(1, 9)
```



Script

```
nums = [1,2,3,4]
nums.insert(1,9)
```

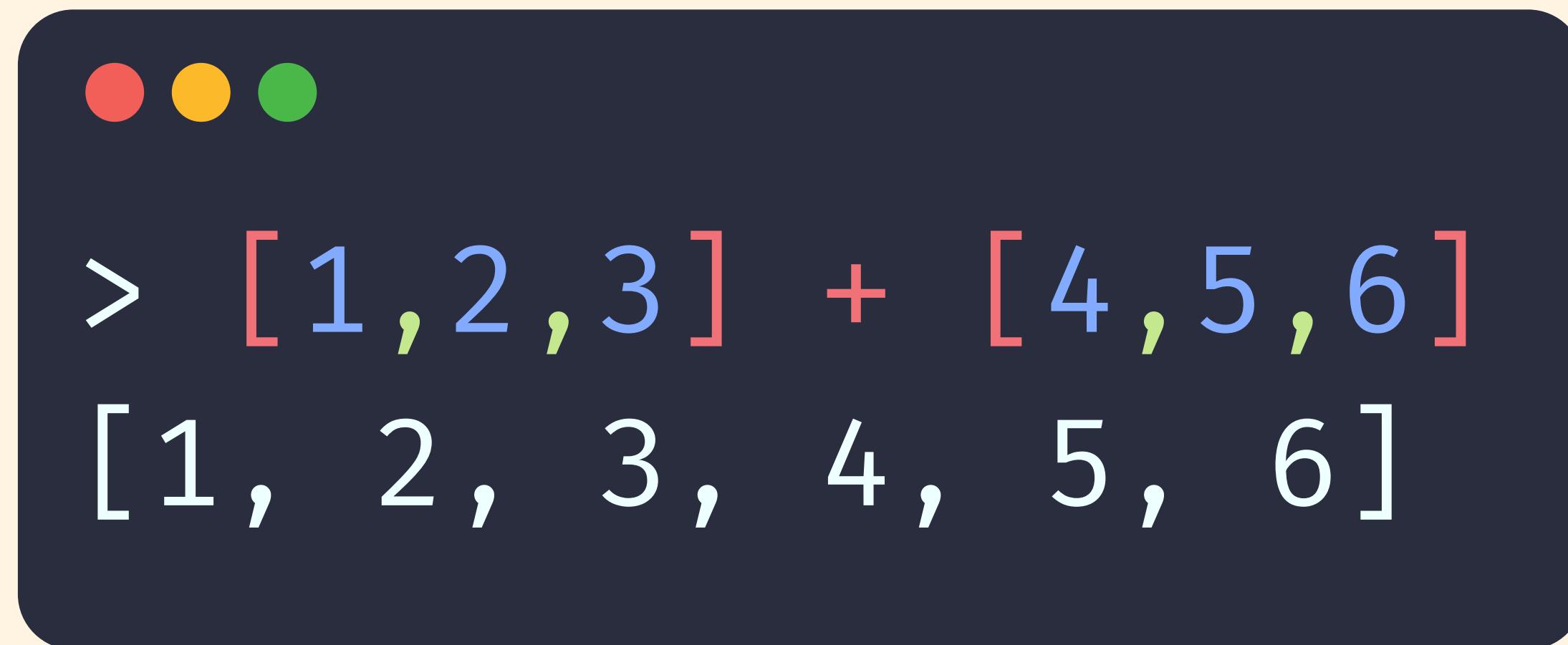


list[start:stop:step]

Slices

```
● ● ●  
> stuff = ['c', 6, 'a', 9, 't', 6]  
> stuff[0:2]  
['c', 6]  
> stuff[0:5:2]  
['c', 'a', 't']
```

Addition



Multiplication

```
...  
> [1,2,3] * 2  
[1, 2, 3, 1, 2, 3]
```

Unpacking



```
> user = ["Joe", "Bucky", 42]
> first, last, age = user
> first
"Joe"
> last
"Bucky"
> age
42
```

We can "unpack" values from a list into specific variables.

In this example, the first value in the list is stored in a variable called `first`.

The second value is stored in a variable called `last`.

The third value is stored in a variable called `age`

*Unpacking



```
> item = [4, "Pizza", "Plain", 16.98]
> quantity, *others, price = item
> quantity
4
> others
["Pizza", Plain]
> price
16.98
```

Use an asterisk (*) to gather any remaining unassigned values into a variable.



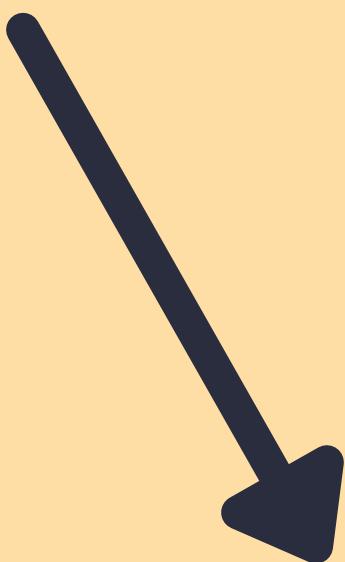
```
> list1 = [12, 9, 3, 7]
```

12	9	3	7
----	---	---	---



```
> list1 = [12, 9, 3, 7]
```

list1

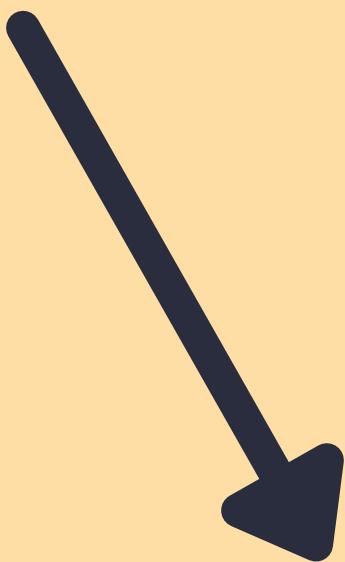


12	9	3	7
----	---	---	---



```
> list1 = [12, 9, 3, 7]  
> list2 = list1
```

list1

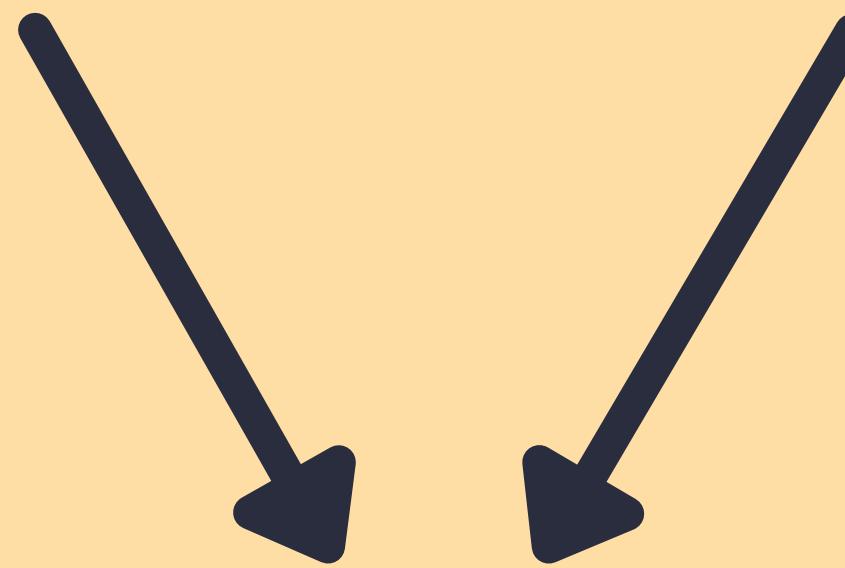


12	9	3	7
----	---	---	---



```
> list1 = [12, 9, 3, 7]  
> list2 = list1
```

list1 list2



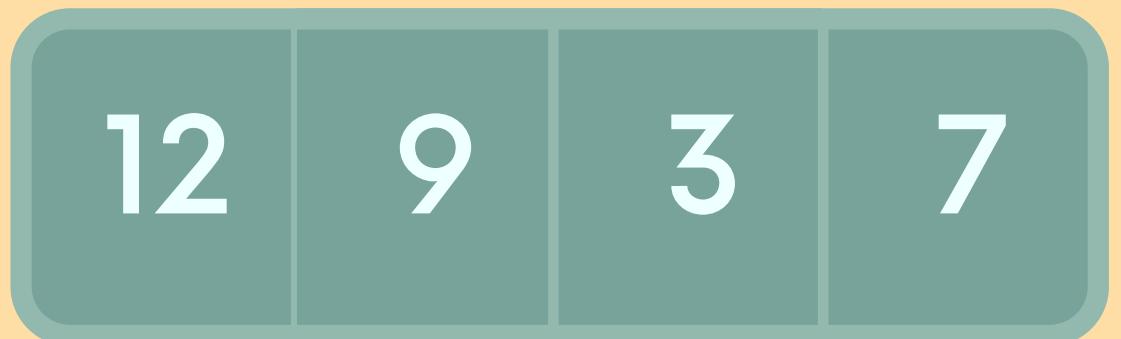
12	9	3	7
----	---	---	---

list1



```
> list1 = [12, 9, 3, 7]
```

list1



```
> list1 = [12, 9, 3, 7]
```

```
> list2 = [12, 9, 3, 7]
```

list1



12	9	3	7
----	---	---	---

list2



12	9	3	7
----	---	---	---



```
> list1 = [12, 9, 3, 7]  
> list2 = [12, 9, 3, 7]
```

`==`



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

True

use `==` to compare the contents inside of two lists. Do they hold the same values?

`is`



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

False

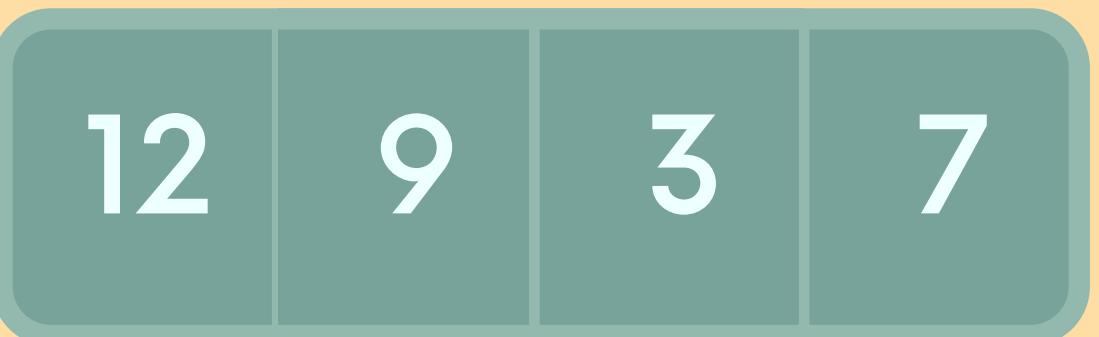
use `is` to compare the identity of two lists. Are they the same "container" in memory?

copy()

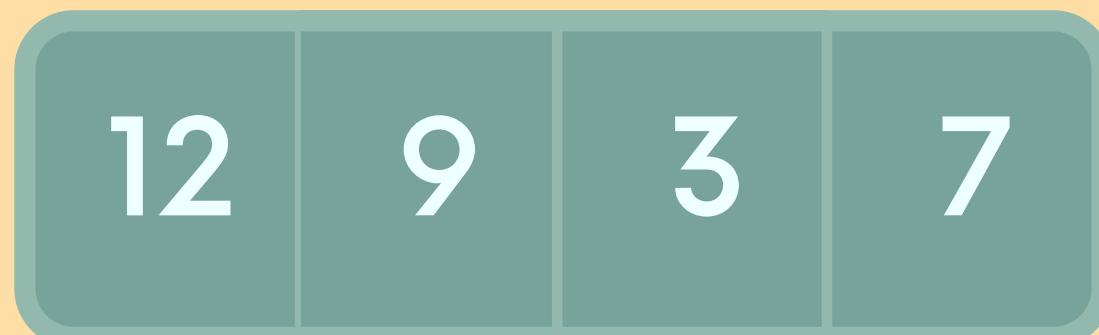
```
...  
> list1 = [12, 9, 3, 7]  
> list2 = list1.copy()
```

The copy method returns a **shallow copy** of a list. Nested objects are not copied.

list1



list2

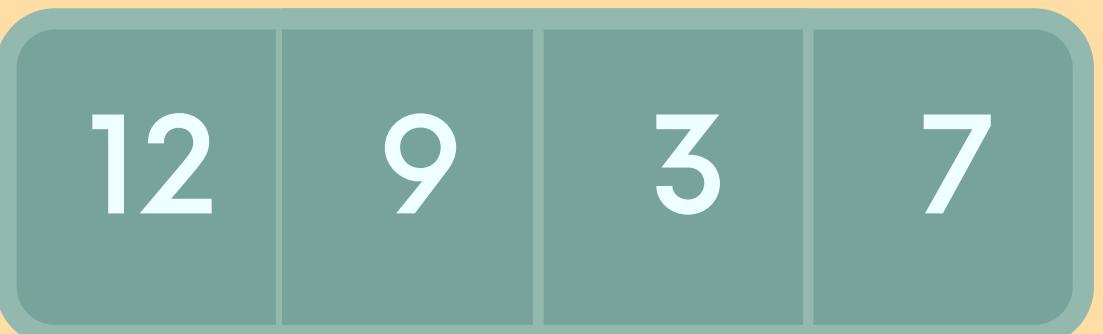
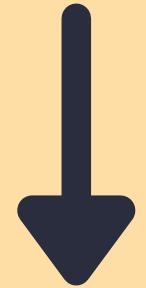


Copying with [:]

```
> list1 = [12, 9, 3, 7]  
> list2 = list1[:]
```

We can also copy lists by creating slices of an entire list. It's not the most readable, but it works!

list1



list2



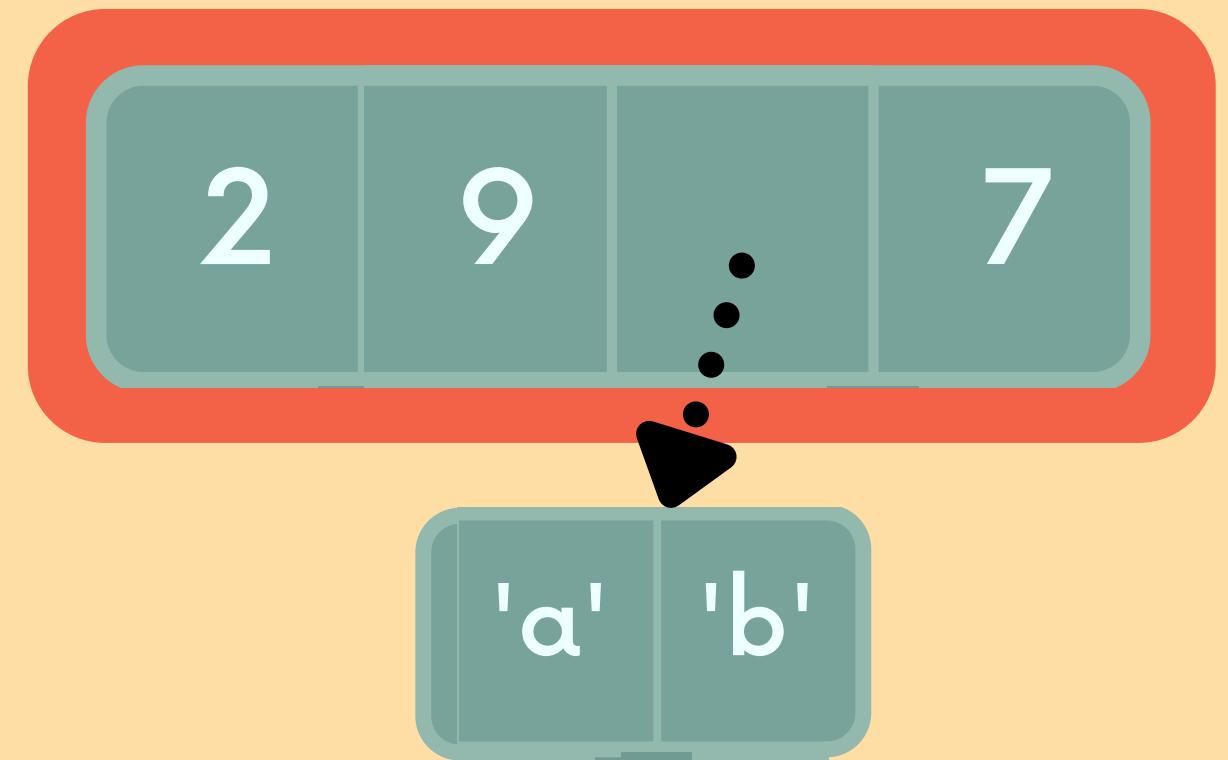
shallow copy



```
list1 = [2, 9, ['a', 'b'], 7]
```

The copy method returns a **shallow copy** of a list. Nested objects are not copied.

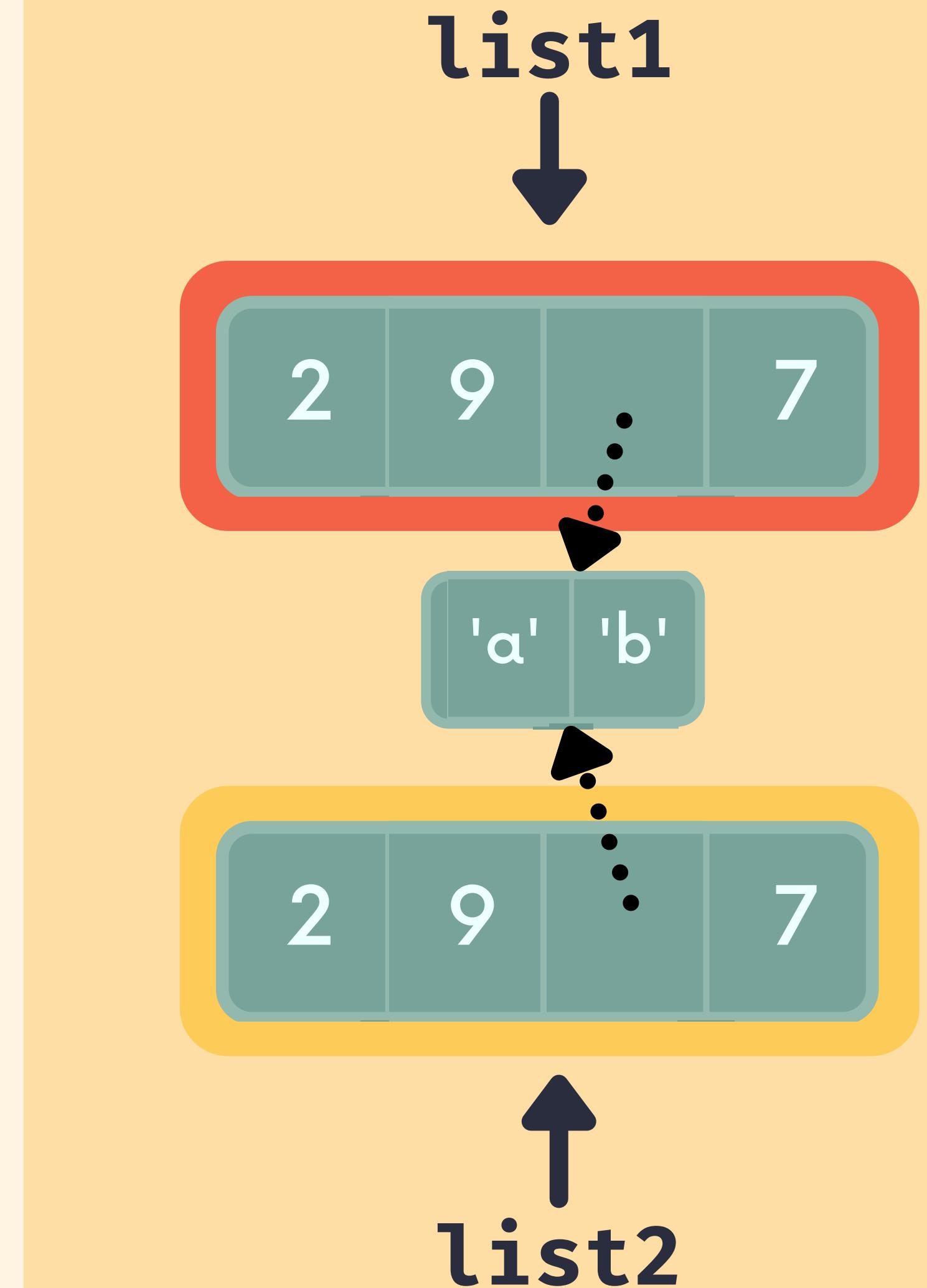
list1
↓



shallow copy

```
...  
list1 = [2, 9, ['a', 'b'], 7]  
list2 = list1.copy()
```

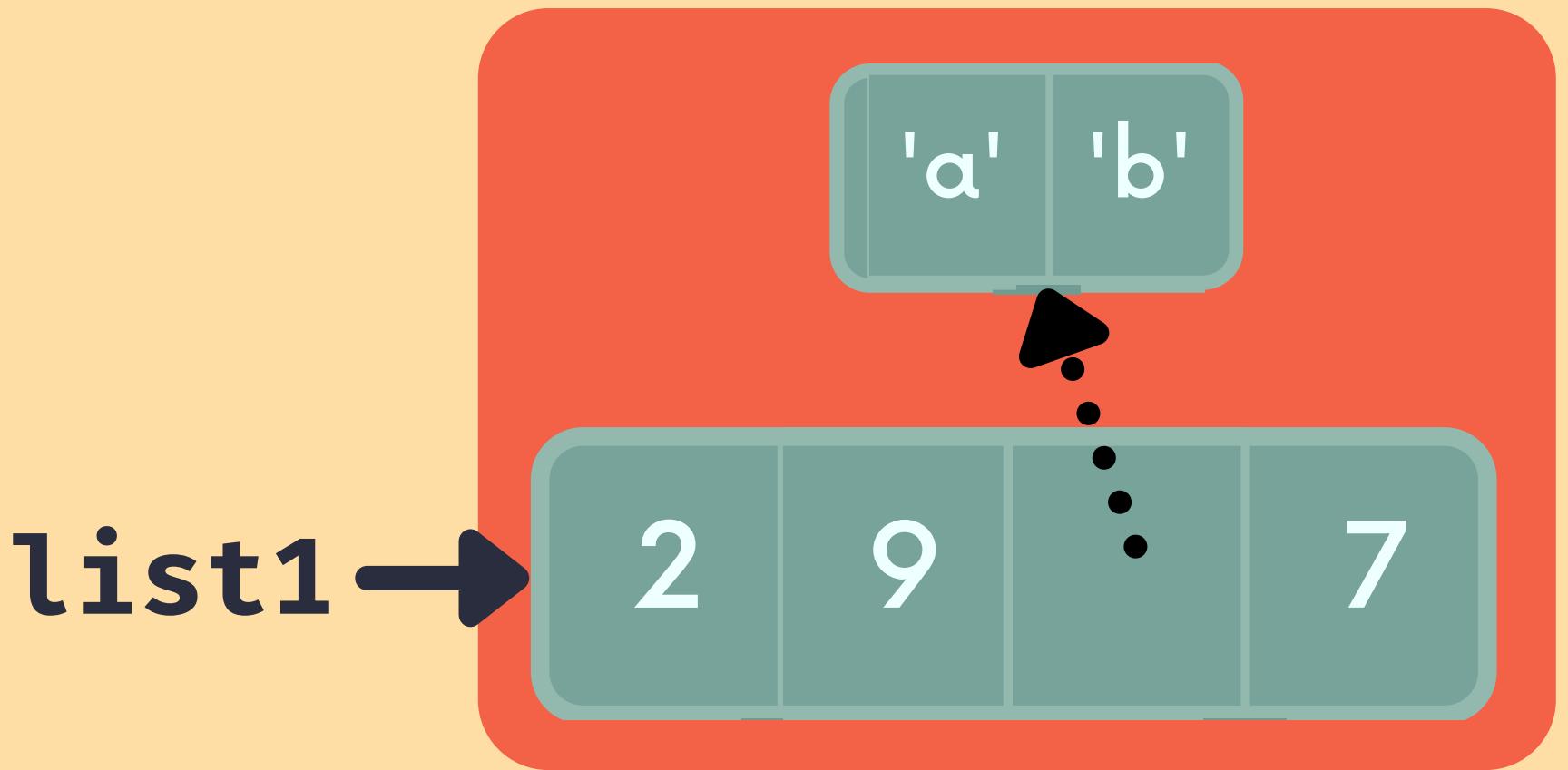
The copy method returns a **shallow copy** of a list. Nested objects are not copied.



deep copy

```
import copy  
list1 = [2,9,['a','b'],7]
```

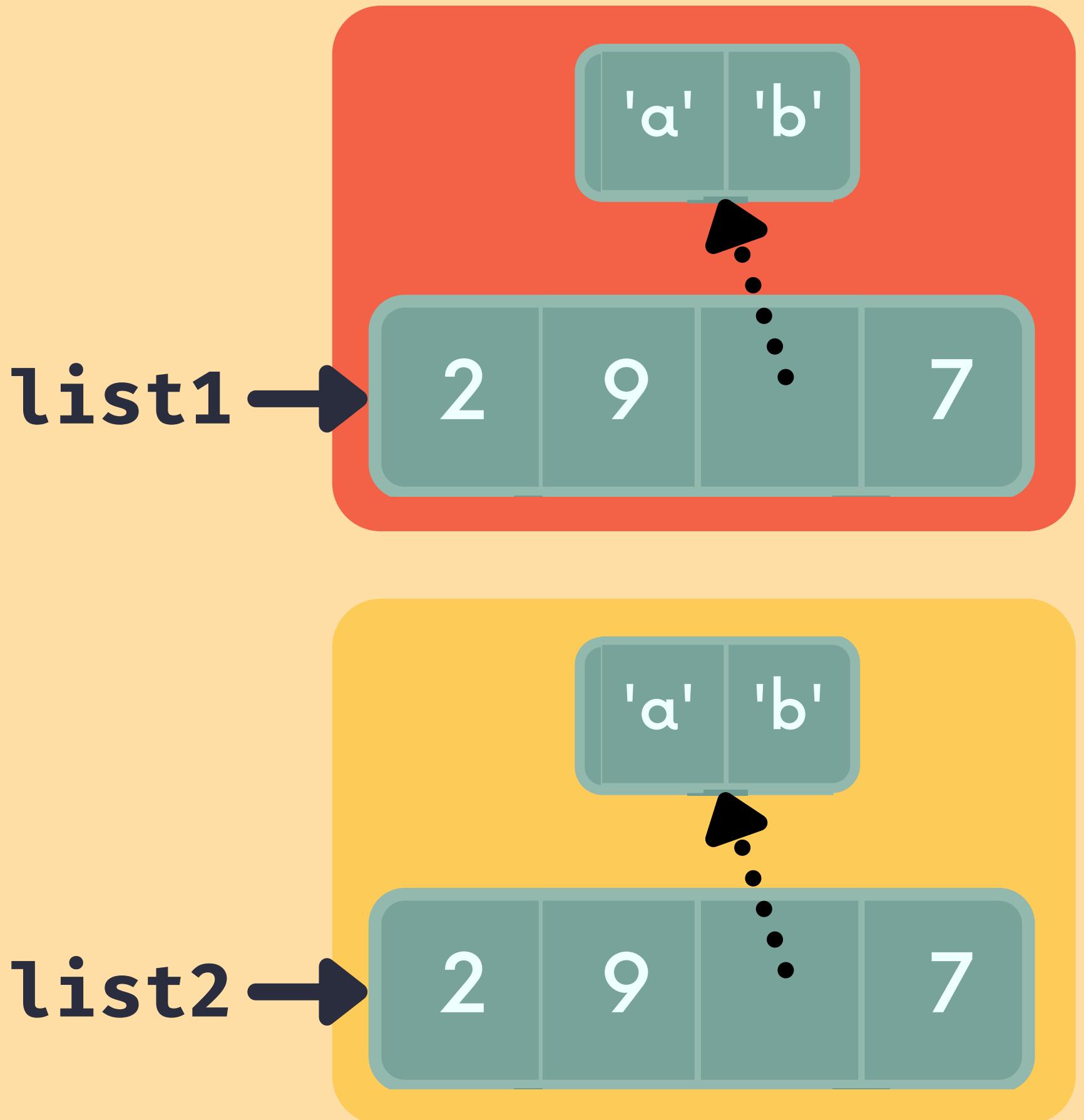
The deepcopy() method will make a copy of a list AND any nested objects contained inside that list.



deep copy

```
import copy  
  
list1 = [2,9,['a','b'],7]  
  
list2 = copy.deepcopy(list1)
```

The deepcopy() method will make a copy of a list AND any nested objects contained inside that list.



clear



```
> langs = ["Python", "C", "JavaScript", "C"]
> langs.clear()
> langs
[]
```

the clear() method removes all items from a list

remove



```
> langs = ["Python", "C", "JavaScript", "C"]
> langs.remove("C")
> langs
[Python, JavaScript, C]
```

The `remove(x)` method will remove the FIRST element in the list that has a value of `x`

pop



```
> langs = ["Python", "C", "JavaScript", "C"]
> langs.pop()
'C'
> langs
['Python', 'C', 'JavaScript']
```

The `pop()` method removes AND returns the last element from a list.

pop(idx)



```
> langs = ["Python", "C", "JavaScript", "C"]
> langs.pop(0)
'Python'
> langs
['C', 'JavaScript', 'C']
```

pop() also accepts a specific index. It will remove the item at that index in the list AND return it

del



```
> langs = ["Python", "C", "JavaScript", "C"]
> del lang[2]
> langs
[Python, C, C]
```

The `del` statement (it's not a method!) can be used to delete an item from a specific index in a list

count



```
> langs = ["Python", "C", "JavaScript", "C"]  
  
> lang.count("C")  
2
```

The count method returns the number of times a value occurs in a list. If the value is not in the list, it returns 0

reverse



```
> nums = [1,2,3,4,5]
> nums.reverse()
> nums
[5, 4, 3, 2, 1]
```

the `reverse()` methods reverses a list **in-place**

sort



```
> nums = [2,8,1,9,3]
> nums.sort()
> nums
[1, 2, 3, 8, 9]
```

the reverse() methods reverses a list in-place

join



```
> fruits = ["Apple", "Kiwi", "Pear"]  
> " ".join(fruits)  
'Apple Kiwi Pear'  
  
> "!!!".join(fruits)  
'Apple!!!Kiwi!!!Pear'
```

`join()` is a string method that joins together the elements of an iterable into a single string. Whatever string you call it on will be used as a separator.

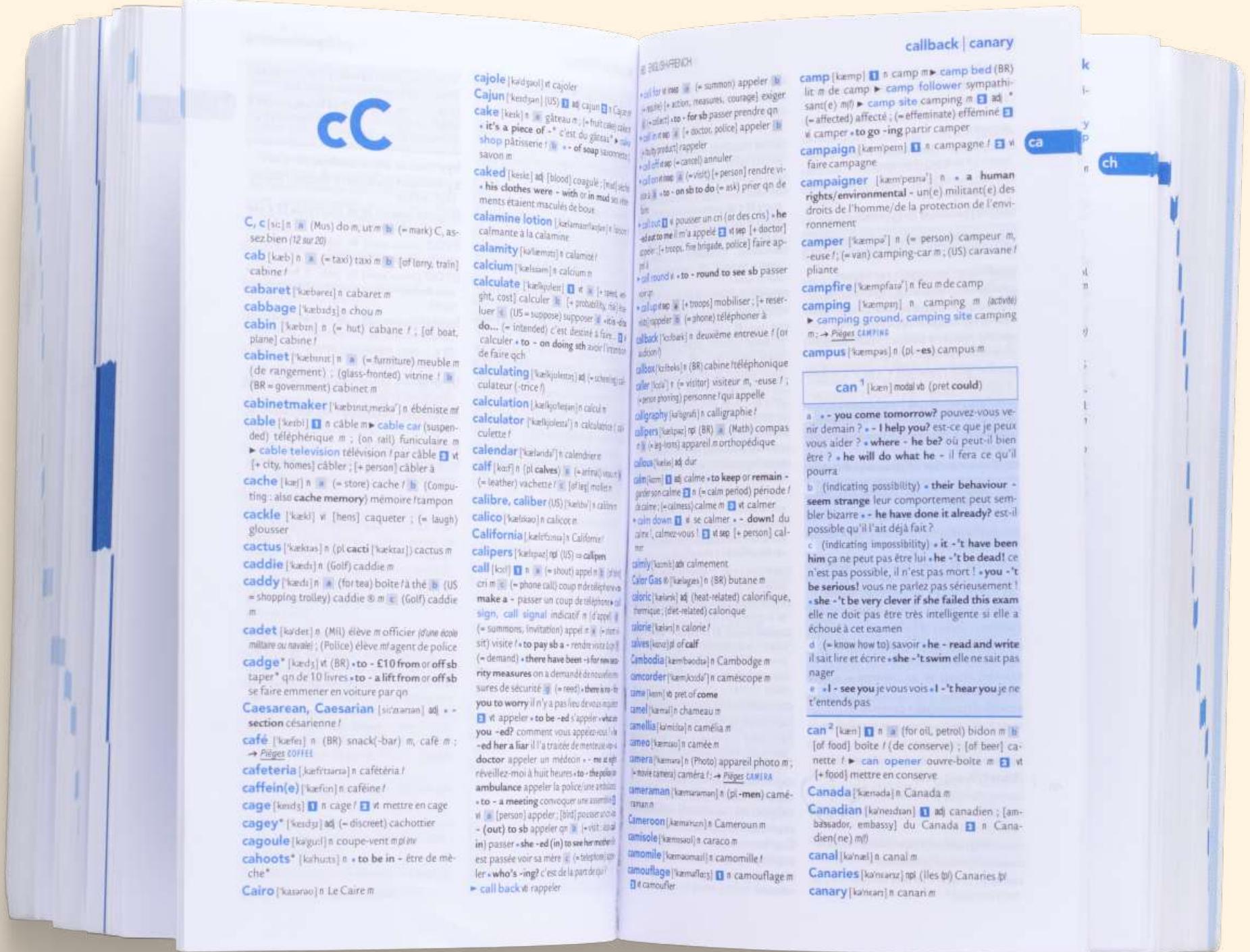
split



```
> birthday = "03/27/2020"  
> birthday.split("/")  
[ '03' , '27' , '2020' ]
```

`split()` is a string method that will split a string on a given character. It returns a list that holds the split strings.

Dictionaries



How Would We Store This?

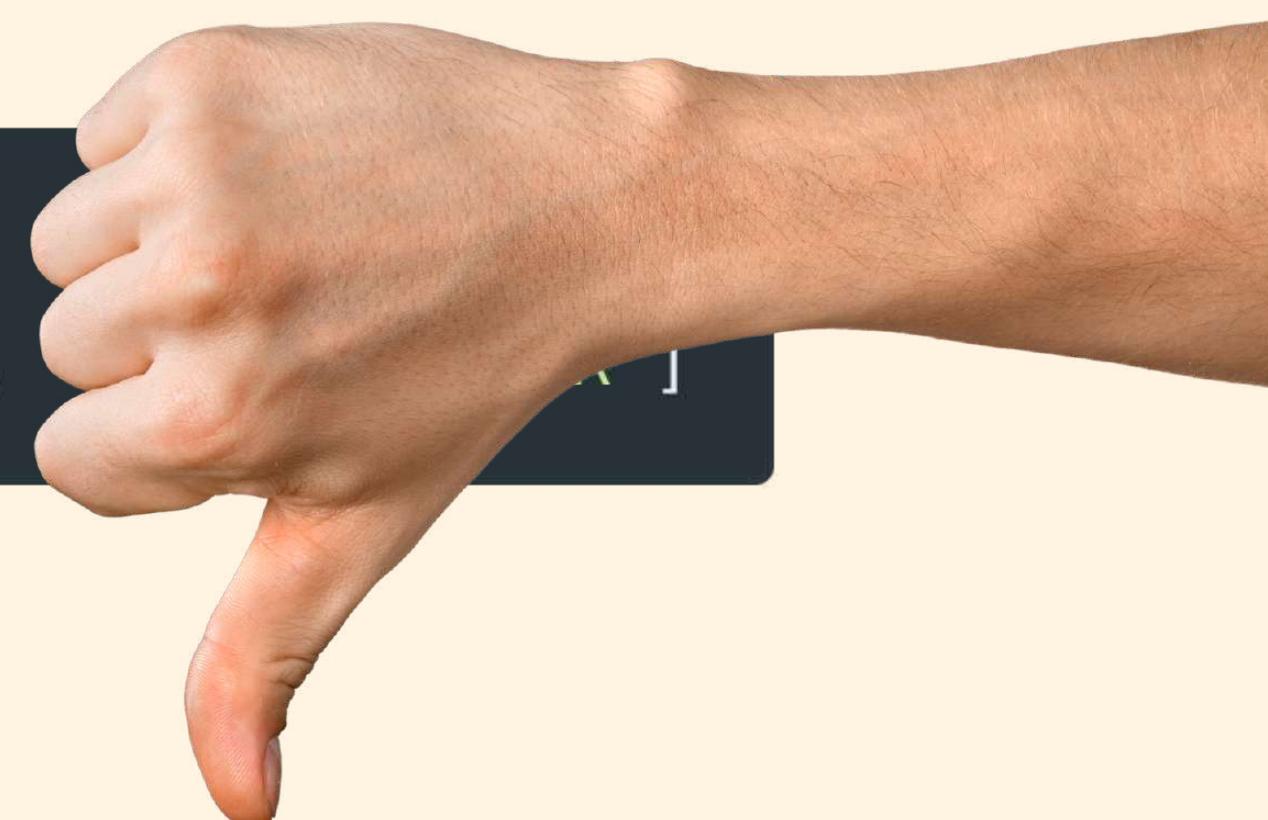


How Would We Store This?



```
movie = ["Amadeus(Director's Cut)", 10379, 8.3, '3h', 2002, 'R']
```

How Would We Store This?



```
movie = ["Amadeus(Director's Cut)", 10379, 8.3,
```

Amadeus (Director's Cut)

★★★★★ (10,379)  8.3 3h 2002 X-Ray AD R

```
movie = {  
    "title": "Amadeus(Director's Cut)",  
    "reviews": 10379,  
    "imdb": 8.3,  
    'runtime': '3h',  
    'year': 2002,  
    'rating': 'R'  
}
```



Key-Value Pairs



```
movie = {  
    "title": "Amadeus(Director's Cut)",  
    "reviews": 10379,  
    "imdb": 8.3,  
    'runtime': '3h',  
    'year': 2002,  
    'rating': 'R'  
}
```



Index-Value Pairs

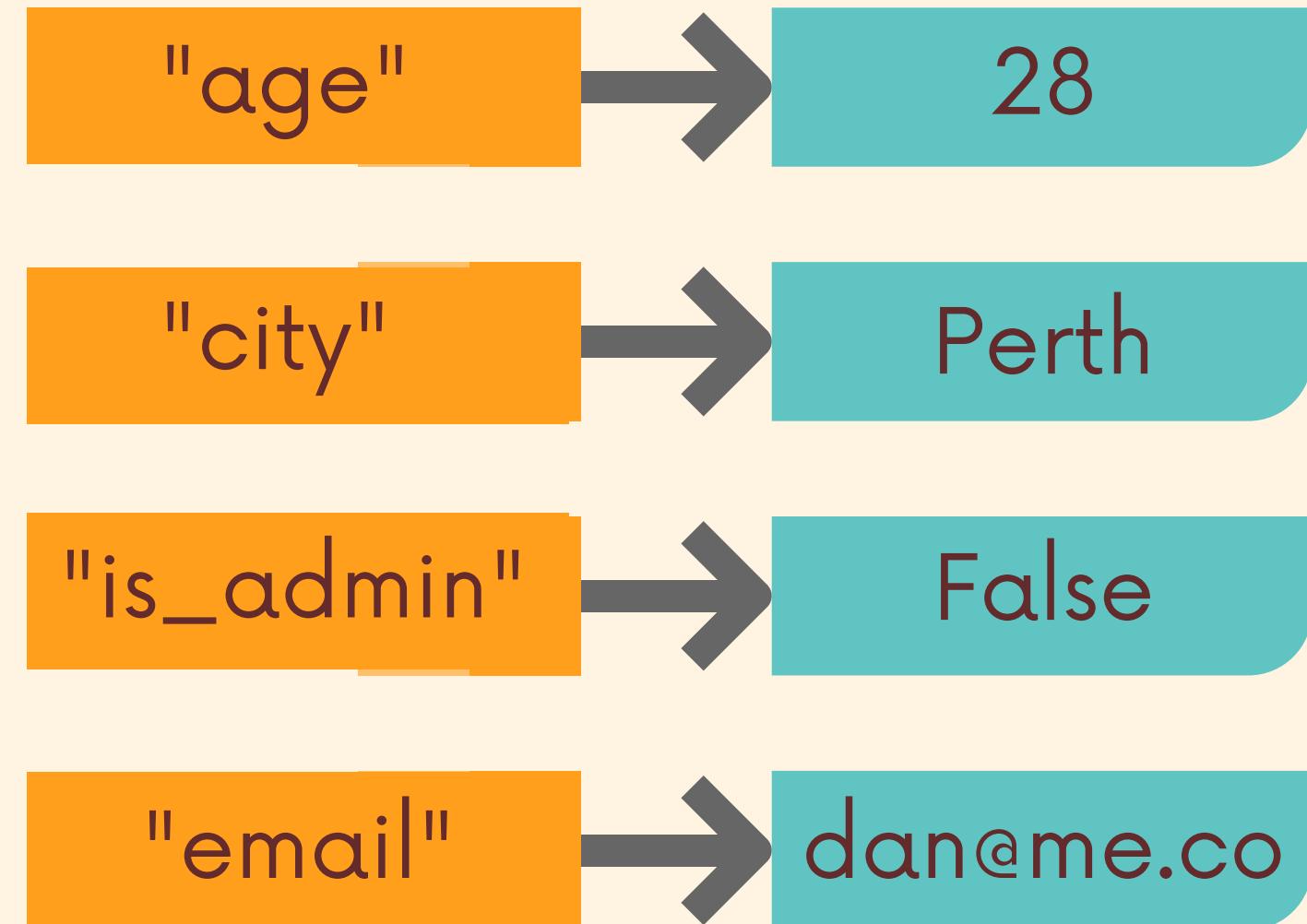
0 → "Monday"

1 → "Tuesday"

2 → "Wednesday"

3 → "Thursday"

Key-Value Pairs



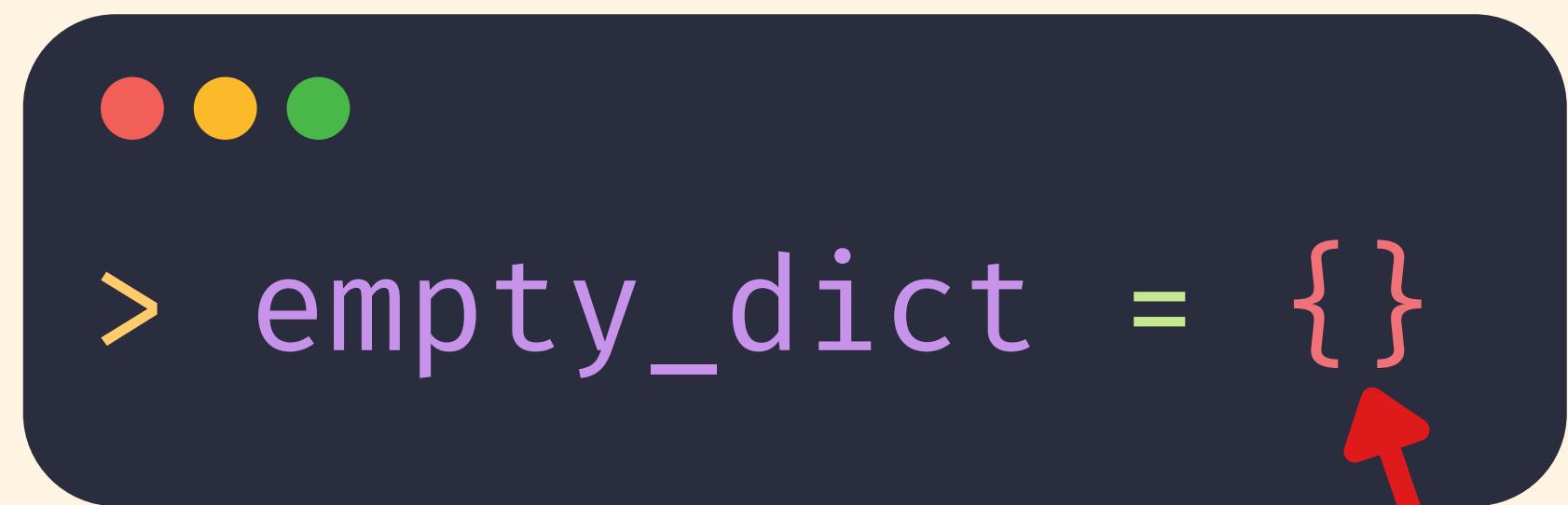


Dictionaries

Dictionaries, known as associative arrays in some other languages, are **indexed by keys** rather than a numerical index

- A dictionary holds key-value pairs
- Keys can be any immutable type: numbers, strings, booleans, etc.
- Values can be whatever you want!



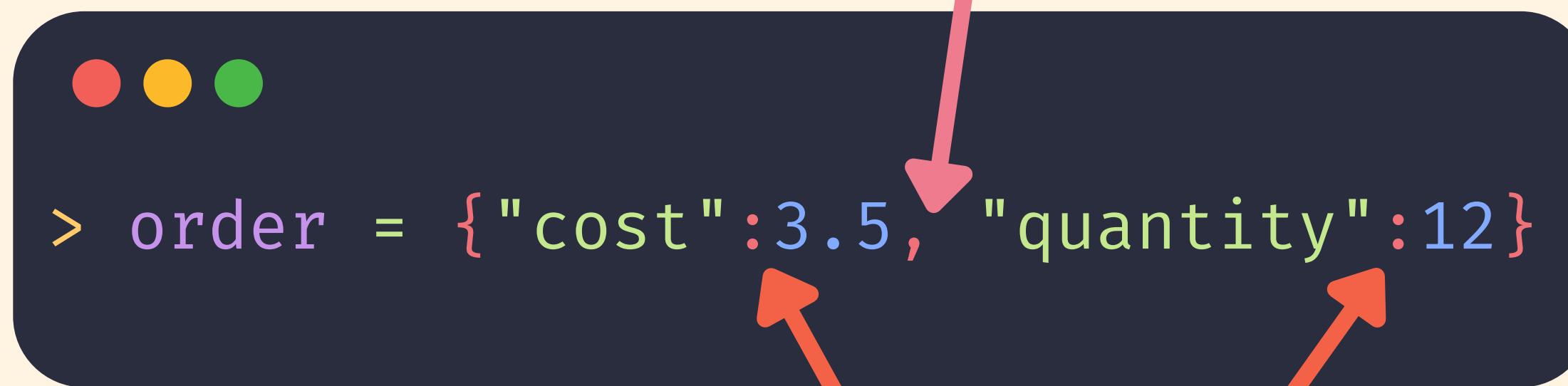


```
> empty_dict = {}
```



Curly Braces

Comma



```
> order = {"cost":3.5, "quantity":12}
```

Colons



```
> order = {"cost":3.5, "quantity":12}
```

cost → 3.5

quantity → 12



Empty Dicts

```
empty_dict = {}  
empty_dict = dict()
```



retrieve values using **dict[key]**

```
● ● ●  
> order = {"cost":3.5, "quantity":12}  
> order["quantity"]  
12
```

cost → 3.5

quantity → 12

retrieve values using **dict[key]**

```
● ● ●  
> order = {"cost":3.5, "quantity":12}  
> order["chicken"]  
KeyError
```

cost → 3.5

quantity → 12



retrieve values using **dict[key]**

```
● ● ●  
> order = {"cost":3.5, "quantity":12}  
> order["chicken"]  
KeyError
```

cost → 3.5

quantity → 12



dict.get()



```
> order = {"cost":3.5, "quantity":12}  
> order.get("chicken")  
  
> order.get("cost")  
3.5
```

The **get()** method will look for a given key in a dictionary. If the key exists, it will return the corresponding value. Otherwise it returns None

update/add values with **dict[key]**



```
> order = {"cost":3.5, "quantity":12}
```

cost → 3.5

quantity → 12

update/add values with **dict[key]**

```
● ● ●  
> order = {"cost":3.5, "quantity":12}  
> order["cost"] = 4.75
```

cost → 4.75

quantity → 12

update/add values with **dict[key]**

```
● ● ●  
> order = {"cost":3.5, "quantity":12}  
> order["cost"] = 4.75  
> order["cost"]  
4.75
```

cost → 4.75

quantity → 12

update/add values with **dict[key]**

```
● ● ●  
> order = {"cost":3.5, "quantity":12}  
> order["shipping"] = 8.99  
> order["shipping"]  
8.99
```

cost → 3.5

quantity → 12

shipping → 8.99

in works with dictionaries too!

```
● ● ●  
> order = {"cost":3.5, "quantity":12}  
  
> 12 in order  
False  
  
> "cost" in order  
True
```

It will only look at the keys, not the values

dict.get()



```
> order = {"cost":3.5, "quantity":12}  
> order.get("chicken")  
  
> order.get("cost")  
3.5
```

The **get()** method will look for a given key in a dictionary. If the key exists, it will return the corresponding value. Otherwise it returns None

.keys, .values, .items

keys()



```
> order = {"cost":3.5, "quantity":12, "product": "taco"}
```

```
> order.keys()
```

```
dict_keys(['cost', 'quantity', 'product'])
```

values()



```
> order.values()
```

```
dict_values([3.5, 12, 'taco'])
```

items()



```
> order.items()
```

```
dict_items([('cost', 3.5), ('quantity', 12),  
(('product', 'taco'))])
```

update



```
> order = {"cost":3.5, "quantity":12}  
> order.update({"product":"taco", "date":"03/14/2019"})  
> order  
{"cost":3.5, "quantity":12, "product":"taco",  
"date":"03/14/2019"}
```

The `update` method will update a dictionary using the key-value pairs from a second dictionary, passed as the argument.

copy



```
> dict1 = {"a":1, "b":2}  
> dict2 = dict1.copy()
```

The **copy** method creates and returns a copy of an existing dictionary. It performs a shallow copy.

* * trick



```
> dict1 = {"a":1, "b":2}  
> dict2 = {"c":3, "d":4}  
> dict3 = **dict1, **dict2  
> dict3  
{ "a":1, "b":2", "c":3, "d":4}
```

We can use two stars `**` to combine multiple dictionaries into a new resulting dictionary.

dict union



```
> dict1 = {"a":1, "b":2}  
> dict2 = {"c":3, "d":4}  
> dict3 = dict1 | dict2  
> dict3  
{"a":1, "b":2", "c":3, "d":4}
```

Python 3.9 added the dict union operator (|) It will return a new dict containing the items from the left and the right dicts. In the case of duplicated keys, the right side "wins"

pop



```
> dict1 = {"a":1, "b": 1, "c":3}  
> pop_value = dict1.pop('b')  
> pop_value  
1
```

The `pop()` method accepts a key and will delete the corresponding key-value pair in the dictionary. It returns the deleted value.

popitem

```
● ● ●  
> dict1 = {"a":1, "b": 1, "c":3}  
> pop_item = dict1.popitem()  
> pop_item  
('c', 3)
```

`popitem()` deletes the most recently added key-value pair. It returns the item as a tuple.

clear

```
...  
> dict1 = {"a":1, "b": 1, "c":3}  
> dict1.clear()  
> dict1  
{}
```

`clear()` deletes all items from a dictionary.

It returns None.

del



```
> dict1 = {"a":1, "b": 1, "c":3}  
> del dict1['a']  
> dict1  
{ "b": 1, "c":3}
```

We can also use the **del statement** to remove items from a dictionary. Remember, it's not a method!

Tuples



Tuples

Immutable, ordered collections



tuples

- Like lists, tuples are ordered, indexed collections
- Unlike lists, **tuples are immutable**. They cannot change once created





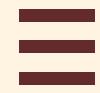
Create a Tuple

```
...  
dishes = ("burrito", "taco", "fajita", "quesadilla")
```

Parentheses

commas





Empty Tuple



```
empty_tuple = ()  
empty_tuple = tuple()
```





1 Item Tuple



```
single_tuple = ("First") ✗
```

```
single_tuple = "First", ✓
```

```
single_tuple = ("First",) ✓
```





1 Item Tuple



```
single_tuple = ("First") ✗
```

```
single_tuple = "First", ✓
```

```
single_tuple = ("First",) ✓
```



Action

Code

Access

```
> dishes = ("burrito", "taco", "fajita", "quesadilla")
> dishes[2]
'fajita'
```

Action

Code

Access

Slice

```
> dishes = ("burrito", "taco", "fajita", "quesadilla")
> dishes[2]
'fajita'
> dishes[1:3]
('taco', 'fajita')
```

Action

Code

Access



Slice



Index



```
> dishes = ("burrito", "taco", "fajita", "quesadilla")
> dishes[2]
'fajita'
> dishes[1:3]
('taco', 'fajita')
> dishes.index("taco")
1
```

Action

Code

Access



```
> dishes = ("burrito", "taco", "fajita", "quesadilla")
```

```
> dishes[2]
```

'fajita'

```
> dishes[1:3]
```

('taco', 'fajita')

```
> dishes.index("taco")
```

1

```
> "nachos" in dishes
```

False

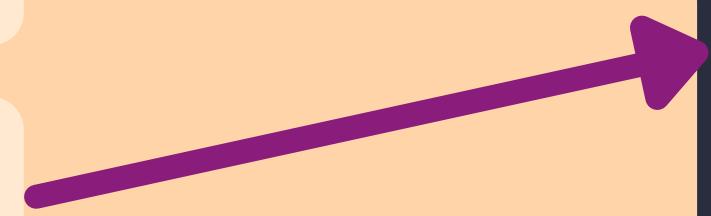
Slice



Index



in



Action

Code

Access



```
> dishes = ("burrito", "taco", "fajita", "quesadilla")
```

```
> dishes[2]
```

'fajita'

```
> dishes[1:3]
```

('taco', 'fajita')

```
> dishes.index("taco")
```

1

```
> "nachos" in dishes
```

False

```
> for dish in dishes:
```

```
    print(dish)
```

burrito

taco

fajita

quesadilla

Slice



Index



in



for

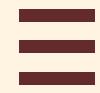


Action

Unpacking

Code

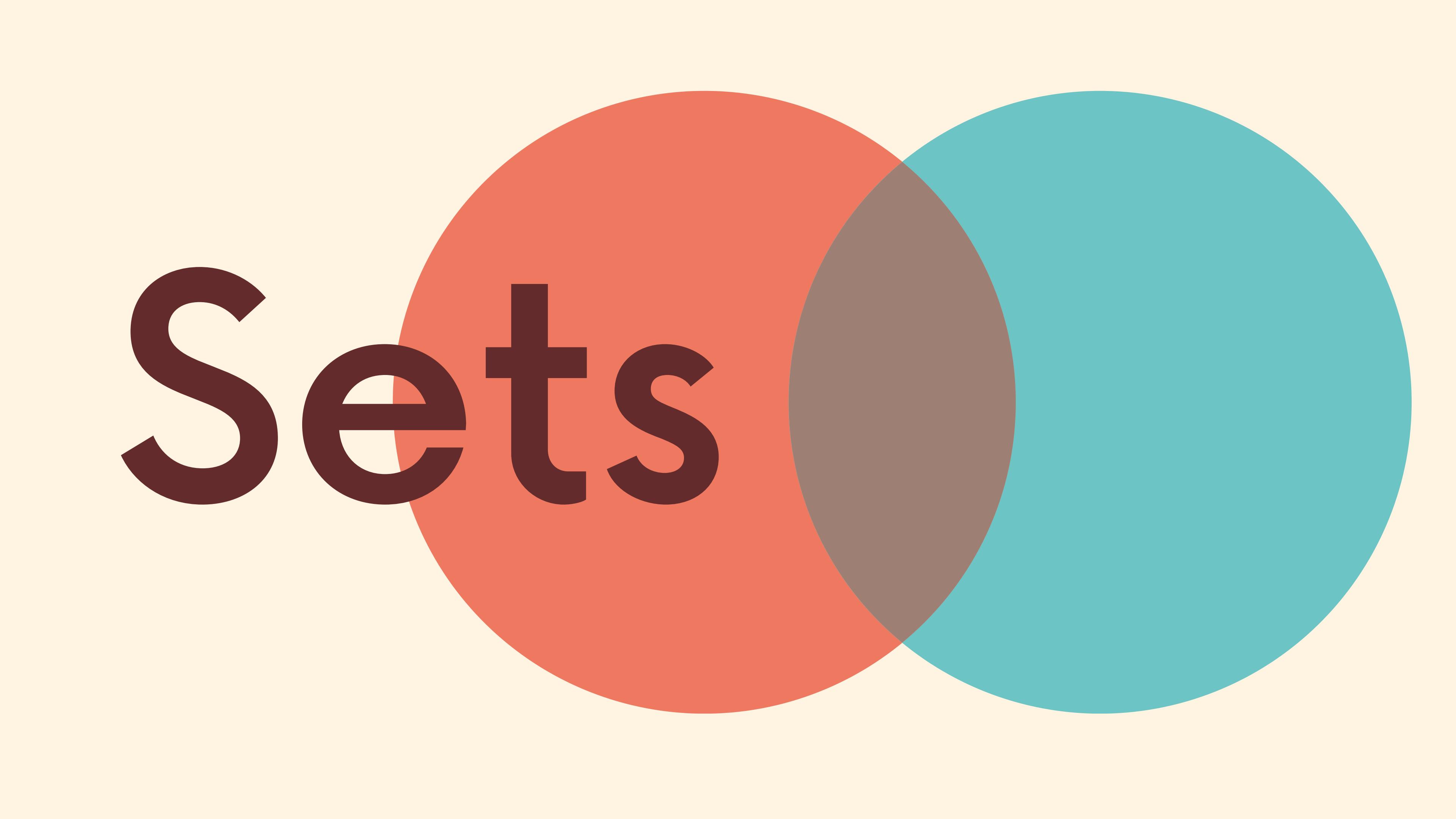
```
> item = ["312", "31th Ave", "New York", "New York"]
> number, street, city, state = item
> number
312
> street
31th Ave
> city
New York
> state
New York
```



why use tuples?

- they are more efficient than lists
- use them for data that shouldn't change
- some methods return them like `dict.items()`
- they can be used as keys in a dictionary





A Venn diagram consisting of two overlapping circles. The left circle is orange and contains the word "Sets" in a large, dark brown serif font. The right circle is teal and overlaps with the orange circle. The background is a light beige color.

Sets

Sets

Unordered collections
with no duplicate elements

Sets

Only immutable elements!

We can...

- Add and delete elements
- Iterate over elements
- Check to see if element is in a set
- Use set operators: union, intersection, etc.



Creating Sets

```
...  
evens = {2, 4, 6, 8}
```





Empty Set



```
empty_set = {} ✗
```

```
empty_set = set() ✓
```



==

Empty Set

```
empty_set = {} ✗  
empty_set = set() ✓
```





add()

```
● ● ●  
> even = {2, 4, 6, 8}  
> even.add(12)  
> even  
{2, 4, 12, 6, 8}
```

Adds a single value to a set. No duplicates in sets!





remove()



```
> langs = {"Python", "C", "JavaScript"}  
> lang.remove("C")  
> langs  
  
{Python, JavaScript}
```

Removes a single value from a set





discard()



```
> langs = {"Python", "C", "JavaScript"}  
> lang.discard("C")  
> langs  
{Python, JavaScript}
```

Like remove() but won't throw error for missing value





clear()



```
> langs = {"Python", "C", "JavaScript"}  
> langs.clear()  
> langs  
set()
```

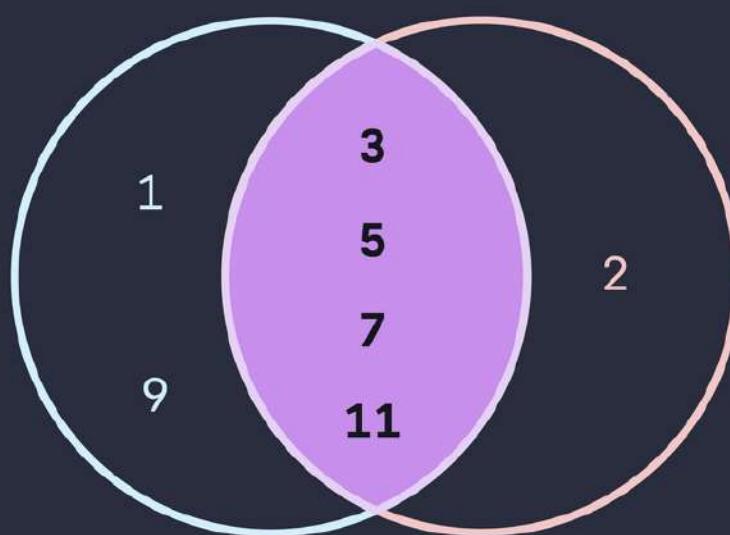
Empties out a set



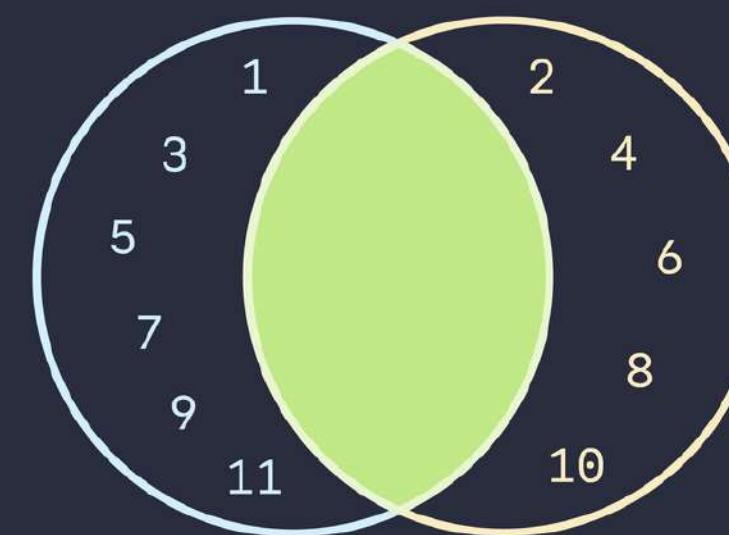
Intersection

returns new set with members
common to left and right

left & right



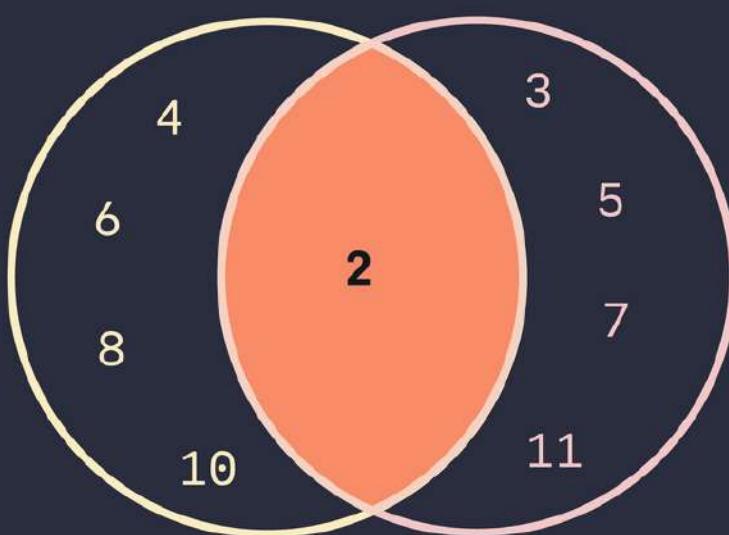
```
> set_odd & set_prime  
{3, 5, 7, 11}
```



```
> set_odd & set_even  
{}
```



```
> set_odd & set_all  
{1, 3, 5, 7, 9, 11}
```

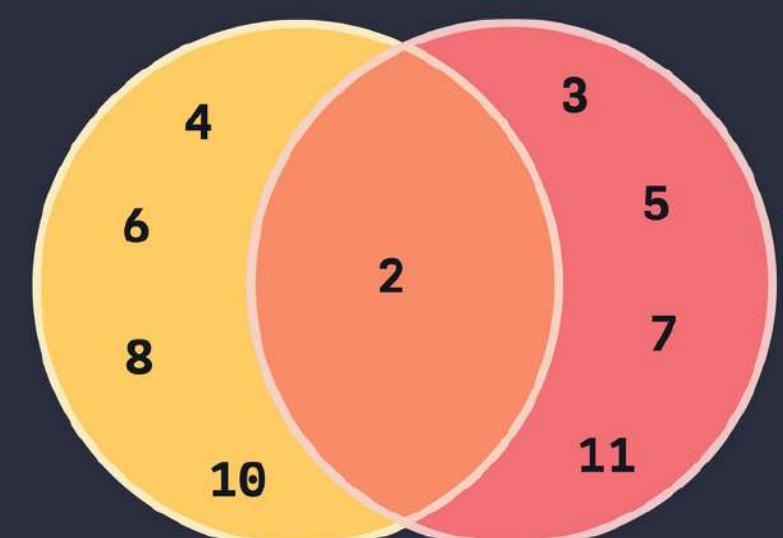
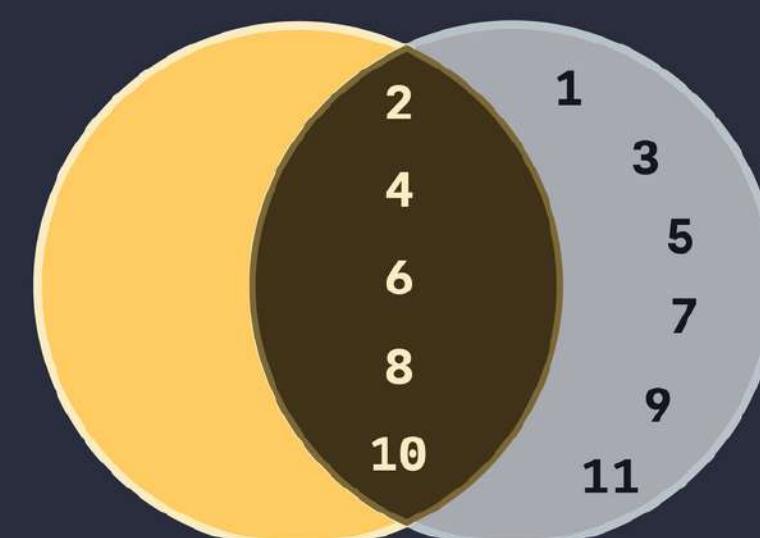
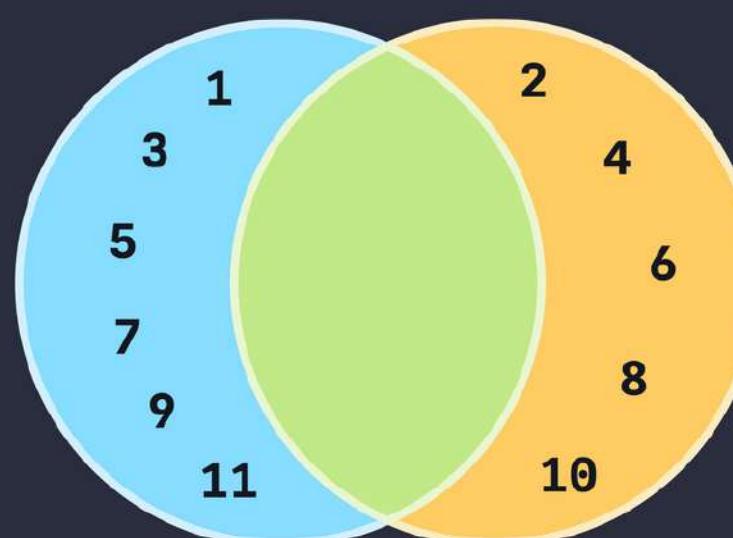
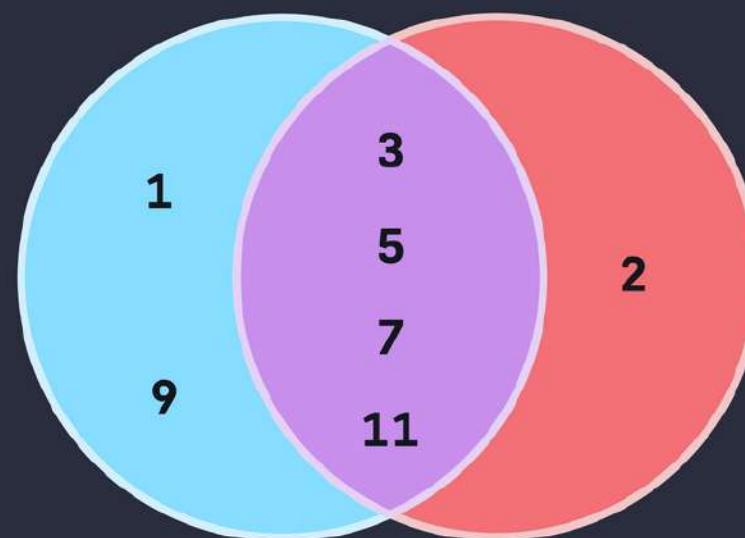


```
> set_even & set_prime  
{2}
```

Union

returns new set with
members from left and right

left | right



```
> set_odd | set_prime
{1, 2, 3, 5, 7, 9, 11}
```

```
> set_odd | set_even
{1, 2, 3, 4, 5, 6,
 7, 8, 9, 10, 11}
```

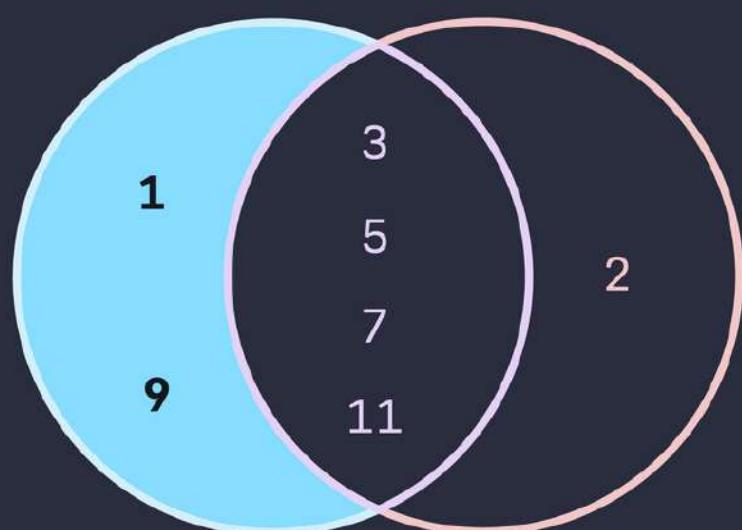
```
> set_even | set_all
{1, 2, 3, 4, 5, 6,
 7, 8, 9, 10, 11}
```

```
> set_even | set_prime
{2, 3, 4, 5, 6, 7, 8,
 10, 11}
```

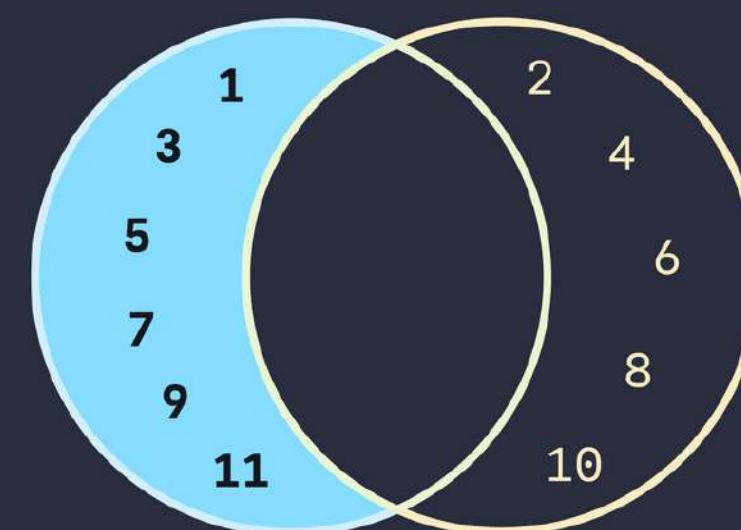
Difference

returns new set with members
from left not in right

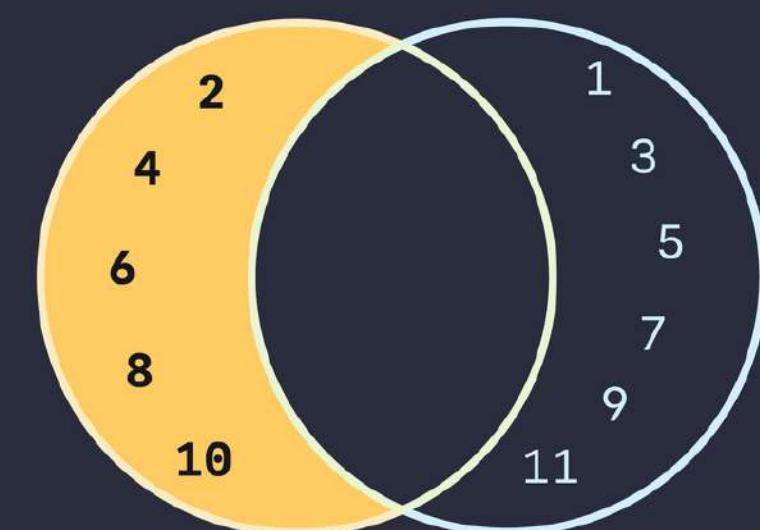
left - right



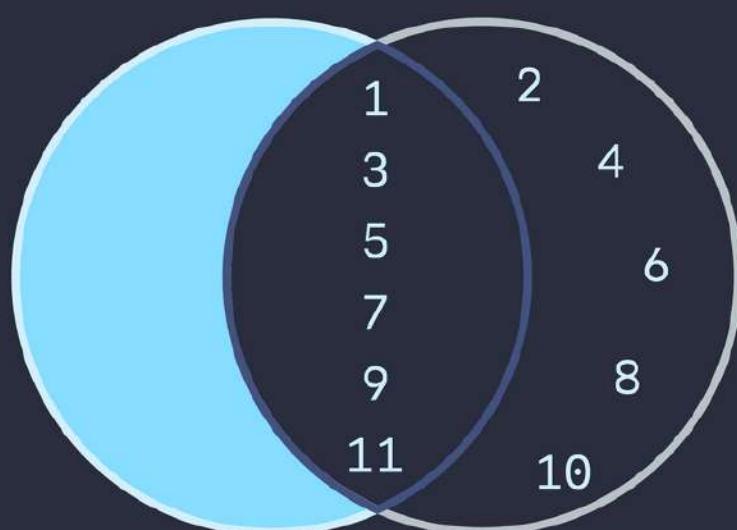
```
> set_odd - set_prime  
{1,9}
```



```
> set_odd - set_even  
{1,3,5,7,9,11}
```



```
> set_even - set_odd  
{2,4,6,8,10}
```



```
> set_odd - set_all  
{}
```

Why use sets?

- Sets make it very easy/fast to check if a value exists in a collection
- Sets are an easy way to remove duplicates from a collection

Returning To Functions





* args

We can use the wildcard or `*` notation to write functions that accept **any number of arguments**





```
def average(*args):
    total = 0
    for arg in args:
        total += arg
    return total/len(args)
```

Gathers all remaining
arguments into a tuple.



```
def average(*args):  
    total = 0  
    for arg in args:  
        total += arg  
    return total/len(args)
```

def average(*args):
 total = 0
 for arg in args:
 total += arg
 return total/len(args)

Pass as many arguments as we want! 5 args in this case:

average(1,2,3,4,5)
3.0



```
def average(*args):  
    total = 0  
    for arg in args:  
        total += arg  
    return total/len(args)
```

```
average(1,2,3,4,5)  
3.0
```

```
average(10,1)  
5.5
```



2 arguments in this example

Name this parameter
whatever you want.
args is common but
NOT required



```
def average(*nums):  
    total = 0  
    for arg in nums:  
        total += arg  
    return total/len(nums)
```

```
average(1,2,3,4,5)  
3.0
```

```
average(10,1)  
5.5
```



* * **kwargs**

We can use the `**` notation to write functions that accept **any number of keyword arguments**





```
def print_ages(**kwargs):
    for k,v in kwargs.items():
        print(f'{k} is {v} years old')
```

Gathers all keyword
arguments into a
dictionary

```
def print_ages(**kwargs):  
    for k,v in kwargs.items():  
        print(f"{k} is {v} years old")
```



```
def print_ages(**kwargs):
    for k,v in kwargs.items():
        print(f'{k} is {v} years old')
```

```
print_ages(max=67,sue=59,kim=14)
max is 67 years old
sue is 59 years old
kim is 14 years old
```

name this whatever you want.
It's just a parameter!

```
def print_ages(**ages):  
    for k,v in ages.items():  
        print(f"{k} is {v} years old")
```

```
print_ages(max=67,sue=59,kim=14)  
max is 67 years old  
sue is 59 years old  
kim is 14 years old
```

Order Matters

parameters

*args

default
parameters

**kwargs

When defining functions, the order of parameters matters!

An Annoying Gotcha

With mutable default arguments

`add_twice` expects a value and a list to be passed in. It appends the value to the list twice and returns the list.

```
def add_twice(val, lst=[]):  
    lst.append(val)  
    lst.append(val)  
    return lst
```

`add_twice` expects a value and a list to be passed in. It appends the value to the list twice and returns the list.



```
def add_twice(val, lst=[]):  
    lst.append(val)  
    lst.append(val)  
    return lst
```

```
add_twice('hi', [1,2,3])  
[1, 2, 3, 'hi', 'hi']
```

`add_twice` expects a value and a list to be passed in. It appends the value to the list twice and returns the list.



```
def add_twice(val, lst=[]):  
    lst.append(val)  
    lst.append(val)  
    return lst
```

```
add_twice('hi', [1,2,3])  
[1, 2, 3, 'hi', 'hi']
```

```
add_twice('lol', ['ha'])  
['ha', 'lol', 'lol']
```



```
def add_twice(val, lst=[]):  
    lst.append(val)  
    lst.append(val)  
    return lst
```



If no list is passed in, we've
added a default value of []

it seems to be
working just fine...

```
...  
def add_twice(val, lst=[]):  
    lst.append(val)  
    lst.append(val)  
    return lst
```

```
add_twice('yay')  
['yay', 'yay']
```



```
def add_twice(val, lst=[]):  
    lst.append(val)  
    lst.append(val)  
    return lst
```

what??

what's going on?

the default value is
being updated each
time it's used!

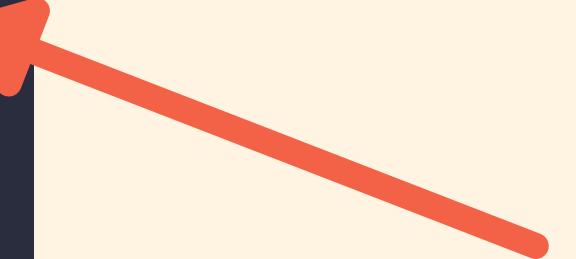
```
add_twice('yay')  
['yay', 'yay']
```

```
add_twice('boo')  
['yay', 'yay', 'boo', 'boo']
```



The Fix

```
def add_twice(val, lst=None):  
    if lst is None:  
        lst = []  
    lst.append(val)  
    lst.append(val)  
    return lst
```



give lst a default
value of None

The Fix

Inside the function check
to see if `lst` is `None`.

If so, set it to an empty list!

```
def add_twice(val, lst=None):  
    if lst is None:  
        lst = []  
    lst.append(val)  
    lst.append(val)  
    return lst
```



```
def add_twice(val, lst=None):  
    if lst is None:  
        lst = []  
    lst.append(val)  
    lst.append(val)  
    return lst
```

```
add_twice('yay')  
['yay', 'yay']  
  
add_twice('boo')  
['boo', 'boo']
```

The Fix

It works!

Argument Unpacking

Turning sequences into separate args



```
def average(*args):  
    total = 0  
    for arg in args:  
        total += arg  
    return total/len(args)
```

This function accepts any number of arguments and returns their average

```
average(1,2,3,4,5)
```

3.0

```
average(10,1)
```

5.5



```
def average(*args):  
    total = 0  
    for arg in args:  
        total += arg  
    return total/len(args)
```

```
nums = [7, 4, 9, 2, 11, 2, 3, 4]  
average(nums)  
TypeError
```

We can't pass a list of values.
The function expects
individual arguments, not a
single collection of numbers



```
def average(*args):  
    total = 0  
    for arg in args:  
        total += arg  
    return total/len(args)
```

```
nums = [7,4,9,2,11,2,3,4]  
average(nums)  
TypeError
```

```
average(*nums) ←
```

5.25

Instead, we can "unpack" the list into individual args using an asterisk

Errors



Common Errors

SyntaxError

NameError

IndexError

TypeError

ValueError

KeyError

raise



```
raise ValueError
```

We can raise our own exceptions (force them happen)
whenever we want, using the `raise` keyword

raise



```
raise ValueError("invalid character")
```

You can also provide a specific message when
raising an exception

try/except



```
raise ValueError("invalid character")
```

You can also provide a specific message when
raising an exception

try/except



```
try:  
    <code that could generate error>  
except:  
    <code that runs if error raised>
```

We can use the try and except keywords to handle exceptions. If an exception is raised in the try block, the except block will run.

try/except

```
● ● ●  
try:  
    num = int(input("Please enter a number: "))  
except ValueError:  
    print("Oh no, that isn't a number!")
```

Usually it's better to except a specific exception and handle it, rather than handling any possible exception that could occur.

Multiple Excepts



```
try:  
    num = int(input("Enter an integer: "))  
    print(10/num)  
except ValueError:  
    print("That's not an int!")  
except ZeroDivisionError:  
    print("Can't divide by zero!")
```

EAFP

Easier to ask for forgiveness than permission

"Assume things exist or will work, and catch exceptions if you're wrong"

Coding style characterized by lots of try/except blocks



Look Before You Leap

Coding style where you explicitly test for pre-conditions before making calls or "leaping". Characterized by lots of if statements

LBYL

"Look"

Check to see if year is numeric.

```
● ● ●  
year = input("Enter a year:")  
if year.isnumeric():  
    year = int(year)  
else:  
    year = 2025
```

"Leap"

Convert year to int once we know it's safe

EAFP

Assume it'll work

Try converting year
to an integer

```
● ● ●  
try:  
    year = int(input("Enter a year:"))  
except ValueError:  
    year = 2025
```

Catch exception if

you're wrong!

This code runs if year
can't be cast to an int

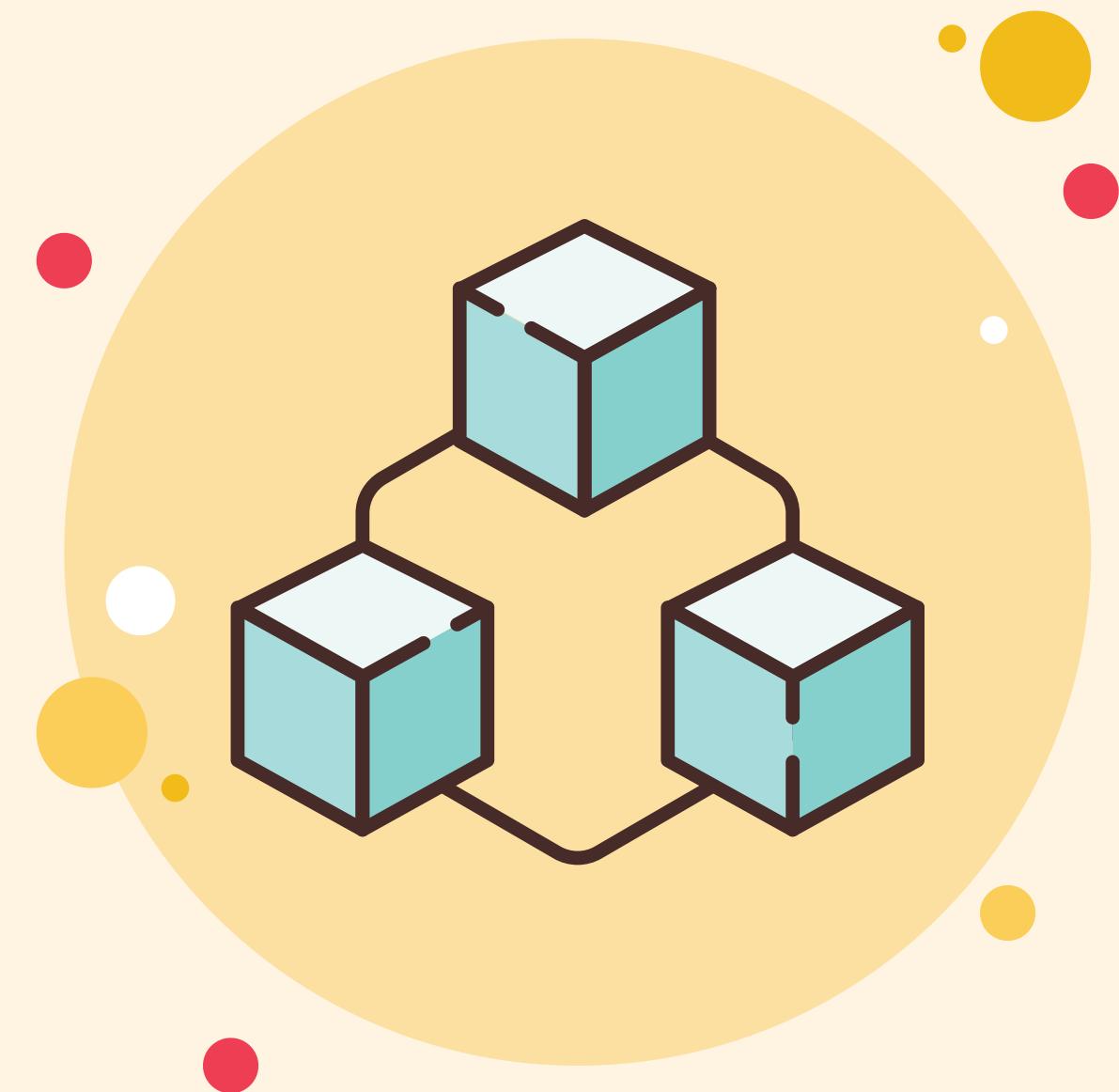
Modules



Modules

A module is simply a Python file that contains code that can be re-used in other files.

Modules allow us to break up complex programs into smaller, more manageable pieces across multiple files.



Built-In
Custom
3rd Party

Standard Library

Python comes with tons of built-in modules that we can use, if we import them.

Each module consists of methods and functionality bundled together



import



```
import random  
random.randint(1, 6)
```

3

To use a module, we must import it first
using the import keyword.

import as



```
import random as rand  
rand.randint(1,6)
```

4

Use the **as keyword** to import a module and give it a custom name in your script.

from...import



```
from random import randint  
randint(1,6)
```

2

Use the **from <module> import <method>** syntax to import specific pieces of a module

from...import



```
from random import randint  
randint(1,6)
```

2

Use the **from <module> import <method>** syntax to import specific pieces of a module

from...import



```
from math import pi, sin
```

```
sin(1)
```

```
0.8414709848078965
```

```
pi
```

```
3.141592653589793
```

import *



```
from random import *
randint(1,6)
```

2

We can import all pieces of a module individually using `import *` however this usually not the best approach to importing!

pip

pip is the Python package installer that we can use to install hundreds of thousands of packages for use in our projects.

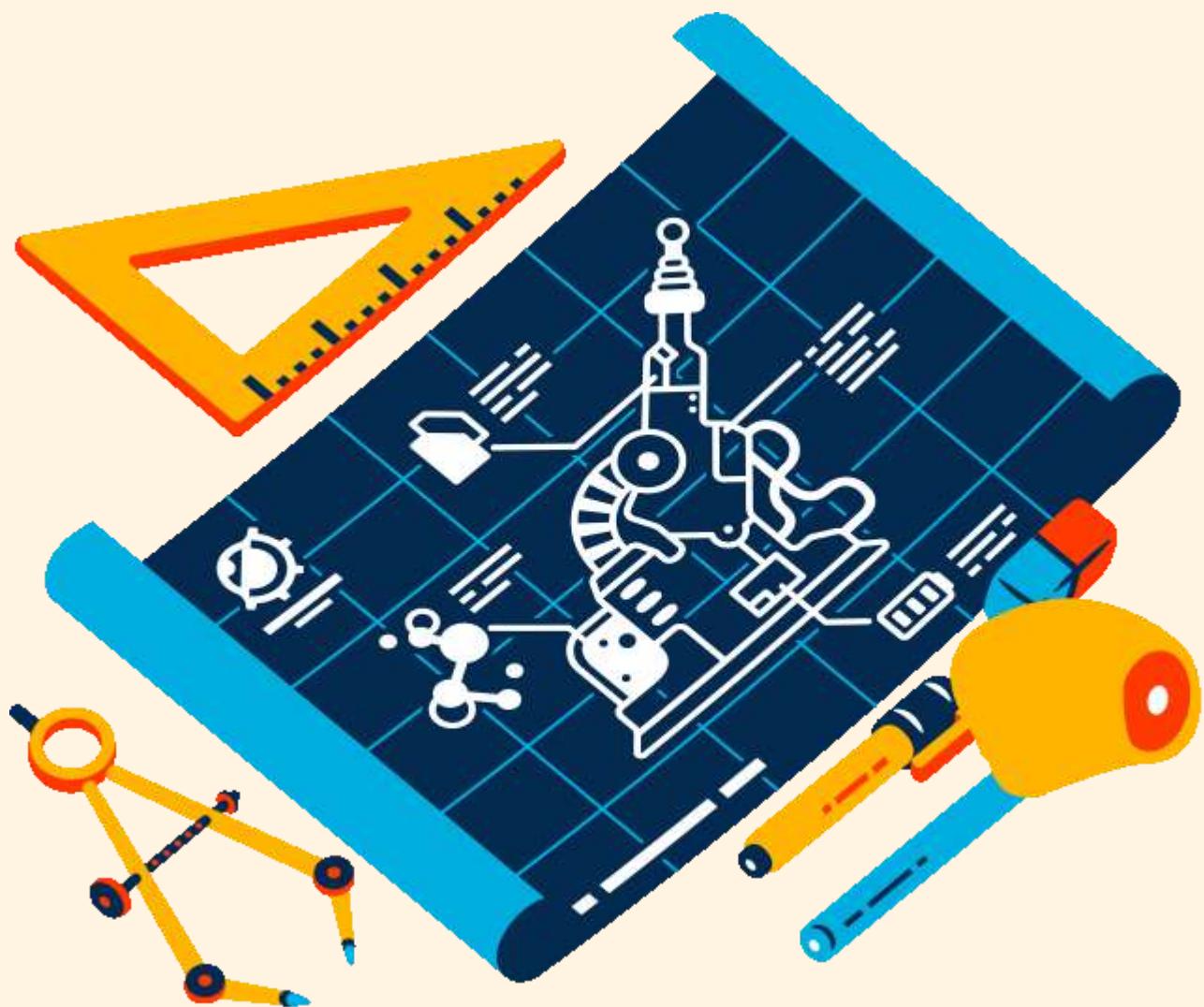


pip install

```
> python3 -m pip install <package>
```

To install a package, use `python3 -m pip install`
followed by the exact name of the package

OOP



a way of
organizing code

WTF is OOP

Object Oriented Programming is an approach to programming that involves modeling "things" into Python objects.

These objects can contain both **data AND functionality** all wrapped up together into a reusable component.



Chess

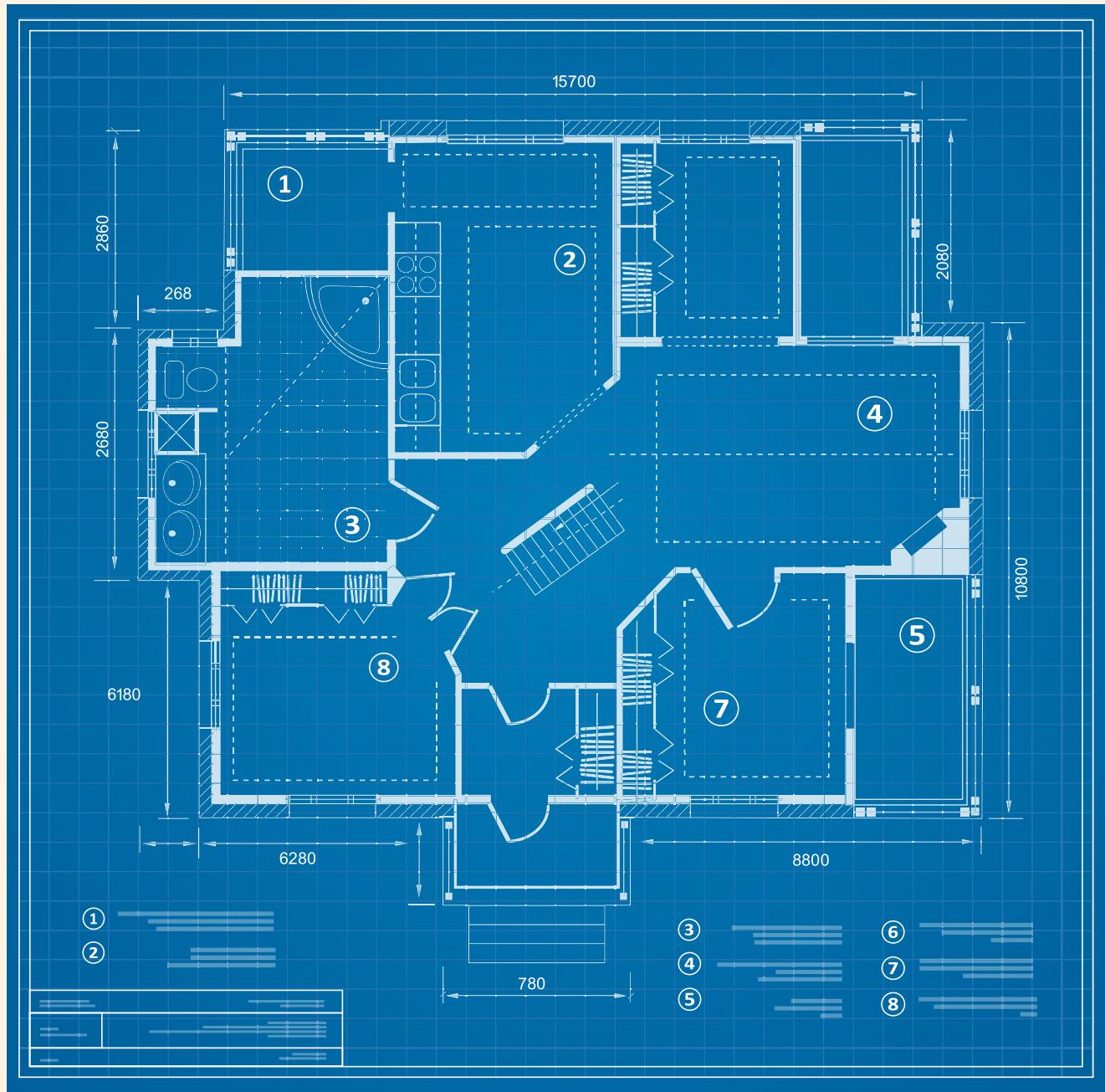
- Board
- Piece
- Player
- AIPlayer
- Match



Class

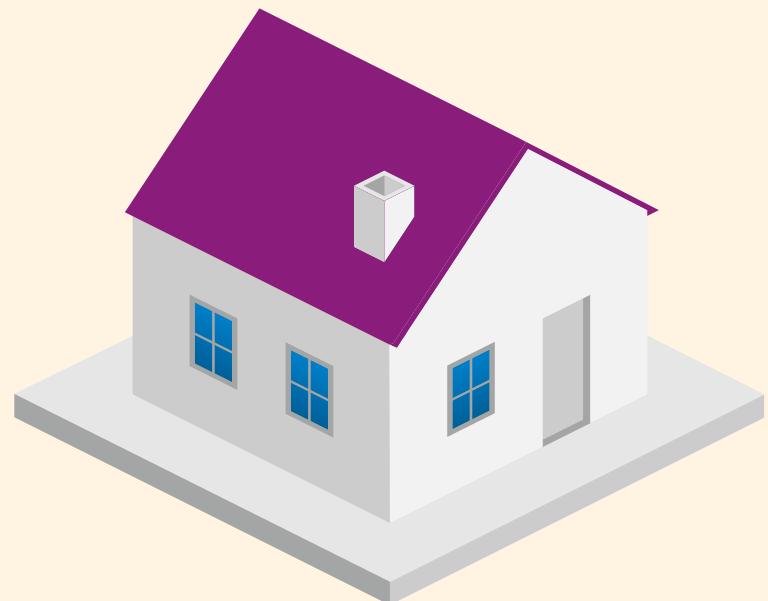
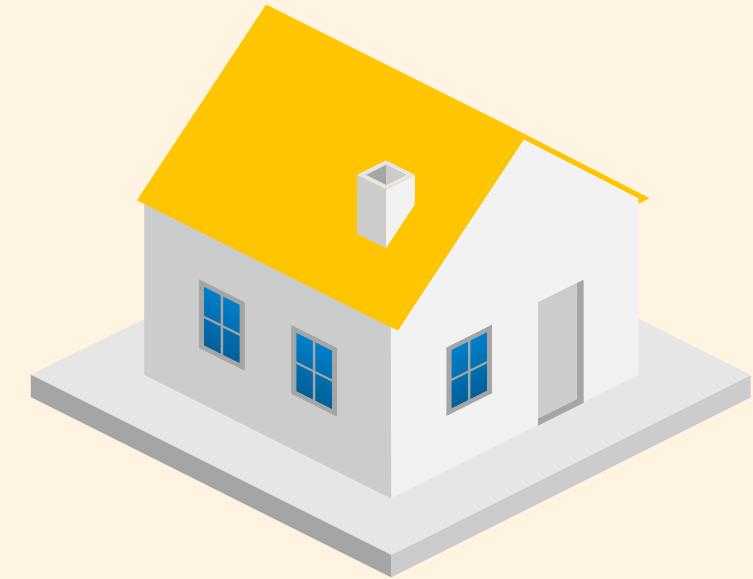
Classes are blueprints or recipes that we can later use to create objects from.

A class describes what properties and functionality individual objects will contain



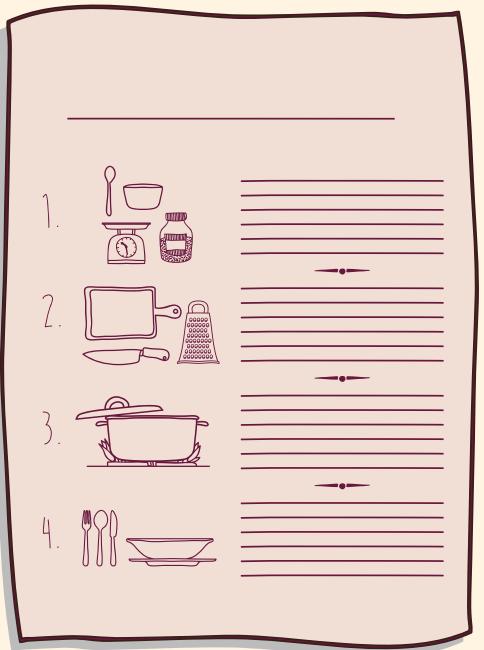
Instance

We call the individual objects
that are created from a class
blueprint instances.



Class

Cupcake Recipe



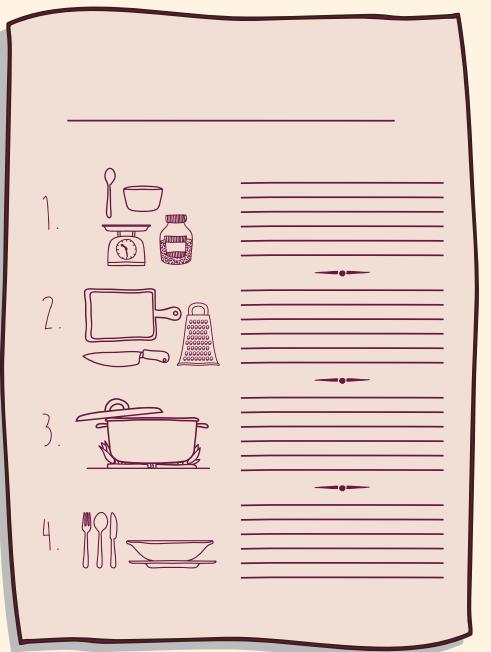
- flour type
- flavor
- frosting
- color

Instances



Class

Cupcake Recipe



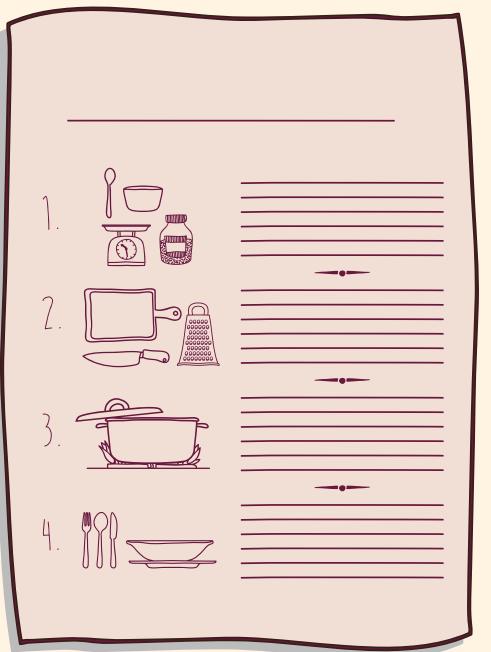
- flour type
- flavor
- frosting
- color

Instances



Class

Cupcake Recipe



- flour type
- flavor
- frosting
- color

Instances



Class

ChessPiece

Data

- color
- piece_type

Instances

Class

ChessPiece

Data

- color
- piece_type

Instances



- white
- knight

Class

ChessPiece

Data

- color
- piece_type

Instances



- white
- knight



- black
- pawn

Class

Player

Data

- name
- class
- inventory
- health

Functionality

- save()
- restart()
- heal()

Instances

Class

Player

Data

- name
- class
- inventory
- health

Functionality

- save()
- restart()
- heal()

Instances

- Titus
- Warrior
- ['sword']
- 81



class

Instances

Player

Data

- name
 - class
 - inventory
 - health

Functionality

- save()
 - restart()
 - heal()

- Titus
 - Warrior
 - ['sword']
 - 81
 - Albus
 - Mage
 - ['staff', 'hat']
 - 100



Class

Task

Data

- name
- description
- status
- priority
- owner

Functionality

- assign()
- complete()

Instances

Class

Task

Data

- name
- description
- status
- priority
- owner

Functionality

- assign()
- complete()

Instances

- Bugfix
- Fix the double click bug on home page navbar
- Medium Priority
- Theresa

Class

Task

Data

- name
- description
- status
- priority
- owner

Functionality

- assign()
- complete()

Instances

- Bugfix
- Fix the double click bug on home page navbar
- Medium Priority
- Theresa

- Login Ticket
- Create new login screen for app
- High Priority
- Baker

Class

Deck

Data

- cards

Functionality

- shuffle()
- deal()

Instances

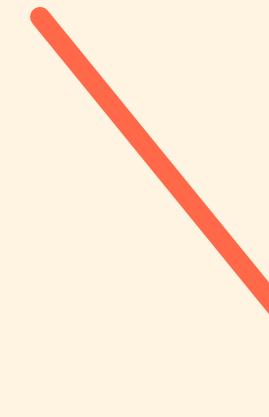






```
class Puppy:  
    def __init__(self, name):  
        self.name = name  
        self.tricks = []
```

Class Keyword



```
class Puppy:  
    def __init__(self, name):  
        self.name = name  
        self.tricks = []
```

Class Name (Capitalized)

```
● ● ●  
class Puppy:  
    def __init__(self, name):  
        self.name = name  
        self.tricks = []
```



Special init method
automatically called
whenever new Puppy
is created

```
class Puppy:  
    def __init__(self, name):  
        self.name = name  
        self.tricks = []
```

self keyword

The self keyword
refers to the "current"
instance of Puppy

```
class Puppy:  
    def __init__(self, name):  
        self.name = name  
        self.tricks = []
```



self keyword

self must be the first parameter to init

```
class Puppy:  
    def __init__(self, name):  
        self.name = name  
        self.tricks = []
```



```
class Puppy:  
    def __init__(self, name):  
        self.name = name  
        self.tricks = []
```



```
elton = Puppy("Elton")  
elton.name  
'Elton'  
elton.tricks  
[]
```

instantiating

To create a Puppy instance, call Puppy() and provide a name





```
class Puppy:  
    def __init__(self, name):  
        self.name = name  
        self.tricks = []
```

Init

The init method runs.
self.name is set to "Elton"
self.tricks is set to []



```
elton = Puppy("Elton")  
elton.name  
'Elton'  
elton.tricks  
[]
```



```
class Puppy:  
    def __init__(self, name):  
        self.name = name  
        self.tricks = []
```



```
otter = Puppy("Otter")  
otter.name  
'Otter'  
otter.tricks  
[]
```

Do it again!

Every time we call `Puppy()` we're creating a new Puppy instance with its own name and tricks list.



Add a method!

This `add_trick()` method appends a new trick to a Puppy instance's tricks list

```
class Puppy:  
    def __init__(self, name):  
        self.name = name  
        self.tricks = []  
  
    def add_trick(self, new_trick):  
        self.tricks.append(new_trick)
```

```
class Puppy:  
    def __init__(self, name):  
        self.name = name  
        self.tricks = []  
  
    def add_trick(self, new_trick):  
        self.tricks.append(new_trick)
```

Call it on an instance!
calling add_trick() on elton
adds 'sit' to his tricks list

```
elton = Puppy("Elton")  
elton.tricks  
[]  
elton.add_trick('sit')  
elton.tricks  
['sit']
```

**add another
instance method**
perform_trick will
perform a trick if a
Puppy instance
knows the trick.

```
● ● ●  
  
class Puppy:  
    def __init__(self, name):  
        self.name = name  
        self.tricks = []  
  
    def add_trick(self, new_trick):  
        self.tricks.append(new_trick)  
  
    def perform_trick(self, trick):  
        if trick in self.tricks:  
            print(f"{self.name} performs {trick}!")  
        else:  
            print(f"{self.name} doesn't know {trick}")
```

Each Puppy instance is a different object

Elton knows sit but not stay

```
elton = Puppy('elton')
elton.add_trick('sit')

elton.perform_trick('sit')
"elton performs sit!"

elton.perform_trick('stay')
"elton doesn't know stay"
```

```
lando = Puppy('Lando')
lando.add_trick('sit')
lando.add_trick('down')
lando.add_trick('stay')

lando.perform_trick('stay')
" Lando performs stay!"
```

Lando is a totally
separate Puppy with
his own tricks list

Lando knows sit, down, and stay



```
elton = Puppy('elton')
elton.add_trick('sit')

elton.perform_trick('sit')
"elton performs sit!"

elton.perform_trick('stay')
"elton doesn't know stay"
```



```
lando = Puppy('Lando')
lando.add_trick('sit')
lando.add_trick('down')
lando.add_trick('stay')

lando.perform_trick('stay')
"Lando performs stay!"
```

Class Attributes

species is defined
on the class itself

All instances of Puppy
will have the same
value for species

```
class Puppy:  
    species = 'canine'  
  
    def __init__(self, name):  
        self.name = name  
        self.tricks = []
```

```
class Puppy:  
  
    species = 'canine'  
  
    def __init__(self, name):  
        self.name = name  
        self.tricks = []
```

All instances of Puppy
will have the same
value for species

```
bozo = Puppy('Bozo')  
enya = Puppy('Enya')  
  
bozo.species  
'canine'  
  
enya.species  
'canine'
```

Class Methods



```
class Puppy:  
    species = 'canine'  
  
    @classmethod  
    def register_anon(cls):  
        return cls('unknown')  
  
    def __init__(self, name):  
        self.name = name  
        self.tricks = []
```

We can define methods that are available on the class directly. These class methods are not concerned with a specific instance of the class.

Class Methods

We use the
`@classmethod`
decorator

```
class Puppy:  
    species = 'canine'  
  
    @classmethod  
    def register_anon(cls):  
        return cls('unknown')  
  
    def __init__(self, name):  
        self.name = name  
        self.tricks = []
```

Class Methods



```
class Puppy:  
    species = 'canine'  
  
    @classmethod  
    def register_anon(cls):  
        return cls('unknown')  
  
    def __init__(self, name):  
        self.name = name  
        self.tricks = []
```



use `cls` as the first parameter. It will refer to the class itself

Class Methods

```
● ● ●  
class Puppy:  
    species = 'canine'  
  
    @classmethod  
    def register_anon(cls):  
        return cls('unknown')
```

```
def __init__(self, name):  
    self.name = name  
    self.tricks = []
```

We can call the method
directly on the class!

```
● ● ●  
stray = Puppy.register_anon()
```

Inheritance

```
class Cat:  
    def __init__(self, name):  
        self.name = name  
    def meow(self):  
        print(f"{self.name} meows")
```

```
class Lion(Cat):  
    def roar(self):  
        print(f"{self.name} roars")
```

The Lion class inherits
from the base Cat class

Inheritance



```
class Cat:  
    def __init__(self, name):  
        self.name = name  
    def meow(self):  
        print(f"{self.name} meows")
```



```
class Lion(Cat):  
    def roar(self):  
        print(f"{self.name} roars")
```



```
eli = Lion('Eli')  
eli.meow()  
"Eli meows"
```

eli is a Lion, but he can
meow because Lion
inherits from Cat.

Super()

```
class Cat:  
    def __init__(self, name):  
        self.name = name  
    def meow(self):  
        print(f"{self.name} meows")
```

use `super()` to refer to the base or parent class.

In this case, we use `super()` to access the `__init__` method on from the `Cat` class.

```
class Lion(Cat):  
    def __init__(self, name, pride_name):  
        super().__init__(name)  
        self.pride_name = pride_name  
  
    def roar(self):  
        print(f"{self.name} roars")
```