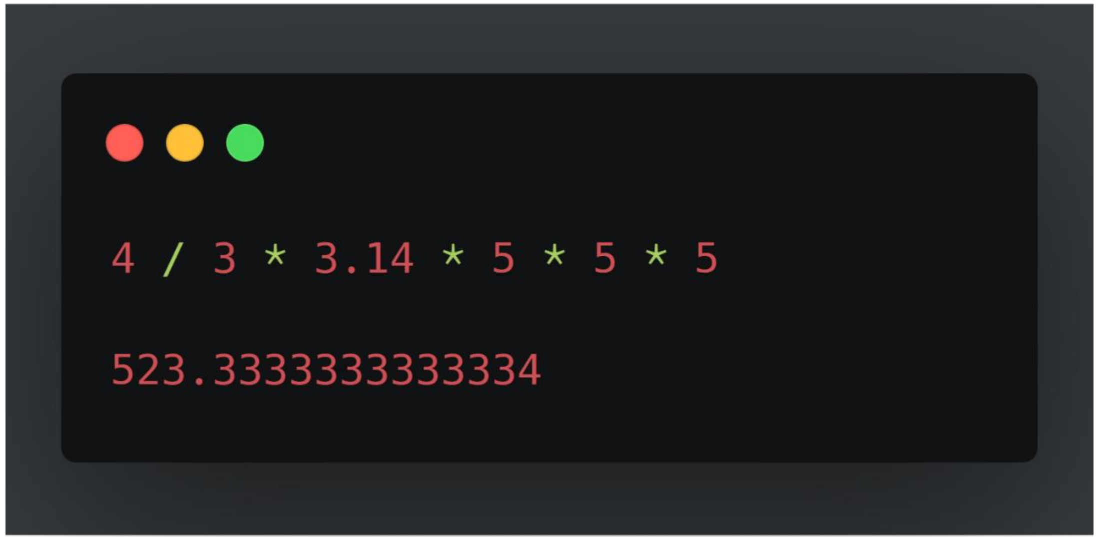
Introduction to

python

Let us start

Agenda

- VARIABLES
- ARITHMETIC OPERATORS
- BOOLEAN LOGIC
- STRING FORMATTING
- PRINTING
- CONDITIONAL STATEMENTS
- LOOPS
- EXERCISES

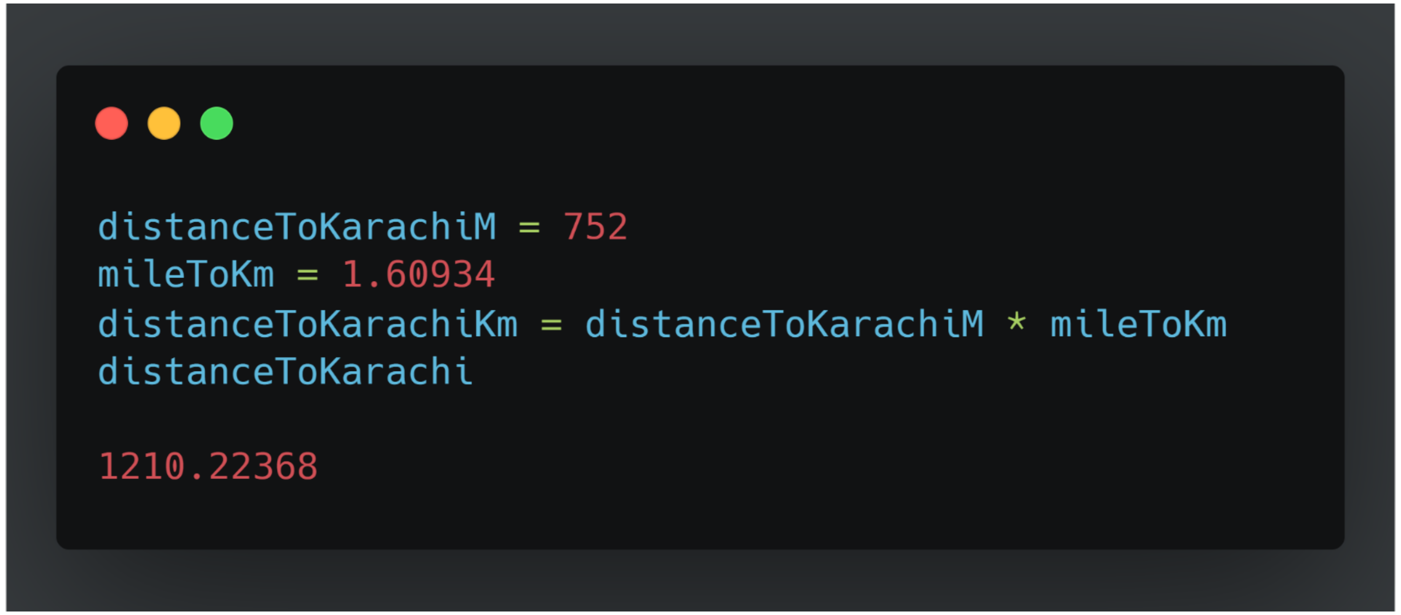


```
4 / 3 * 3.14 * 5 * 5 * 5

523.3333333333334
```

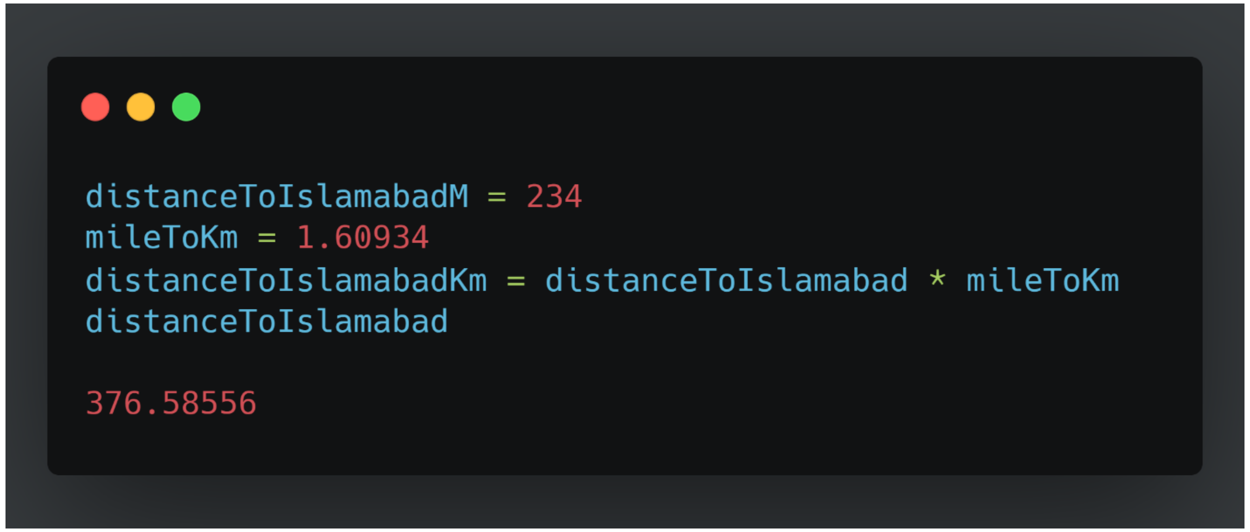
Python as a calculator

Variables

- Great calculator but how can we make it store values?
- Do this by defining variables
- Can later be called by the variable name
- Variable names are case sensitive and unique



```
distanceToKarachiM = 752
mileToKm = 1.60934
distanceToKarachiKm = distanceToKarachiM * mileToKm
distanceToKarachi

1210.22368
```

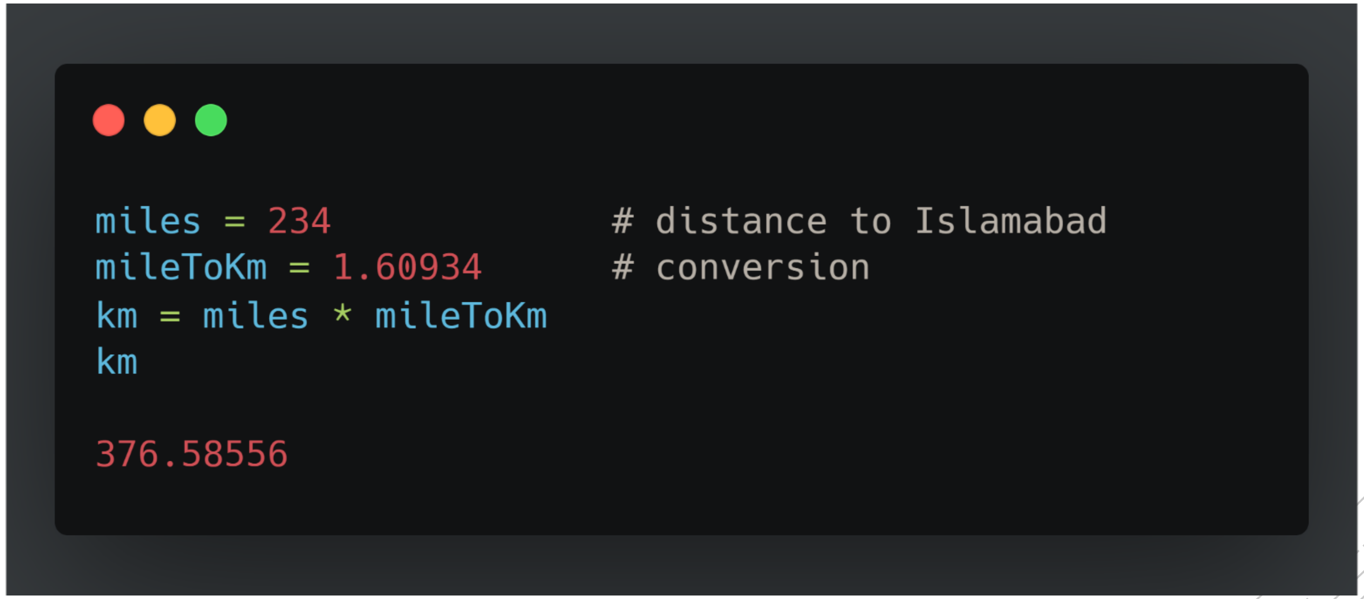


```
distanceToIslamabadM = 234
mileToKm = 1.60934
distanceToIslamabadKm = distanceToIslamabad * mileToKm
distanceToIslamabad

376.58556
```

We can now reuse the variable mileToKm in the next lines without having to define it again!

Commenting

- Useful when your code needs further explanation. Either for your future self and anybody else.
- Useful when you want to remove the code from execution but not permanently
- Comments in Python are done with #

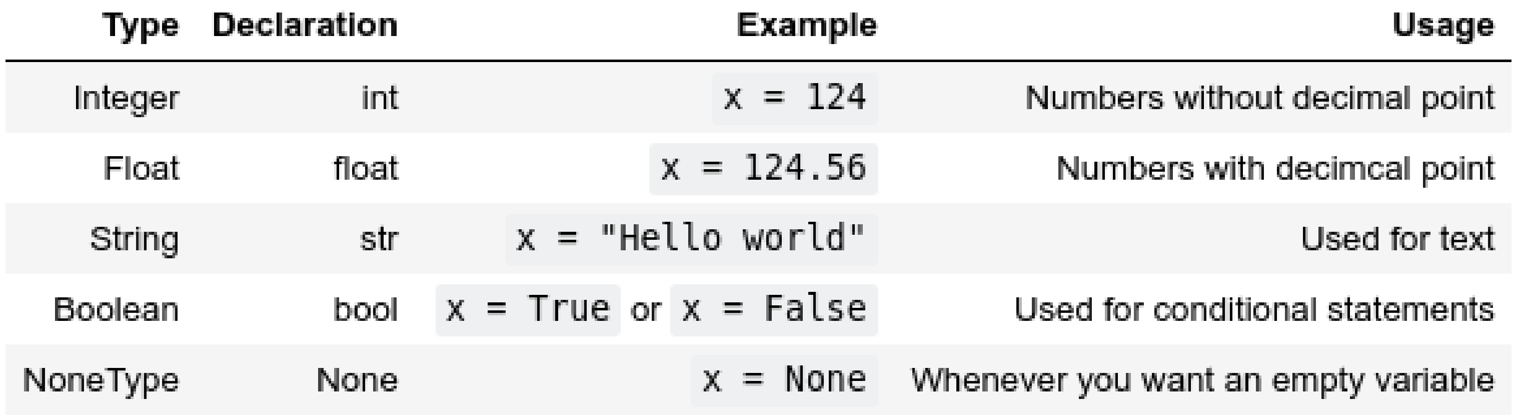


```
miles = 234                # distance to Islamabad
mileToKm = 1.60934         # conversion
km = miles * mileToKm
km

376.58556
```

Types

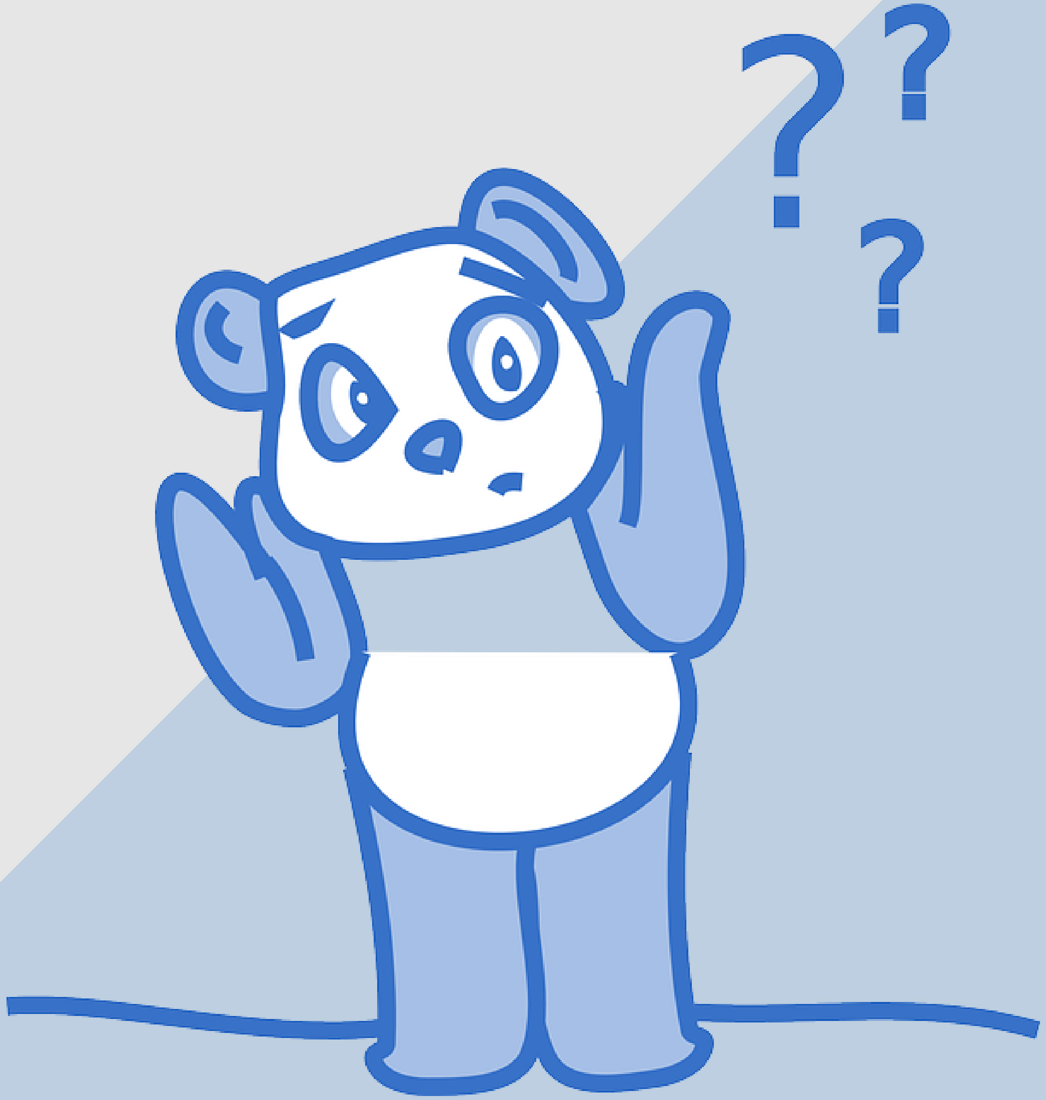
Variables actually have a type, which defines the way it is stored.

The basic types are:



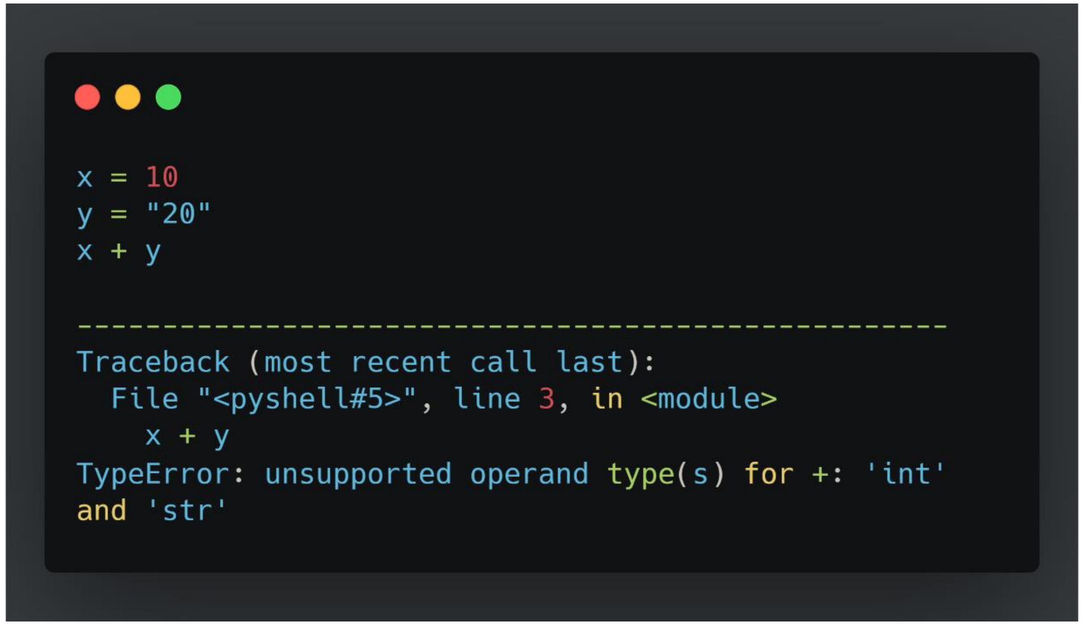
Type	Declaration	Example	Usage
Integer	int	`x = 124`	Numbers without decimal point
Float	float	`x = 124.56`	Numbers with decimcal point
String	str	`x = "Hello world"`	Used for text
Boolean	bool	`x = True` or `x = False`	Used for conditional statements
NoneType	None	`x = None`	Whenever you want an empty variable

Why should we care?

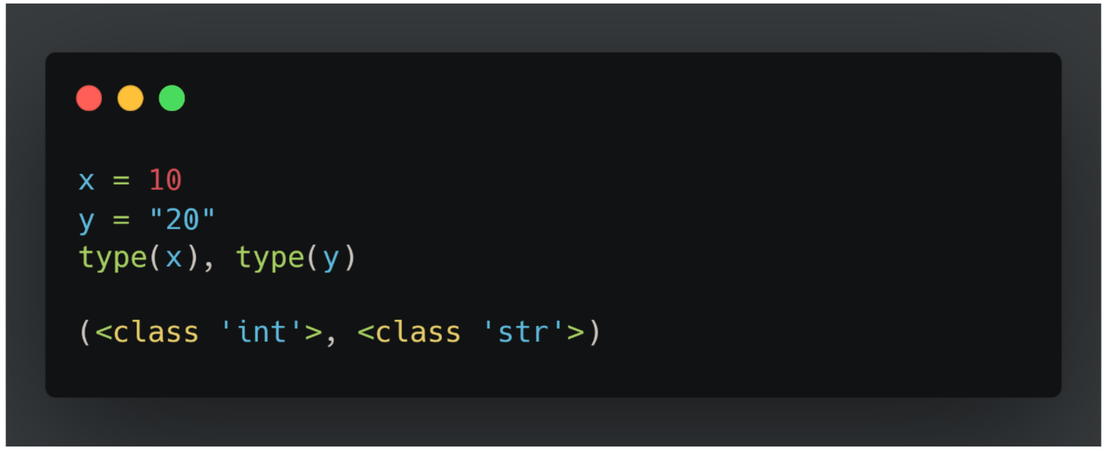


- **Important lesson to remember!**
- We can't do arithmetic operations on variables of different types. Therefore make sure that you are always aware of your variables types!

- You can find the type of a variable using **type()**. For example type **type(x)**.



```
x = 10
y = "20"
x + y

---------------------------------------------------
Traceback (most recent call last):
  File "<pyshell#5>", line 3, in <module>
    x + y
TypeError: unsupported operand type(s) for +: 'int'
and 'str'
```

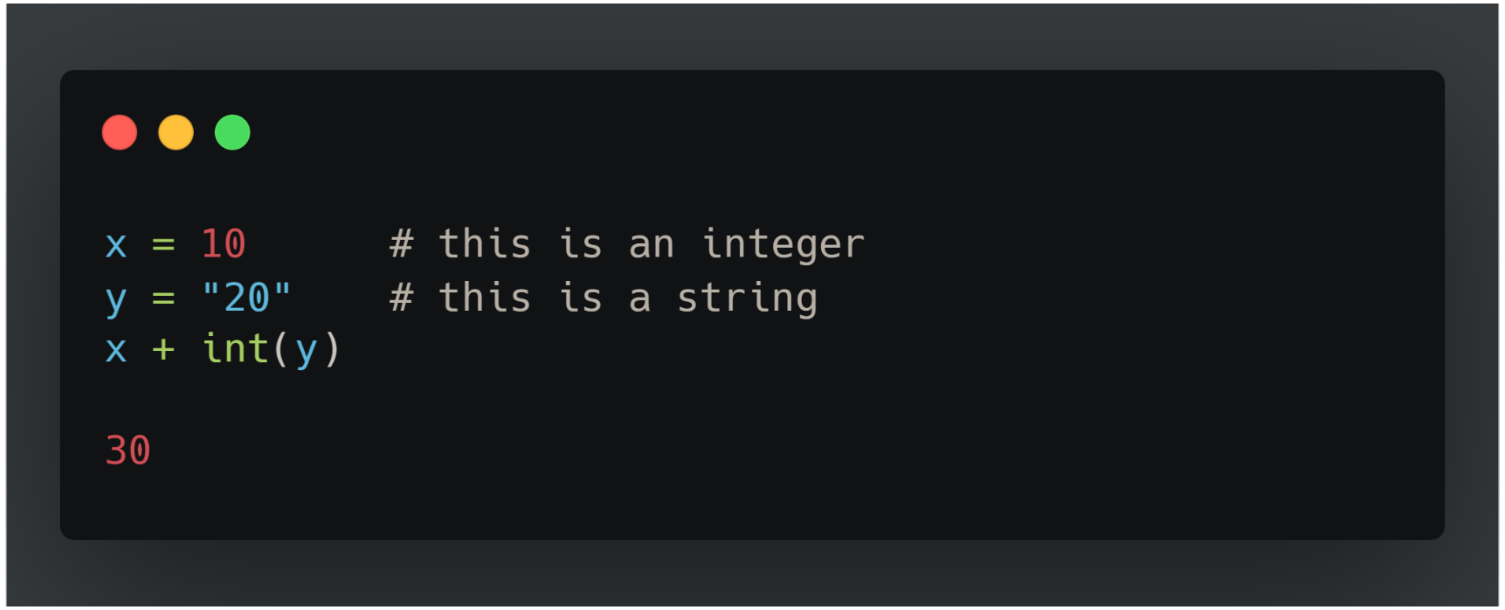


```
x = 10
y = "20"
type(x), type(y)

(<class 'int'>, <class 'str'>)
```

Casting types

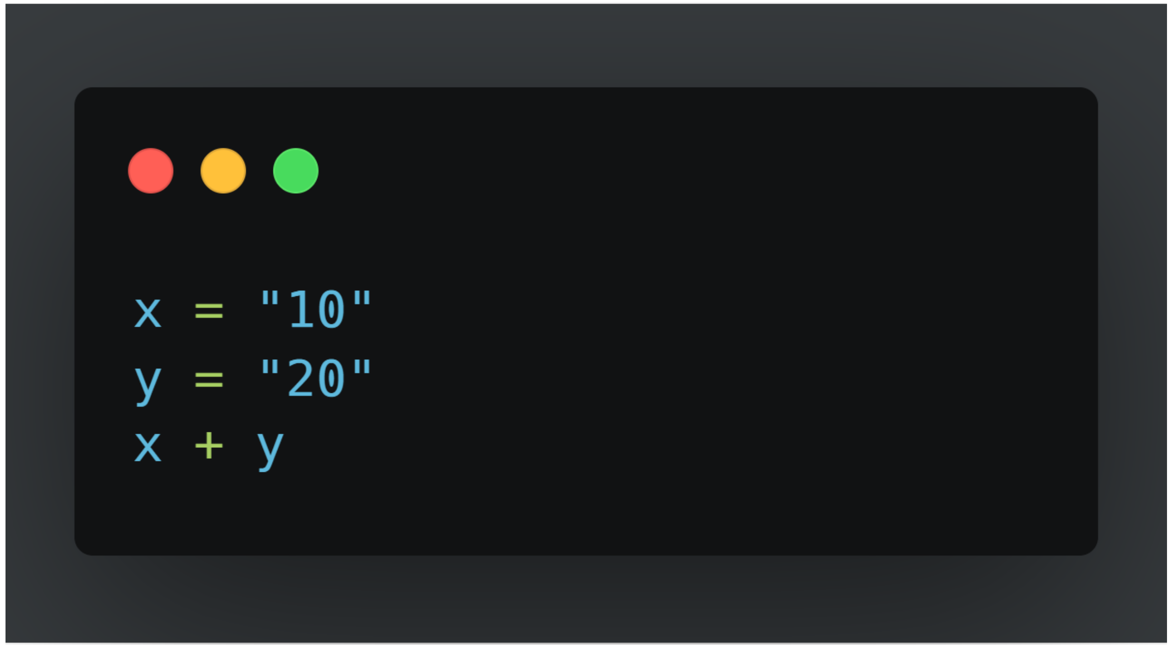
Luckily Python offers us a way of converting variables to different types!

Casting – the operation of converting a variable to a different type



```
x = 10      # this is an integer
y = "20"    # this is a string
x + int(y)

30
```

Similar methods exist for other data types: **int(), float(), str()**



```
x = "10"
y = "20"
x + y
```

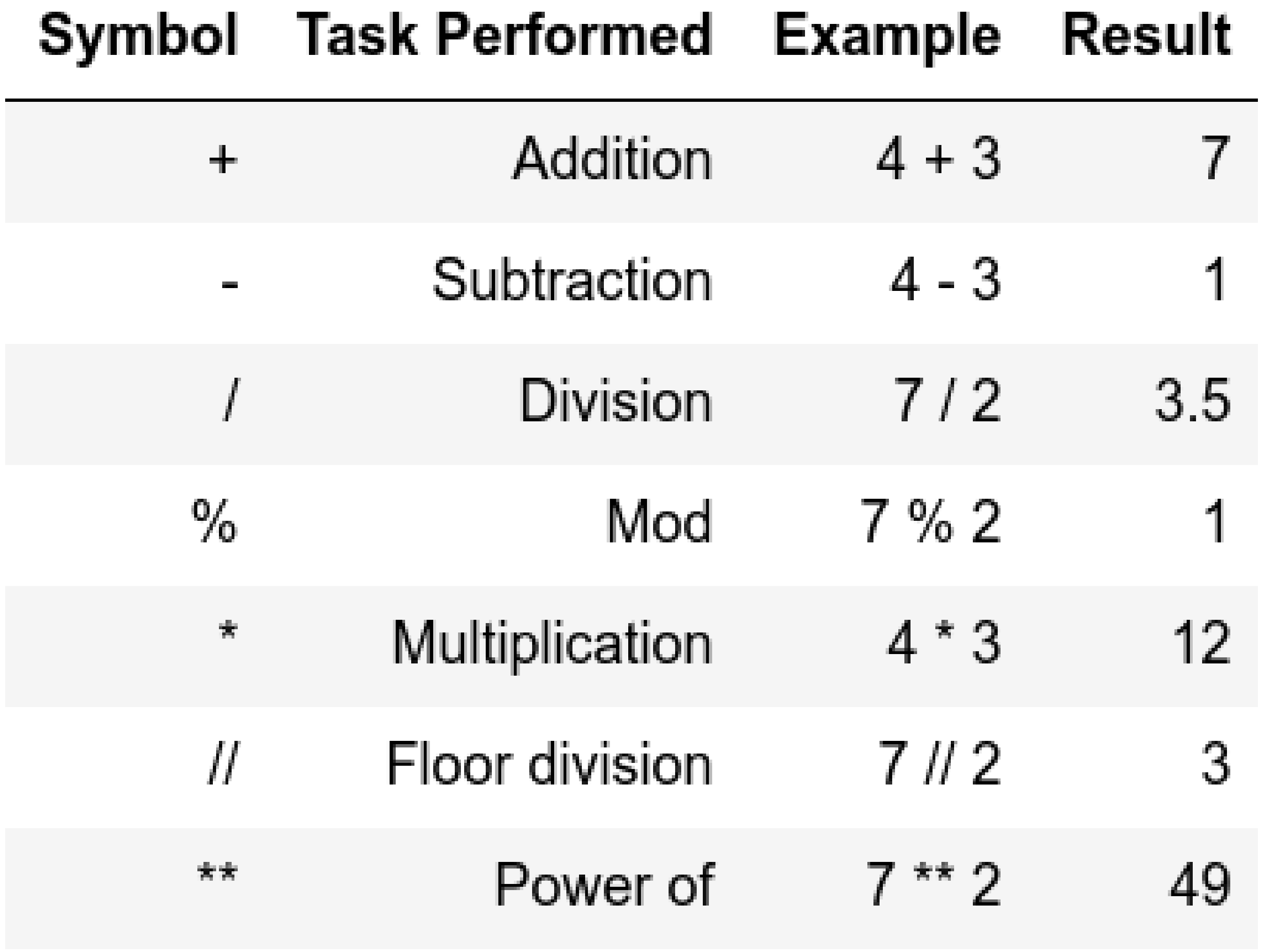
Quick Quiz

Arithmetic operations

Similar to actual Mathematics.

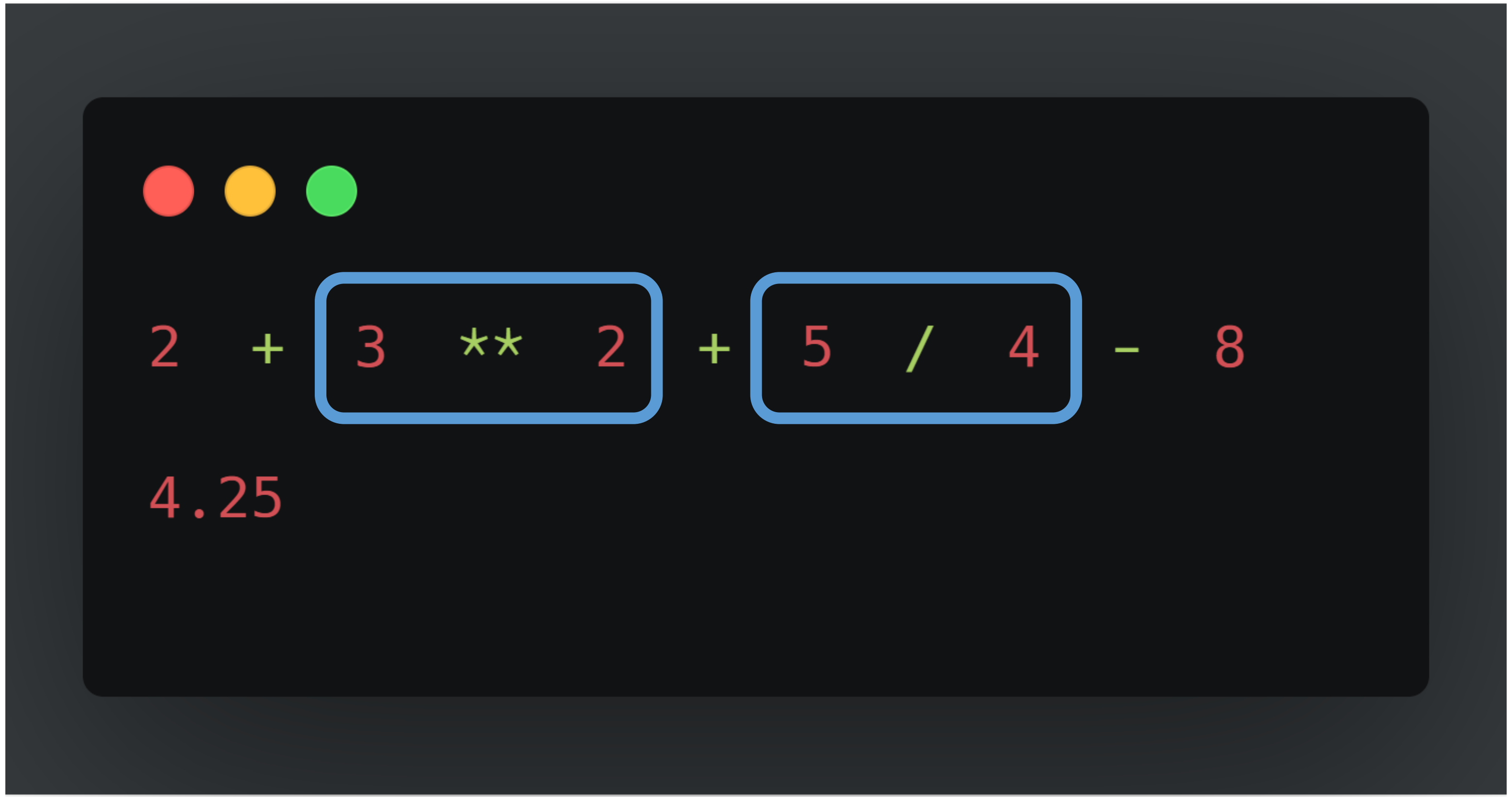
Order of precedence is the same as in Mathematics.

We can also use parenthesis ()



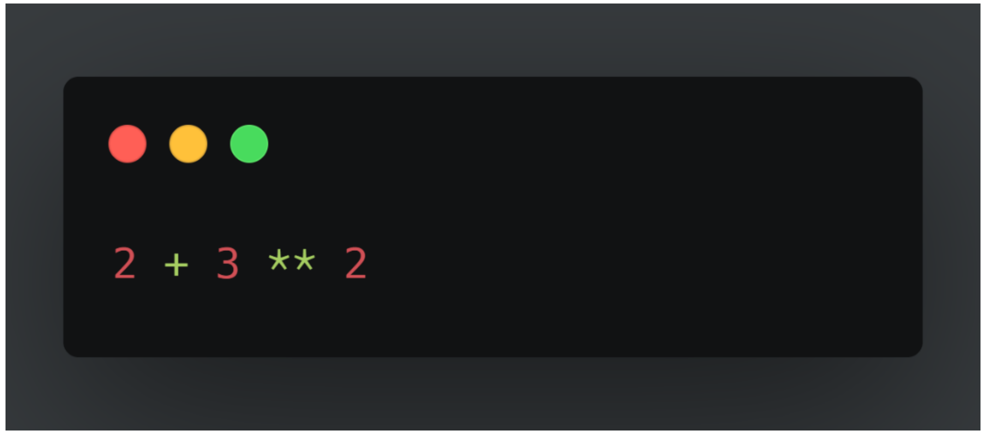
Symbol	Task Performed	Example	Result
+	Addition	4 + 3	7
-	Subtraction	4 - 3	1
/	Division	7 / 2	3.5
%	Mod	7 % 2	1
*	Multiplication	4 * 3	12
//	Floor division	7 // 2	3
**	Power of	7 ** 2	49

Order precedence example

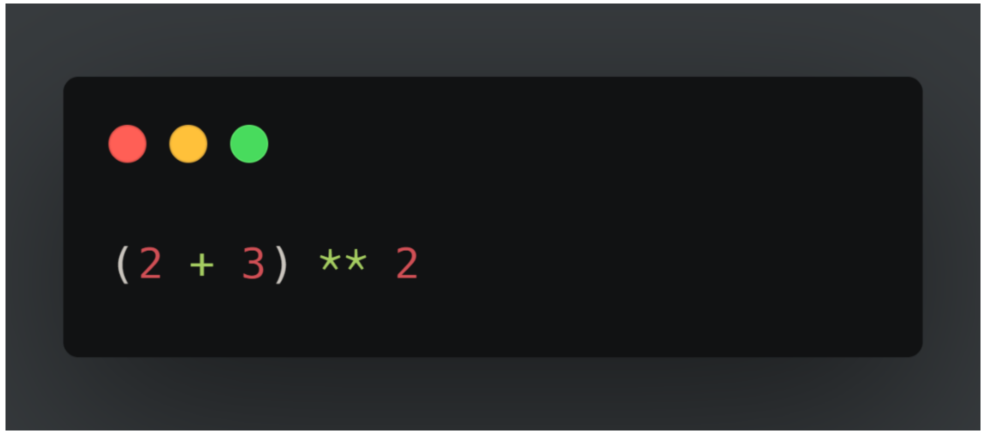


```
2  +  3  **  2  +  5  /  4  -  8

4.25
```

Quick Quiz



```
2 + 3 ** 2
```



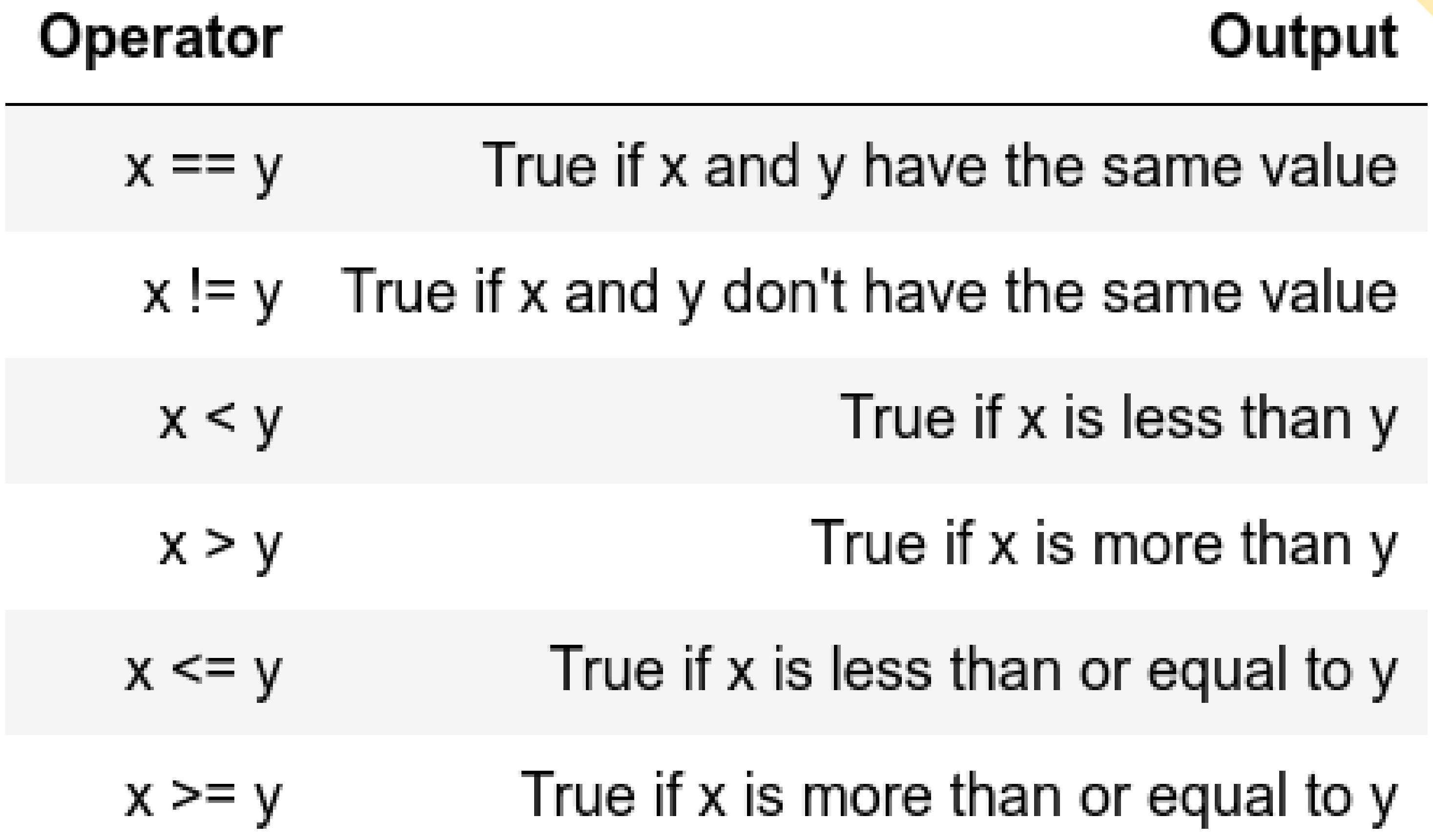
```
(2 + 3) ** 2
```

Comparison operators

- Return Boolean values

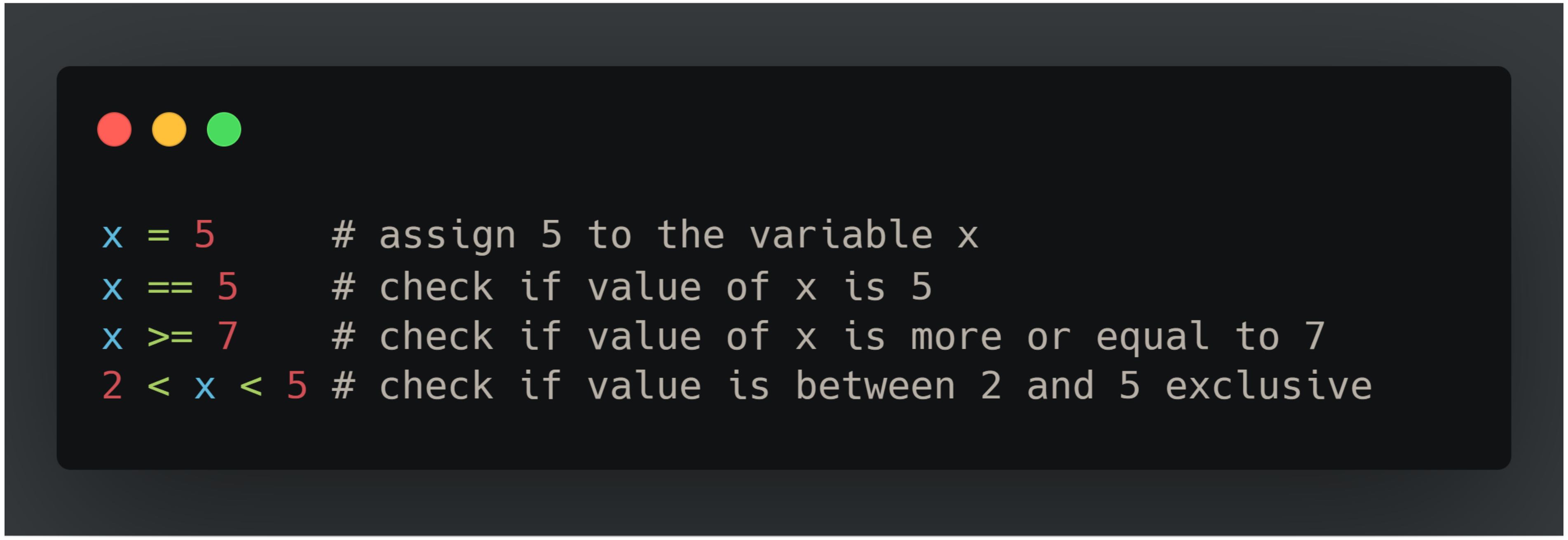
(i.e. True or False)

- Used extensively for conditional statements



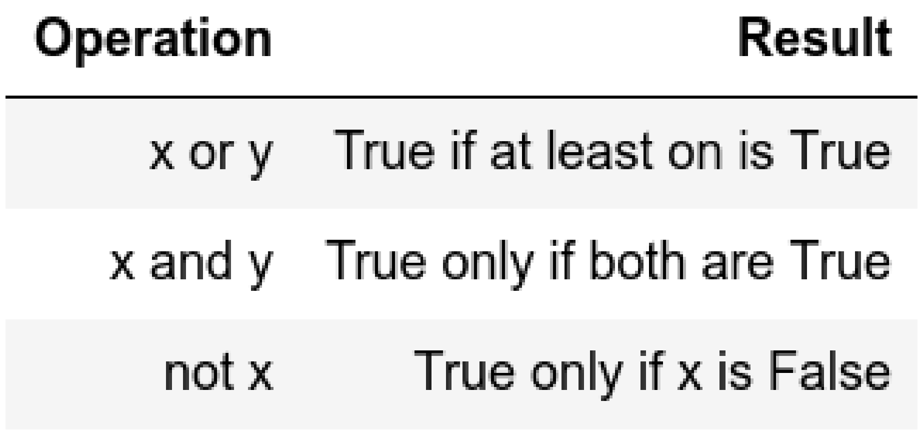
Operator	Output
x == y	True if x and y have the same value
x != y	True if x and y don't have the same value
x < y	True if x is less than y
x > y	True if x is more than y
x <= y	True if x is less than or equal to y
x >= y	True if x is more than or equal to y

Comparison examples



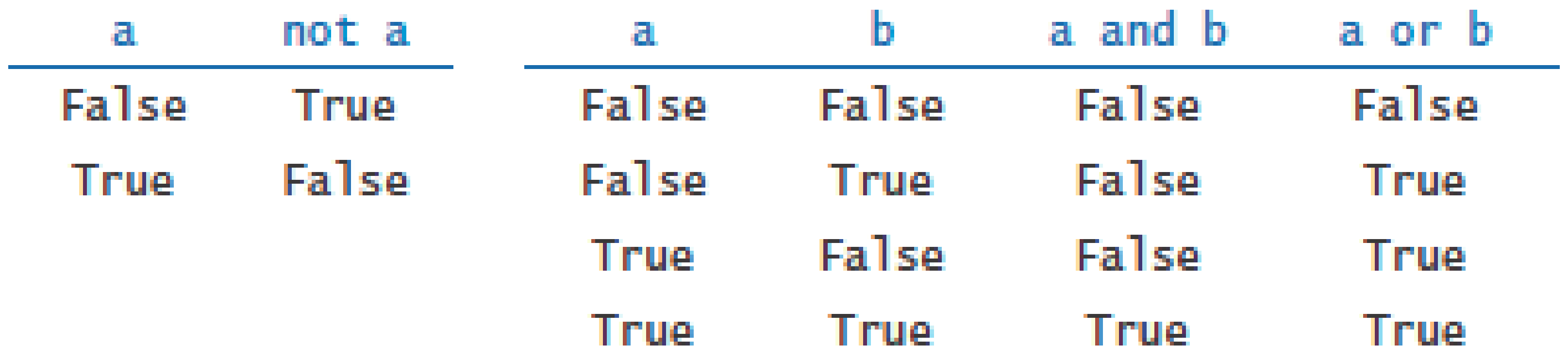
```
x = 5     # assign 5 to the variable x
x == 5    # check if value of x is 5
x >= 7    # check if value of x is more or equal to 7
2 < x < 5 # check if value is between 2 and 5 exclusive
```

Logical operators



Operation	Result
x or y	True if at least on is True
x and y	True only if both are True
not x	True only if x is False

- Allows us to extend the conditional logic
- Will become essential later on

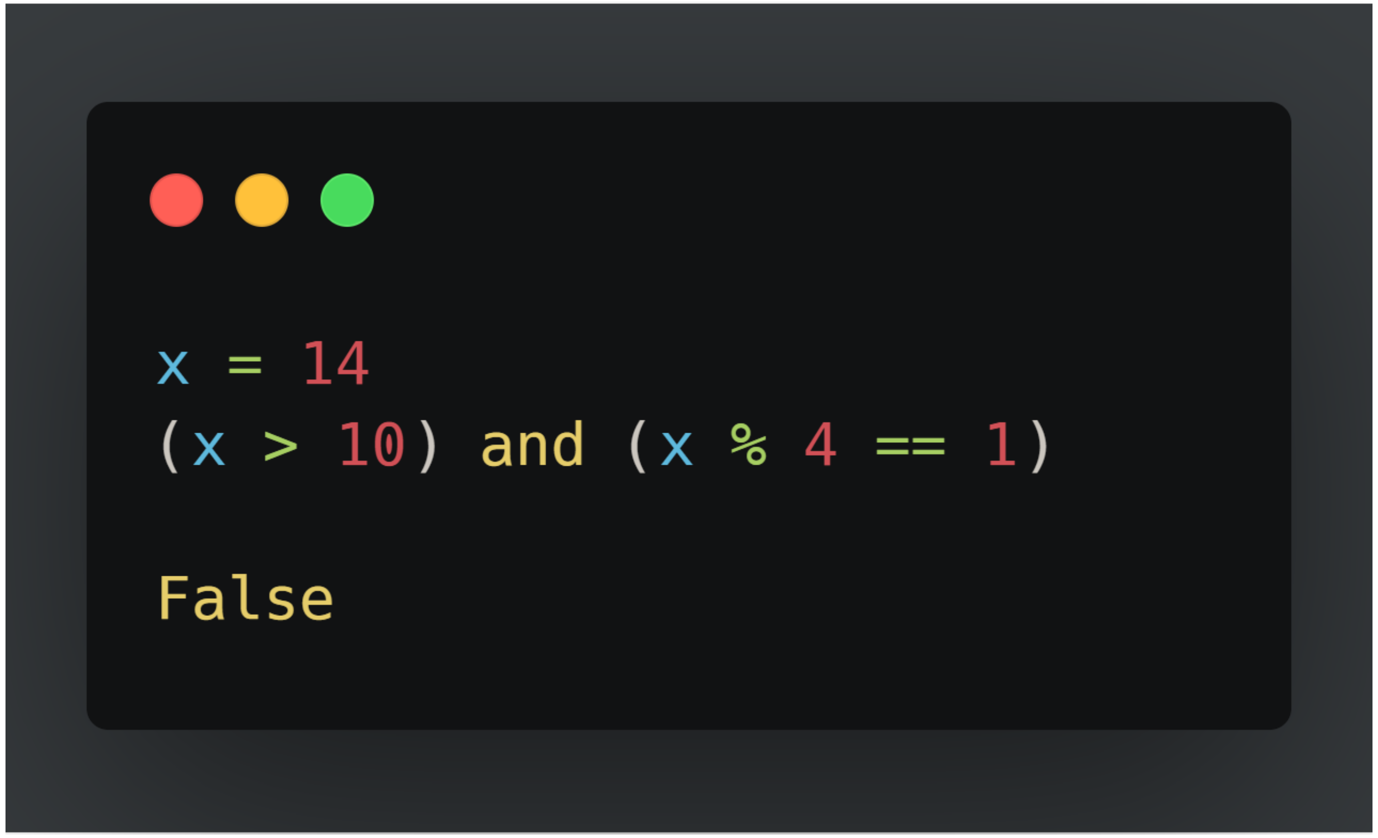


a	not a
False	True
True	False

a	b	a and b	a or b
False	False	False	False
False	True	False	True
True	False	False	True
True	True	True	True

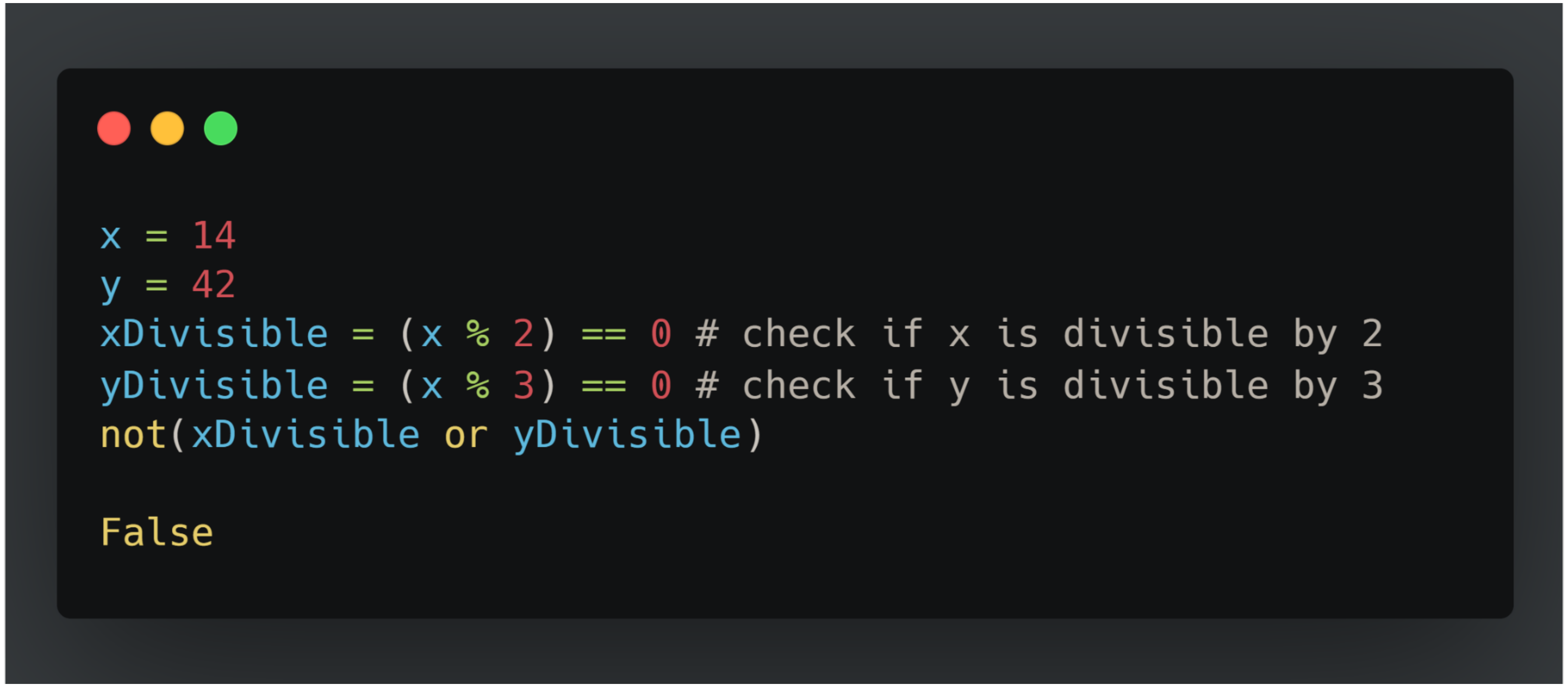
Truth-table definitions of `bool` *operations*

Combining both



```
x = 14
(x > 10) and (x % 4 == 1)

False
```

Another Example

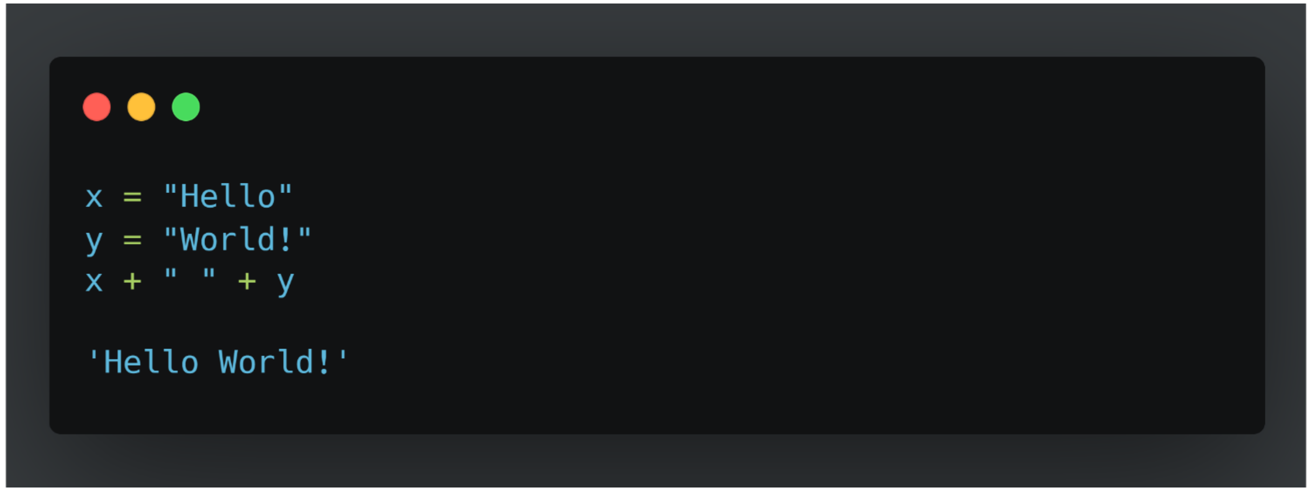


```
x = 14
y = 42
xDivisible = (x % 2) == 0 # check if x is divisible by 2
yDivisible = (x % 3) == 0 # check if y is divisible by 3
not(xDivisible or yDivisible)

False
```

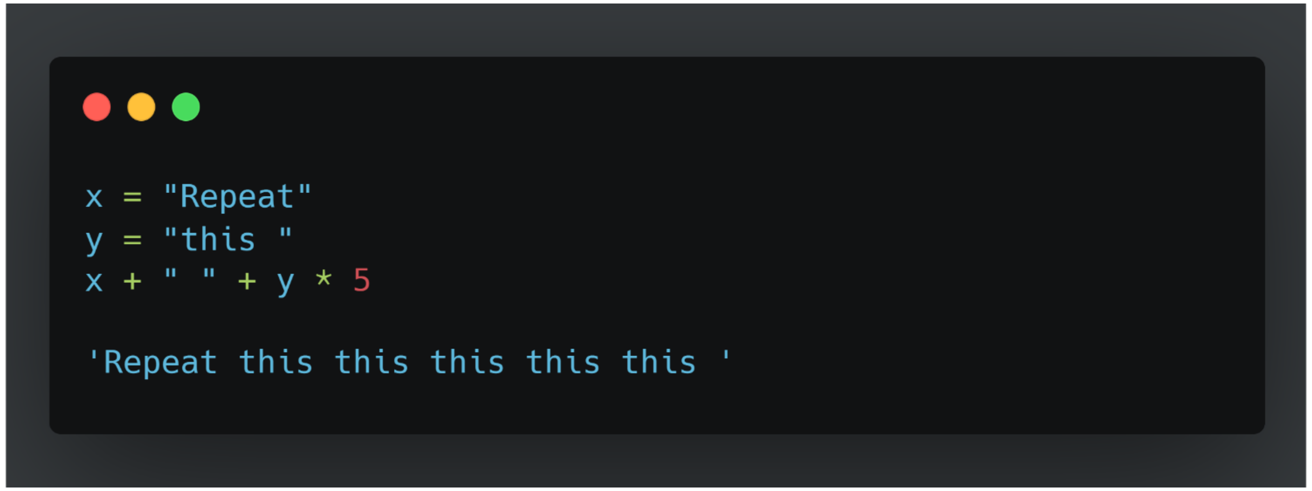
Strings

- Powerful and flexible in Python
- Can be added
- Can be multiplied
- Can be multiple lines

Strings



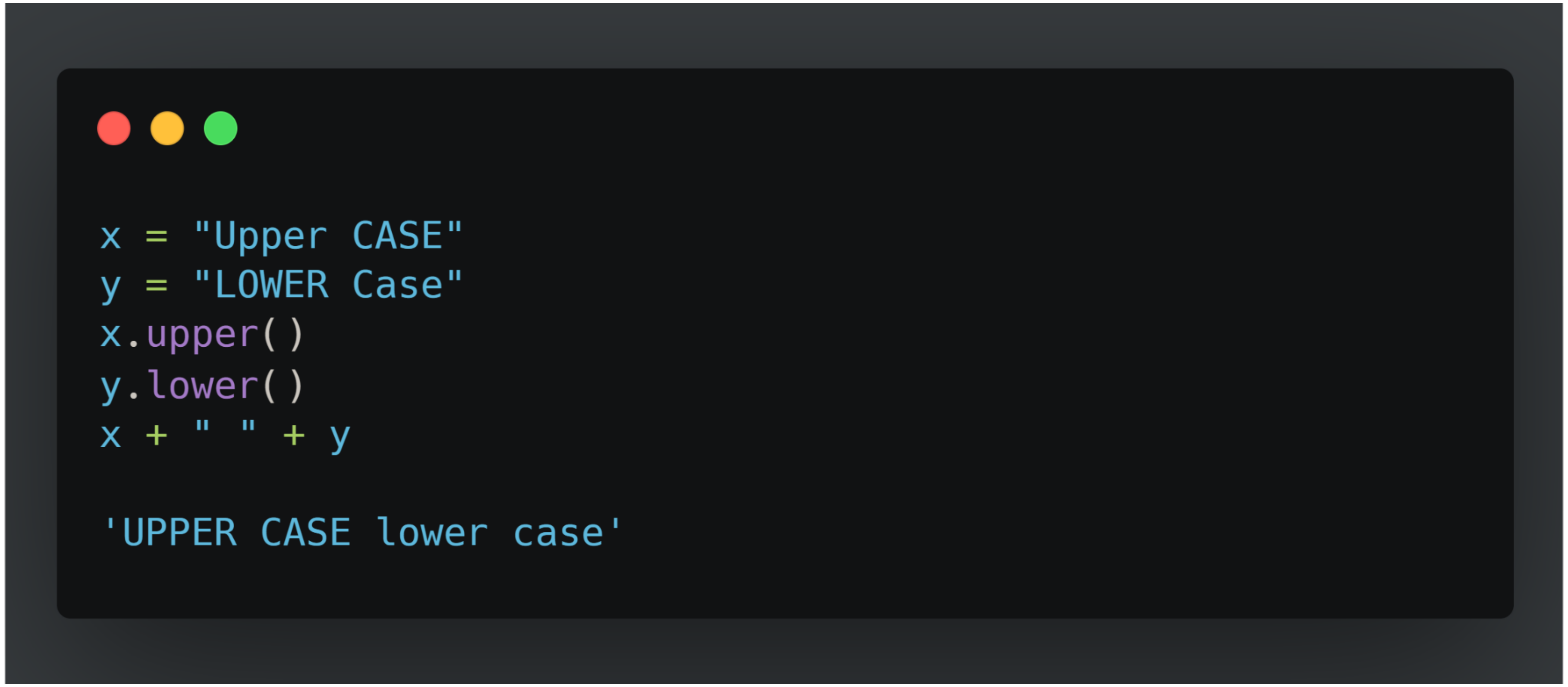
```
x = "Hello"
y = "World!"
x + " " + y

'Hello World!'
```



```
x = "Repeat"
y = "this "
x + " " + y * 5

'Repeat this this this this this '
```

Strings

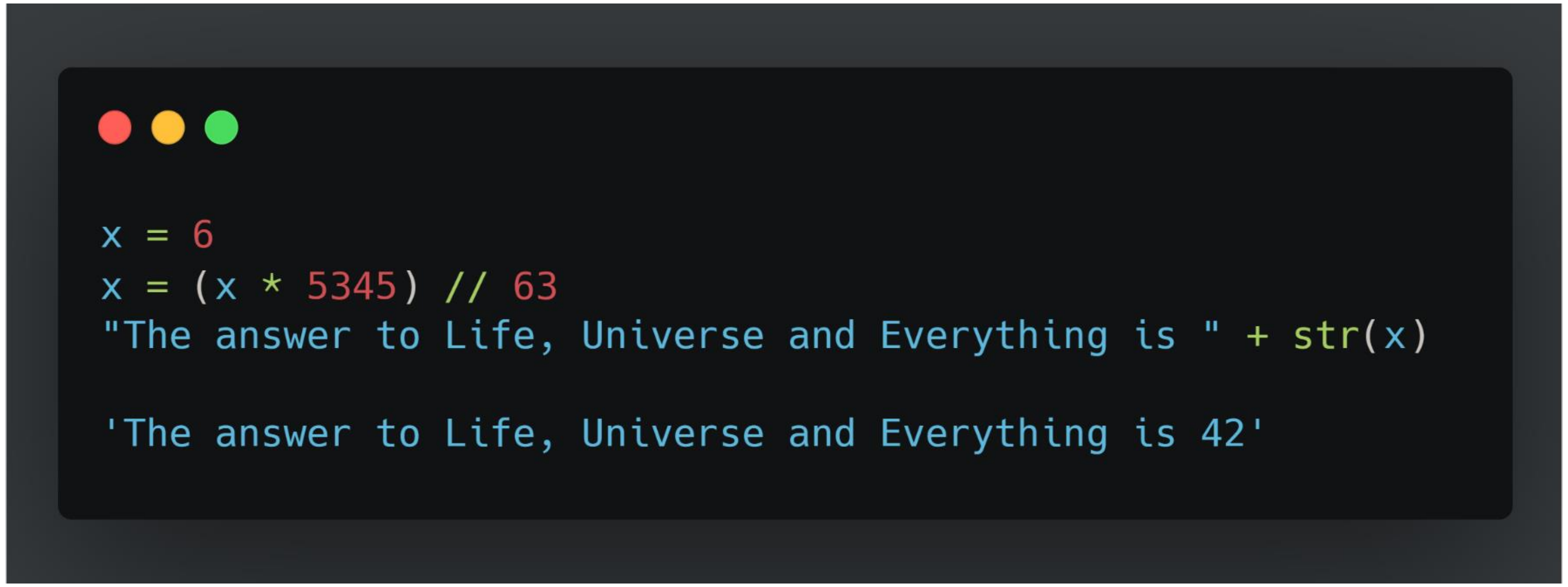
- These are called methods and add extra functionality to the String.
- If you want to see more methods that can be applied to a string simply type in **dir('str')**



```
x = "Upper CASE"
y = "LOWER Case"
x.upper()
y.lower()
x + " " + y

'UPPER CASE lower case'
```

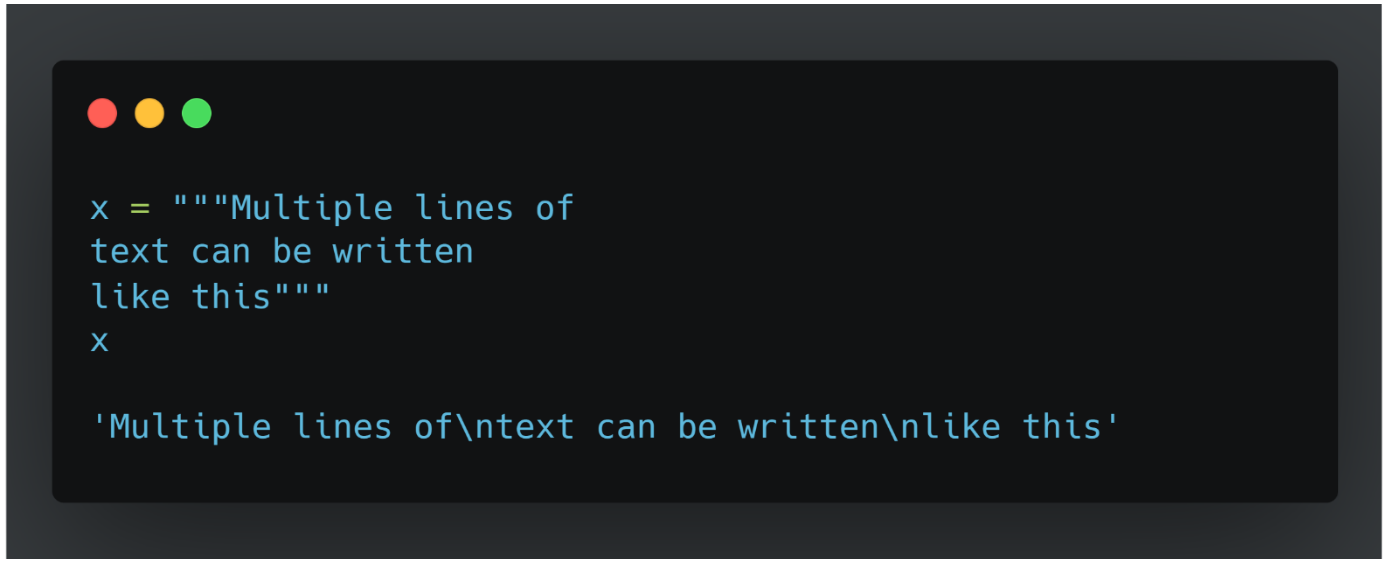
Mixing up strings and numbers

- Often we would need to mix up numbers and strings.
- It is best to keep numbers as numbers (i.e. int or float)
- and cast them to strings whenever we need them as a string.



```
x = 6
x = (x * 5345) // 63
"The answer to Life, Universe and Everything is " + str(x)

'The answer to Life, Universe and Everything is 42'
```

Multiline strings

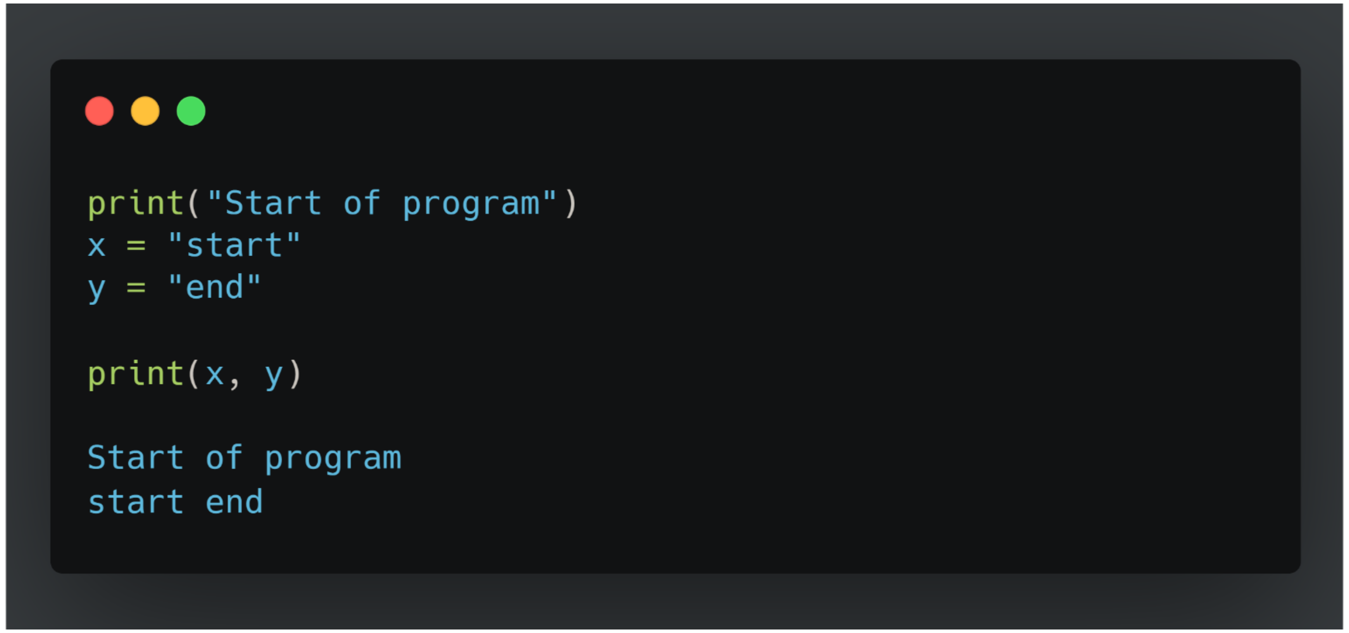


```
x = """Multiple lines of
text can be written
like this"""
x

'Multiple lines of\ntext can be written\nlike this'
```

- ‘\n’ is escape character for new line
- ‘\t’ is escape character for tab space

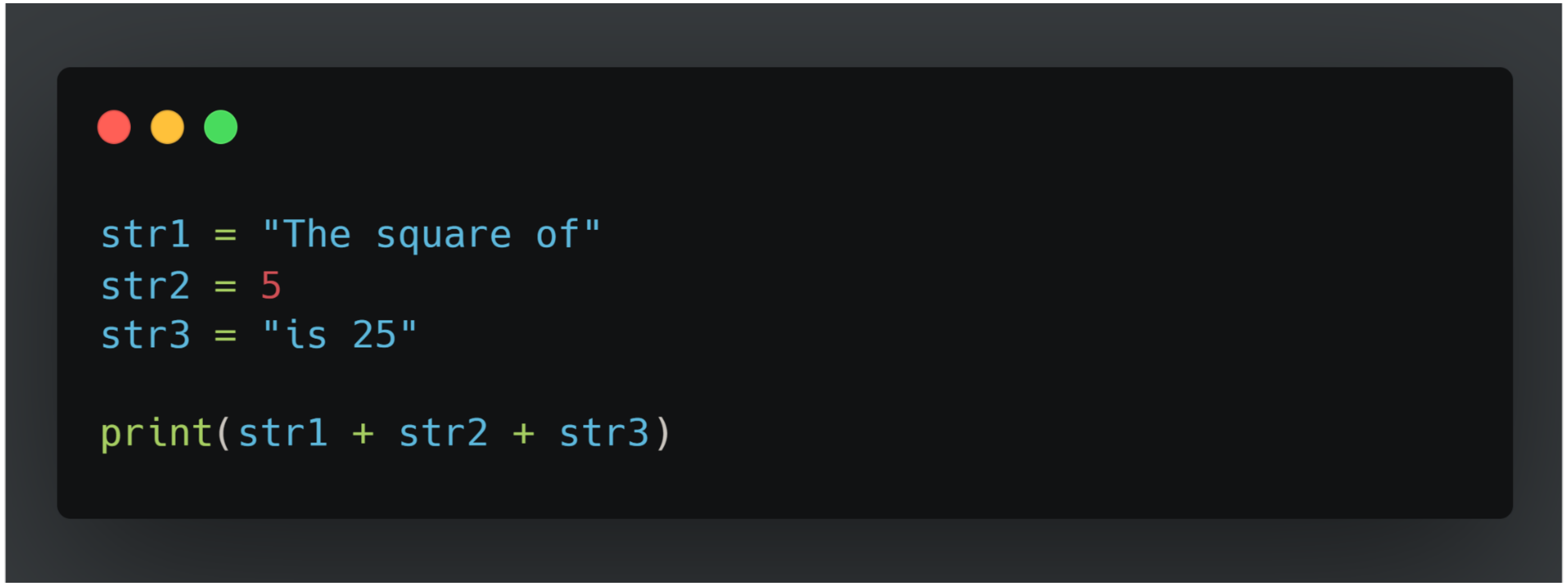
Printing

- When writing scripts, your outcomes aren't printed on the terminal.
- Thus, you must print them yourself with the print() function.
- Beware to not mix up the different type of variables!



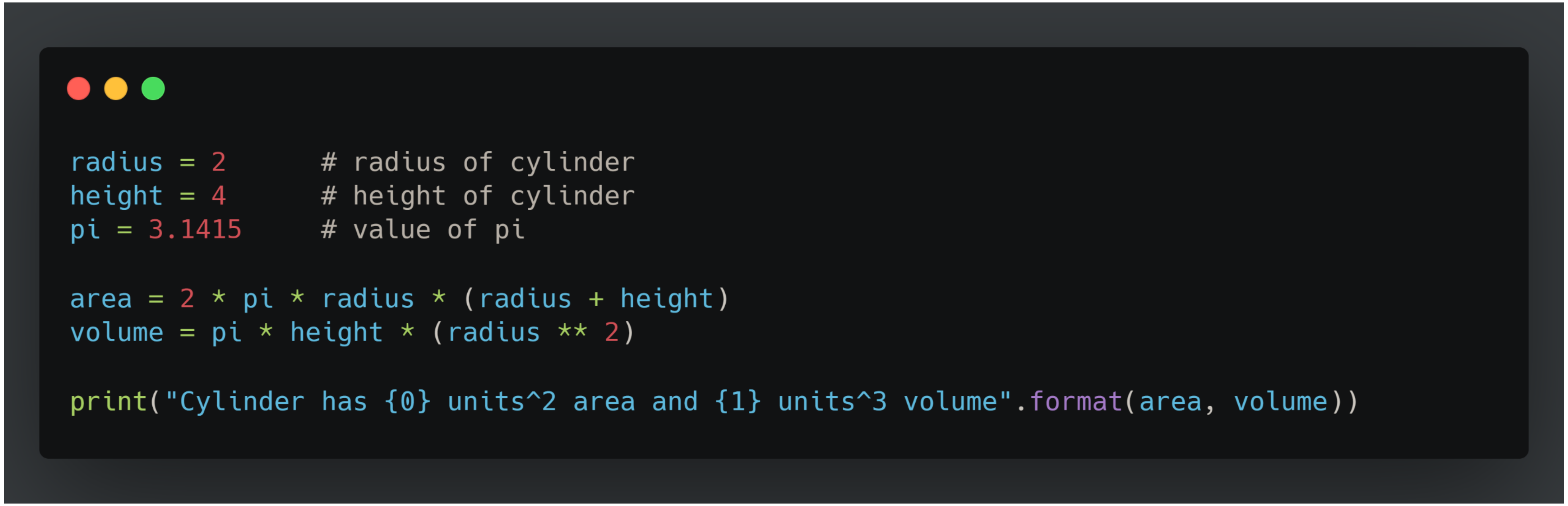
```
print("Start of program")
x = "start"
y = "end"

print(x, y)

Start of program
start end
```

Quick Quiz



```python
str1 = "The square of"
str2 = 5
str3 = "is 25"

print(str1 + str2 + str3)
```

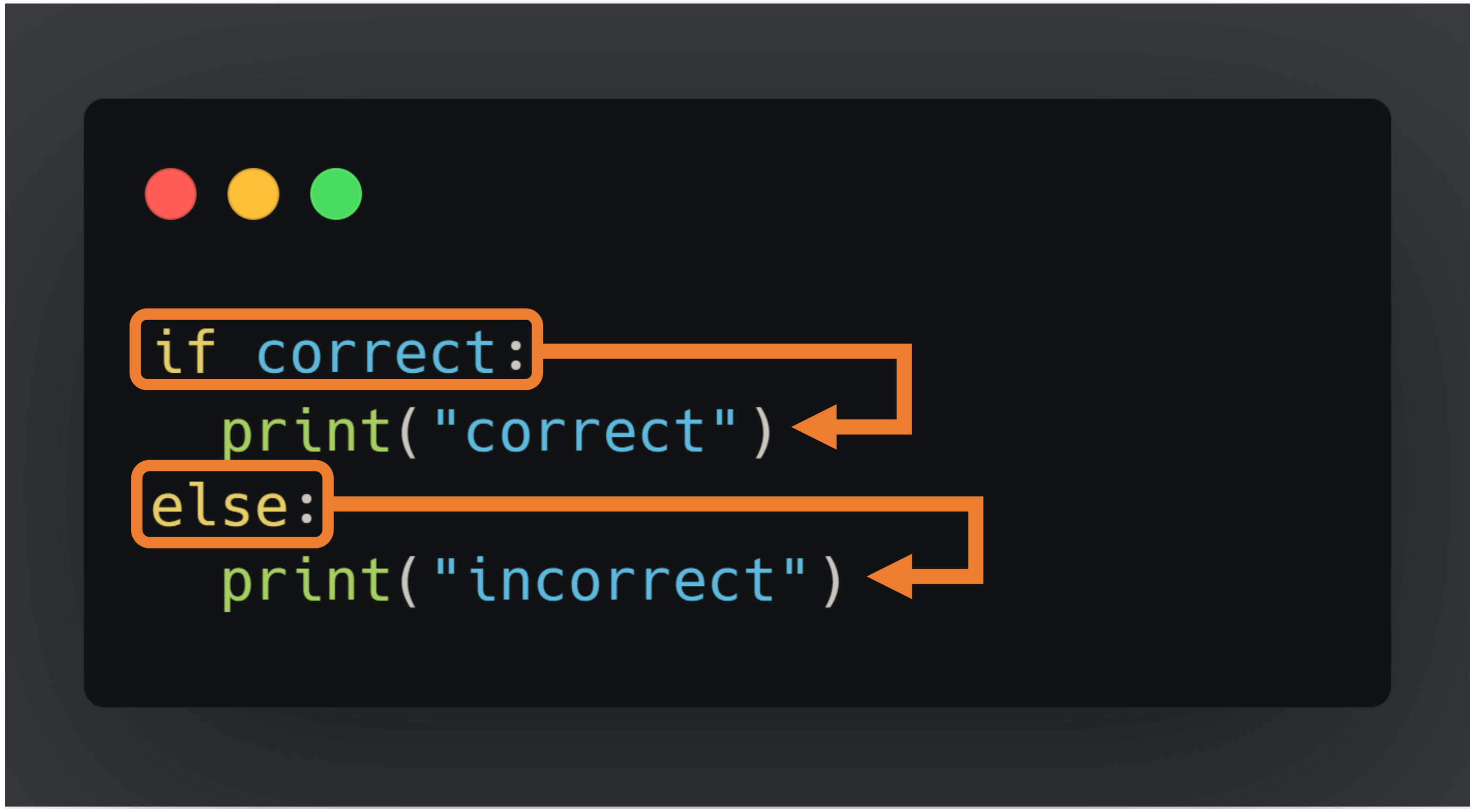
Placeholders



```python
radius = 2       # radius of cylinder
height = 4       # height of cylinder
pi = 3.1415      # value of pi

area = 2 * pi * radius * (radius + height)
volume = pi * height * (radius ** 2)

print("Cylinder has {0} units^2 area and {1} units^3 volume".format(area, volume))
```

If Else

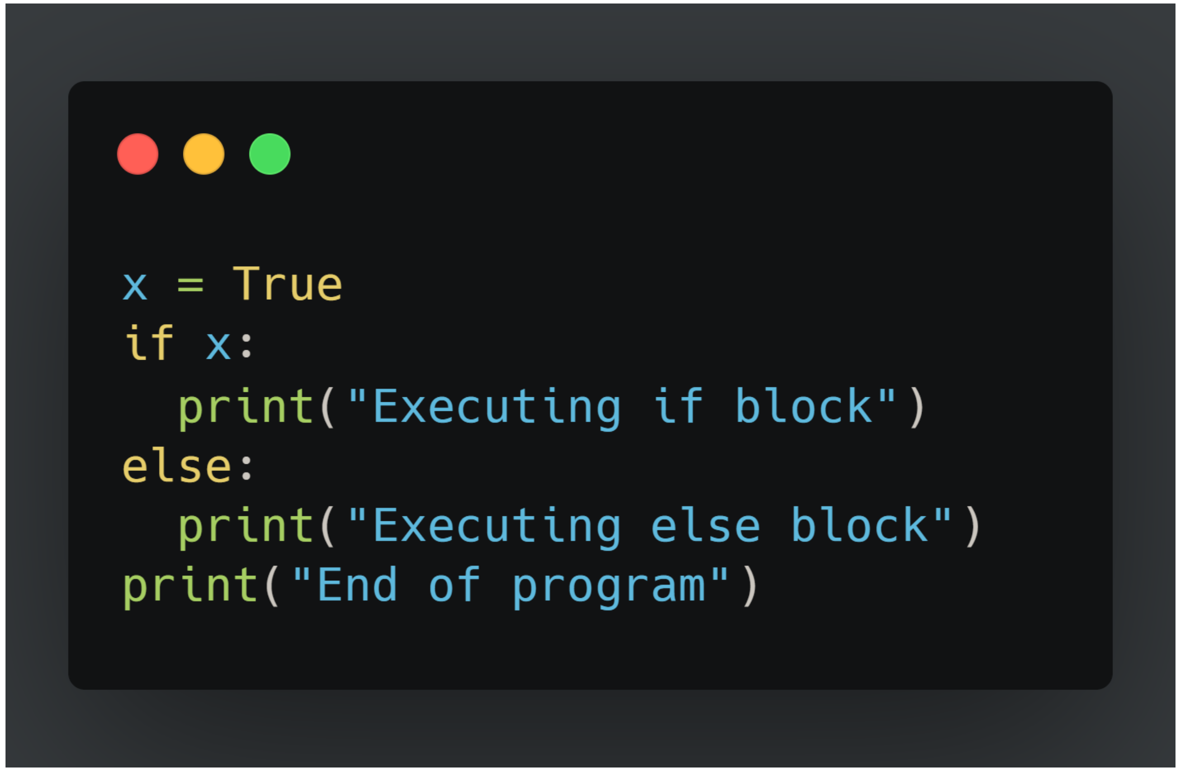
- Fundamental building block of software



```
if correct:
    print("correct")
else:
    print("incorrect")
```

If Else example

Try running the example below.
What do you get?



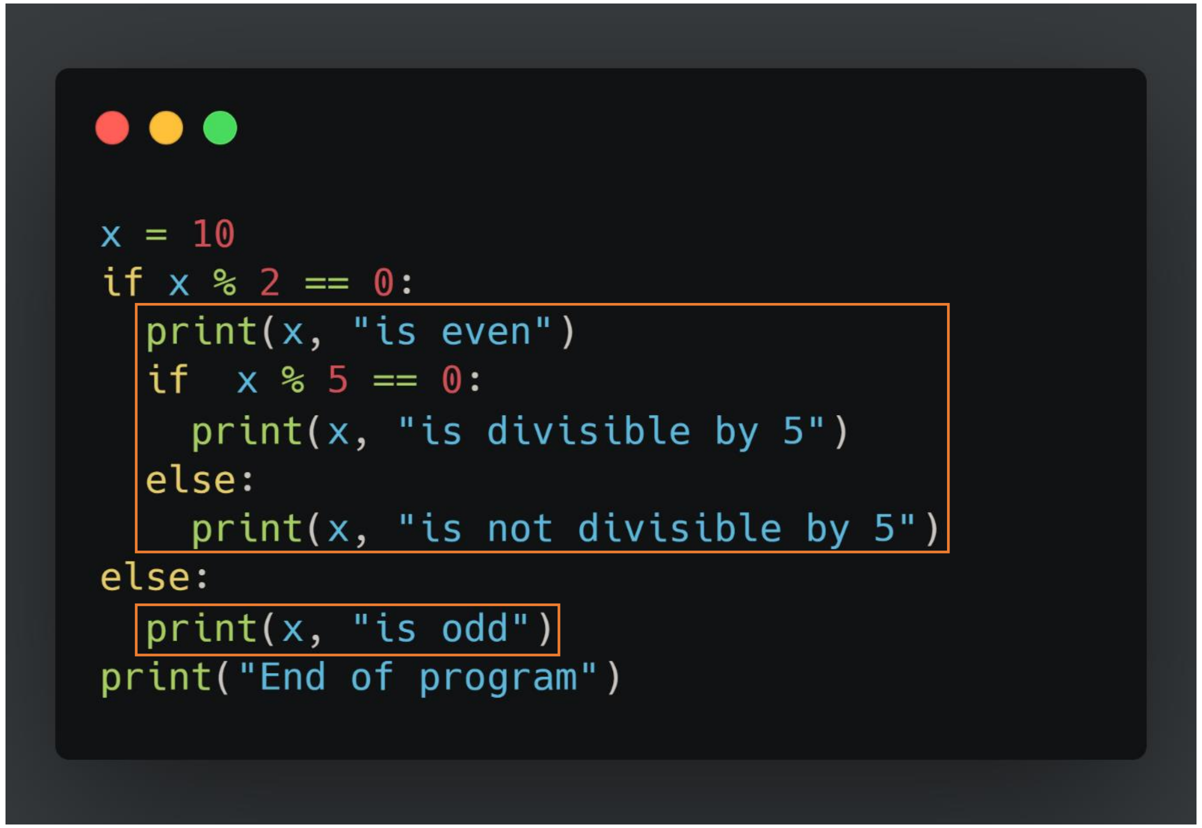
```python
x = True
if x:
  print("Executing if block")
else:
  print("Executing else block")
print("End of program")
```

Indentation matters!

Code is grouped by its indentation

Indentation is the number of whitespace or tab characters before the code.

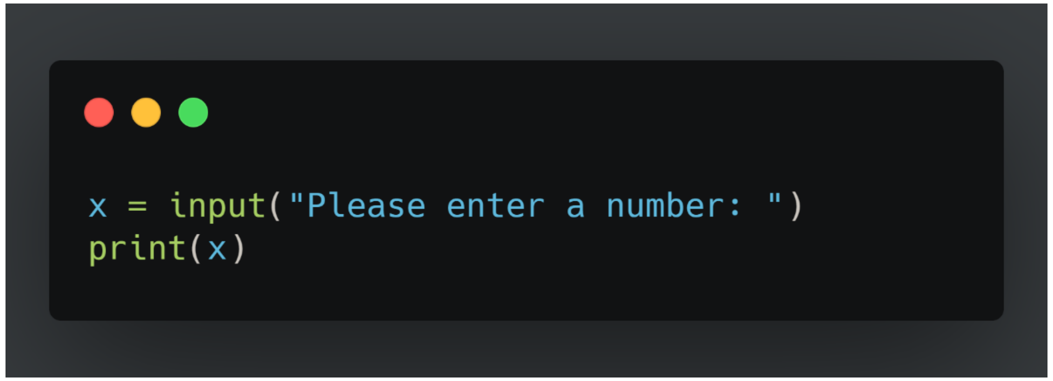
If you put code in the wrong block then you will get unexpected behavior



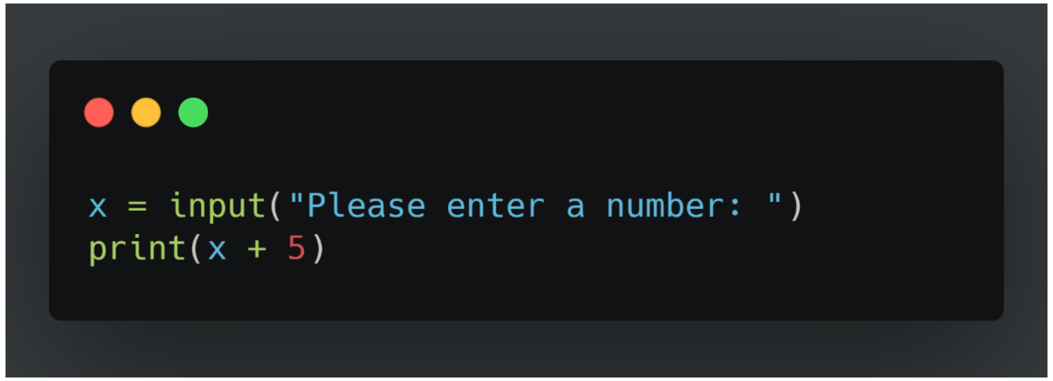
```python
x = 10
if x % 2 == 0:
  print(x, "is even")
  if  x % 5 == 0:
    print(x, "is divisible by 5")
  else:
    print(x, "is not divisible by 5")
else:
  print(x, "is odd")
print("End of program")
```

Input

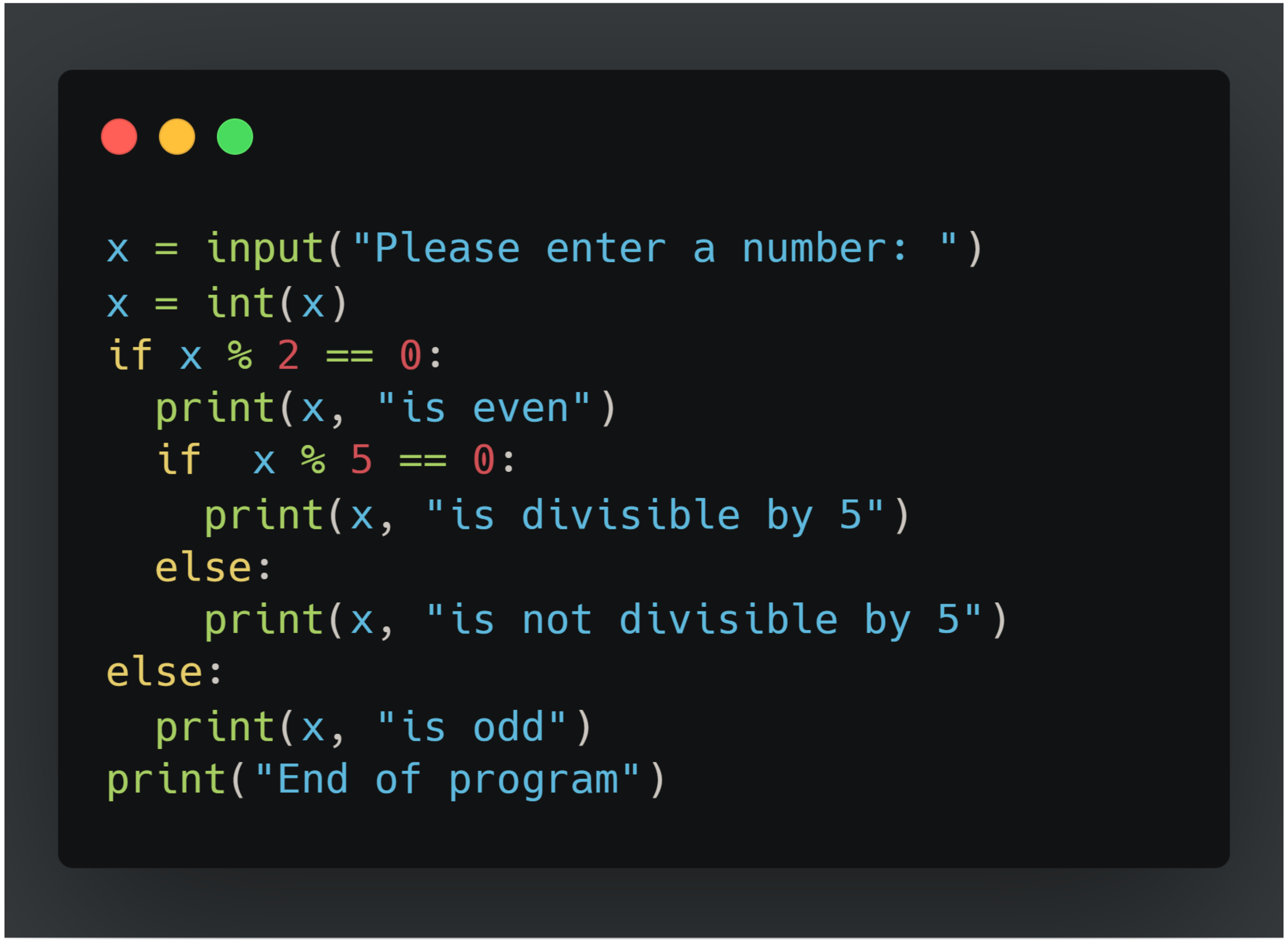
- Most of the programs you write will need to take input from the user
- For this, you must take input from the user using the input() function.
- By default input is in string type



```python
x = input("Please enter a number: ")
print(x)
```



```python
x = input("Please enter a number: ")
print(x + 5)
```

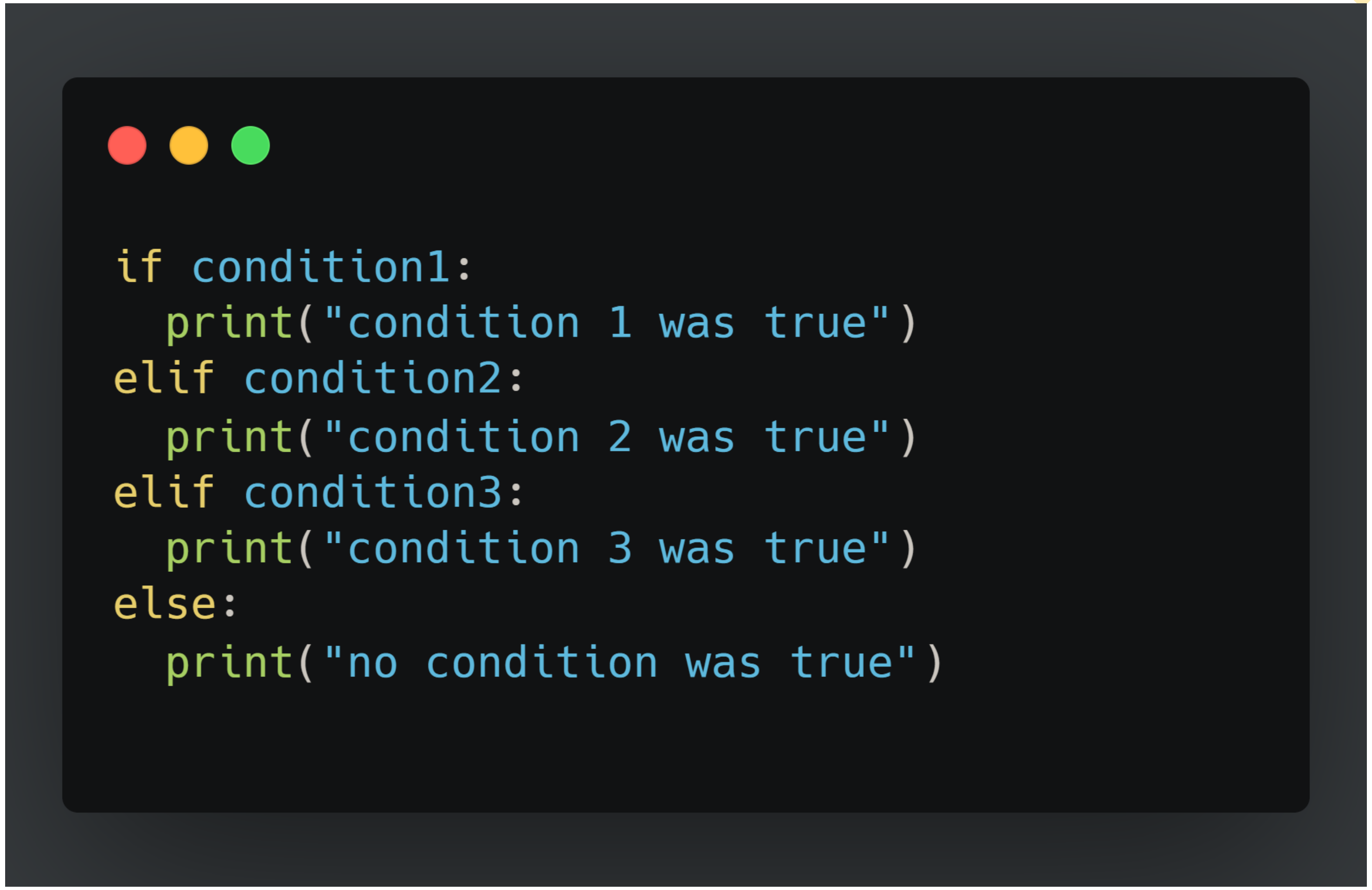


```python
x = input("Please enter a number: ")
x = int(x)
if x % 2 == 0:
  print(x, "is even")
  if  x % 5 == 0:
    print(x, "is divisible by 5")
  else:
    print(x, "is not divisible by 5")
else:
  print(x, "is odd")
print("End of program")
```

Extending if-else blocks

- We can add infinitely more if statements using **elif**

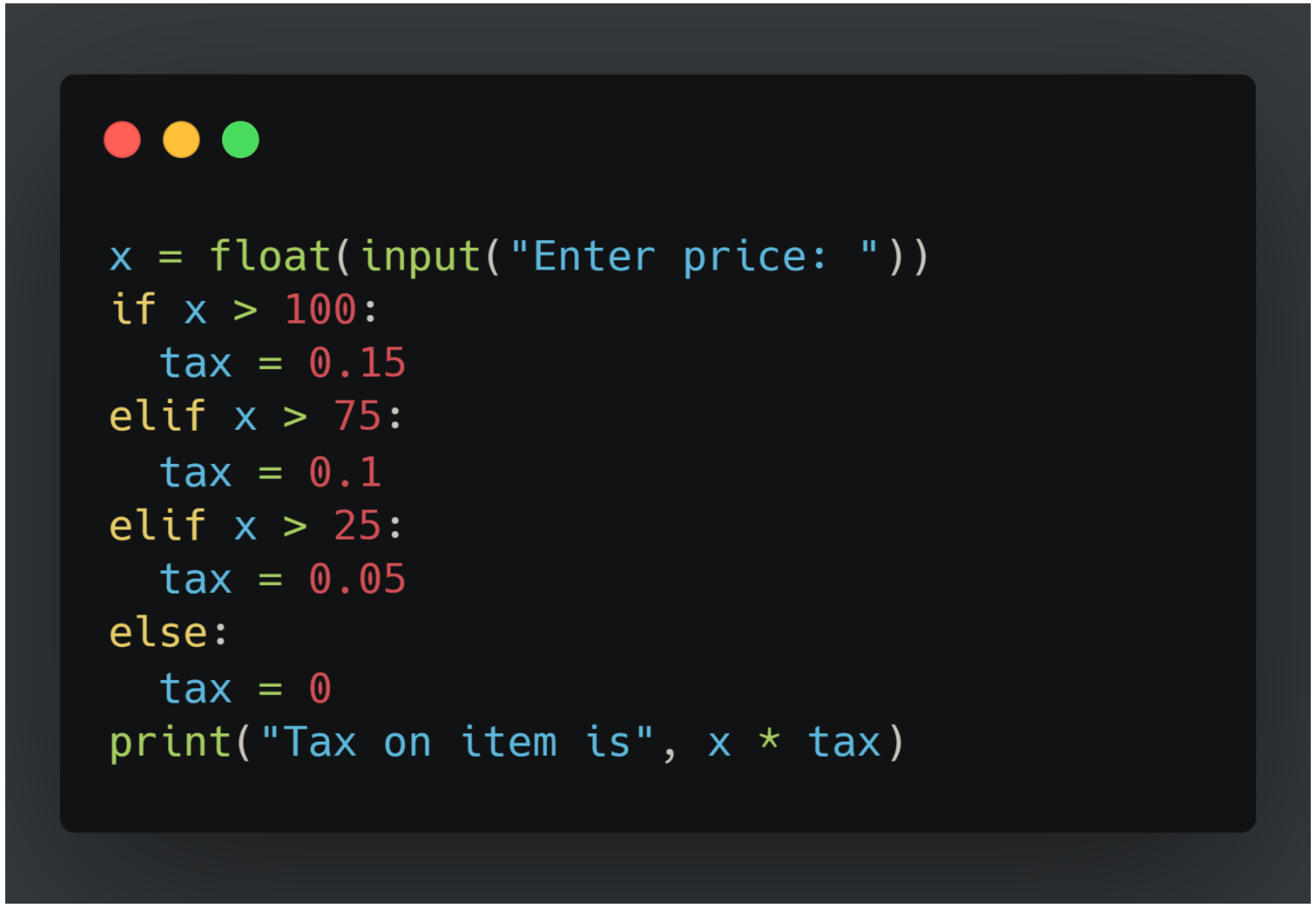
- elif = else + if which means that the previous statements must be false for the current one to evaluate to true



```
if condition1:
  print("condition 1 was true")
elif condition2:
  print("condition 2 was true")
elif condition3:
  print("condition 3 was true")
else:
  print("no condition was true")
```

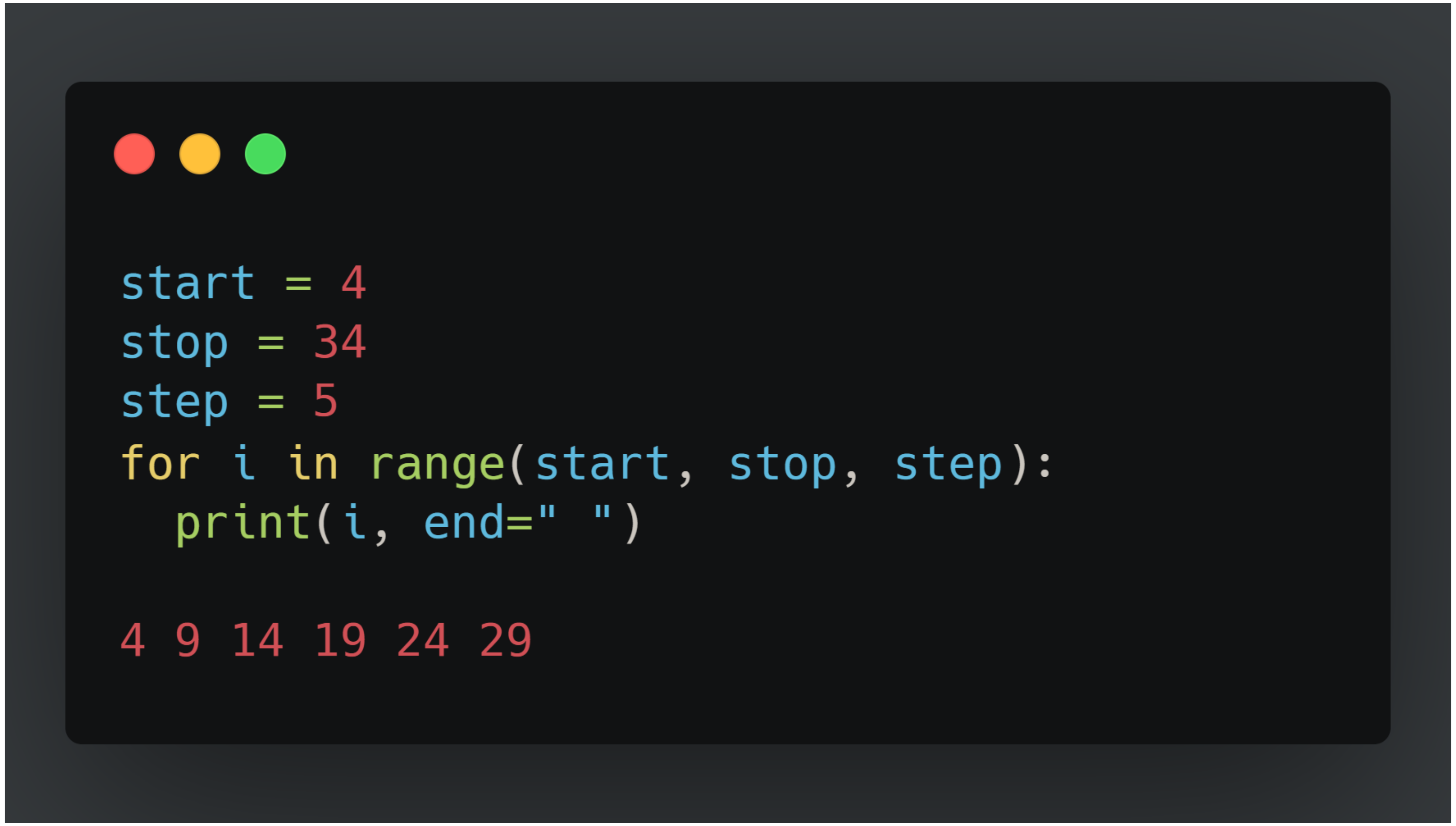
Quick Quiz

- What would happen if both conditions are True?



```
x = float(input("Enter price: "))
if x > 100:
   tax = 0.15
elif x > 75:
   tax = 0.1
elif x > 25:
   tax = 0.05
else:
   tax = 0
print("Tax on item is", x * tax)
```

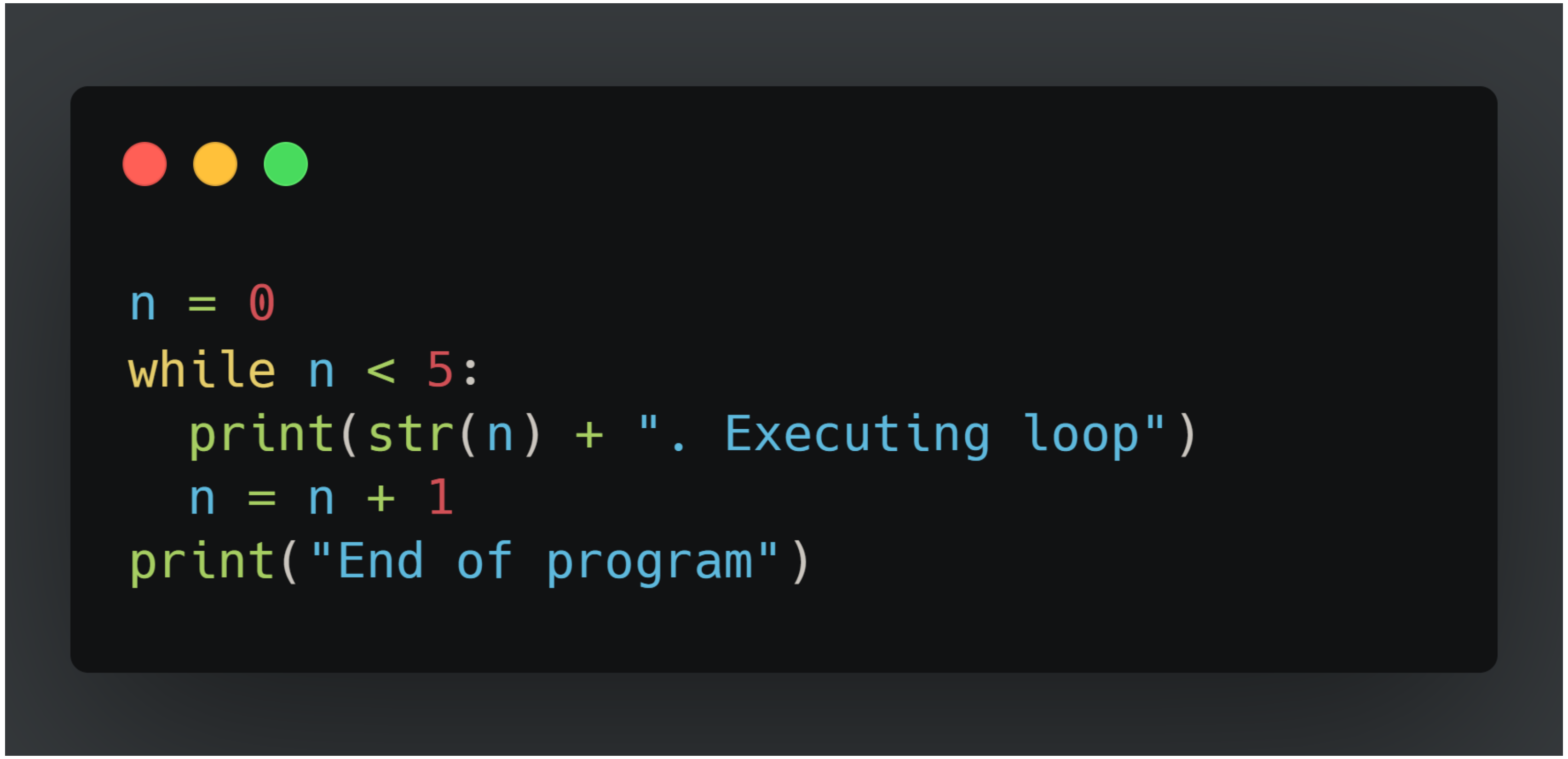
Fixed Loops



```
start = 4
stop = 34
step = 5
for i in range(start, stop, step):
  print(i, end=" ")

4 9 14 19 24 29
```

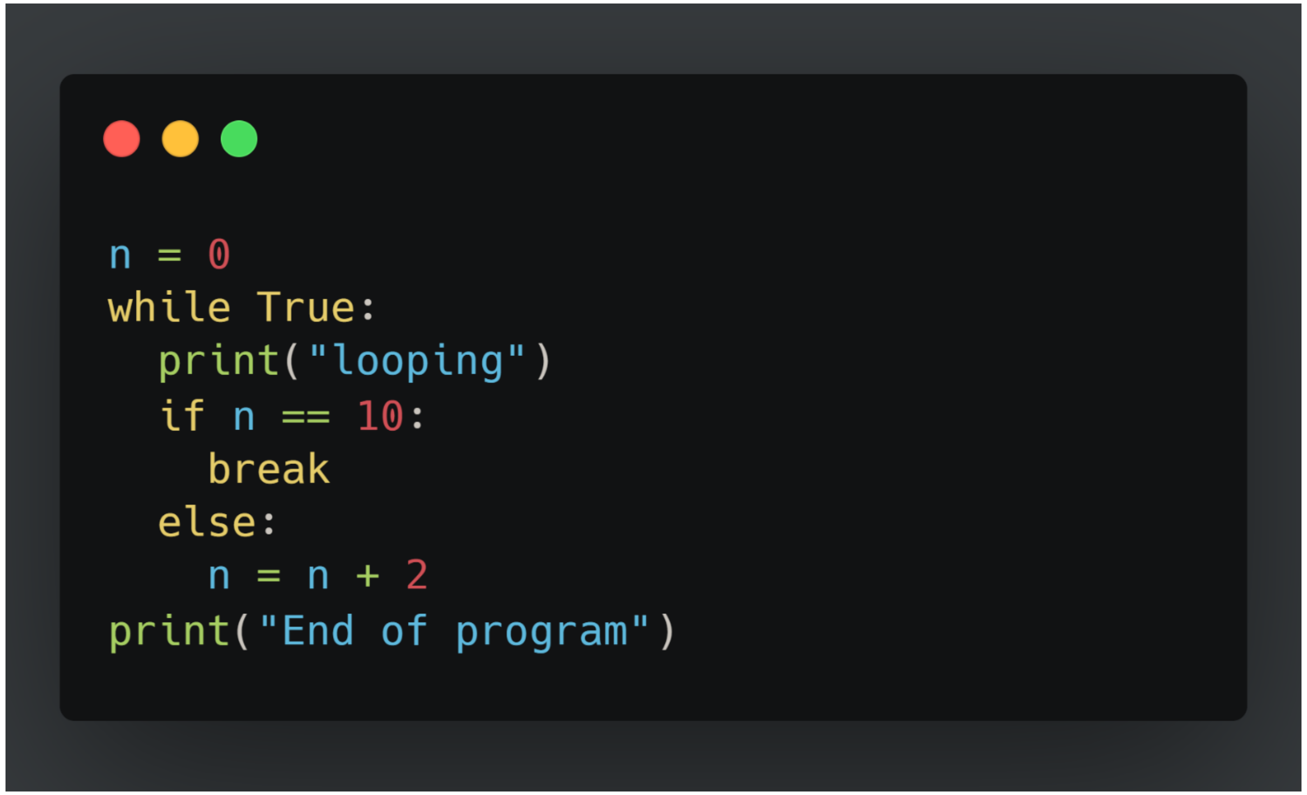
Conditional Loops

- Another useful loop. Similar to the for loop.
- A while loop doesn't run for a predefined number of iterations, like a for loop. Instead, it stops as soon as a given condition becomes false.



```python
n = 0
while n < 5:
  print(str(n) + ". Executing loop")
  n = n + 1
print("End of program")
```

Break statement



```
n = 0
while True:
  print("looping")
  if n == 10:
    break
  else:
    n = n + 2
print("End of program")
```

Allows us to go(break) out of a loop preliminary.

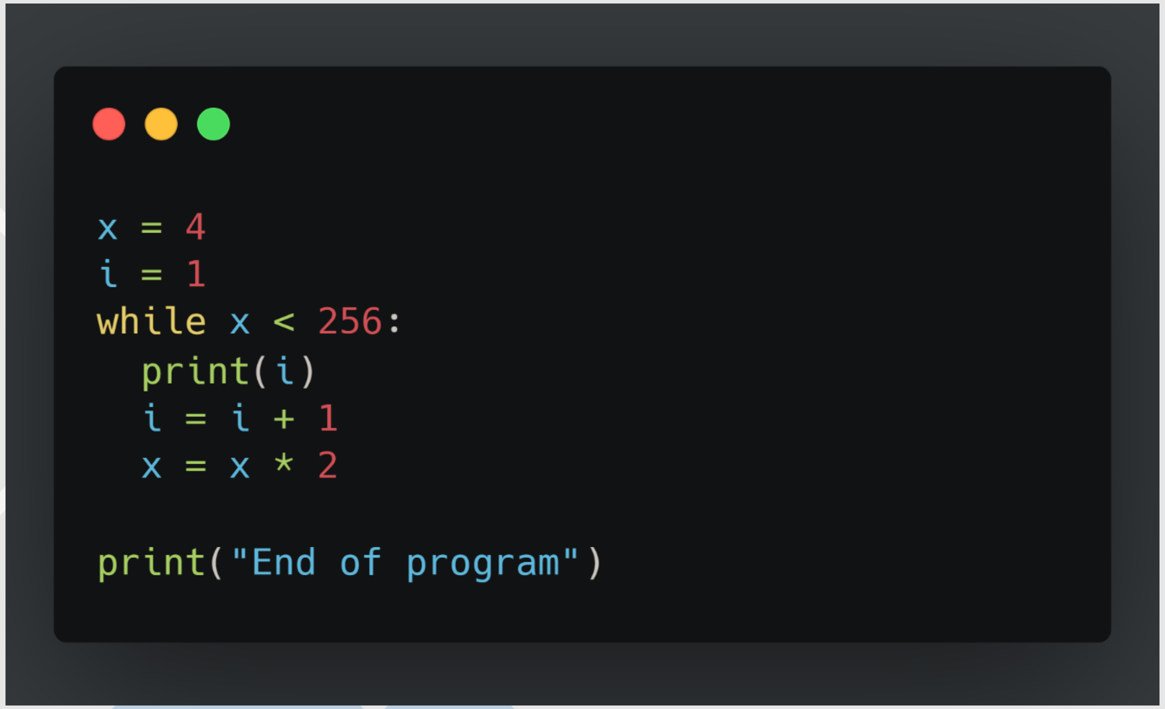
Adds a bit of controllability to a while loop.

Usually used with an if.

Can also be used in a for loop.

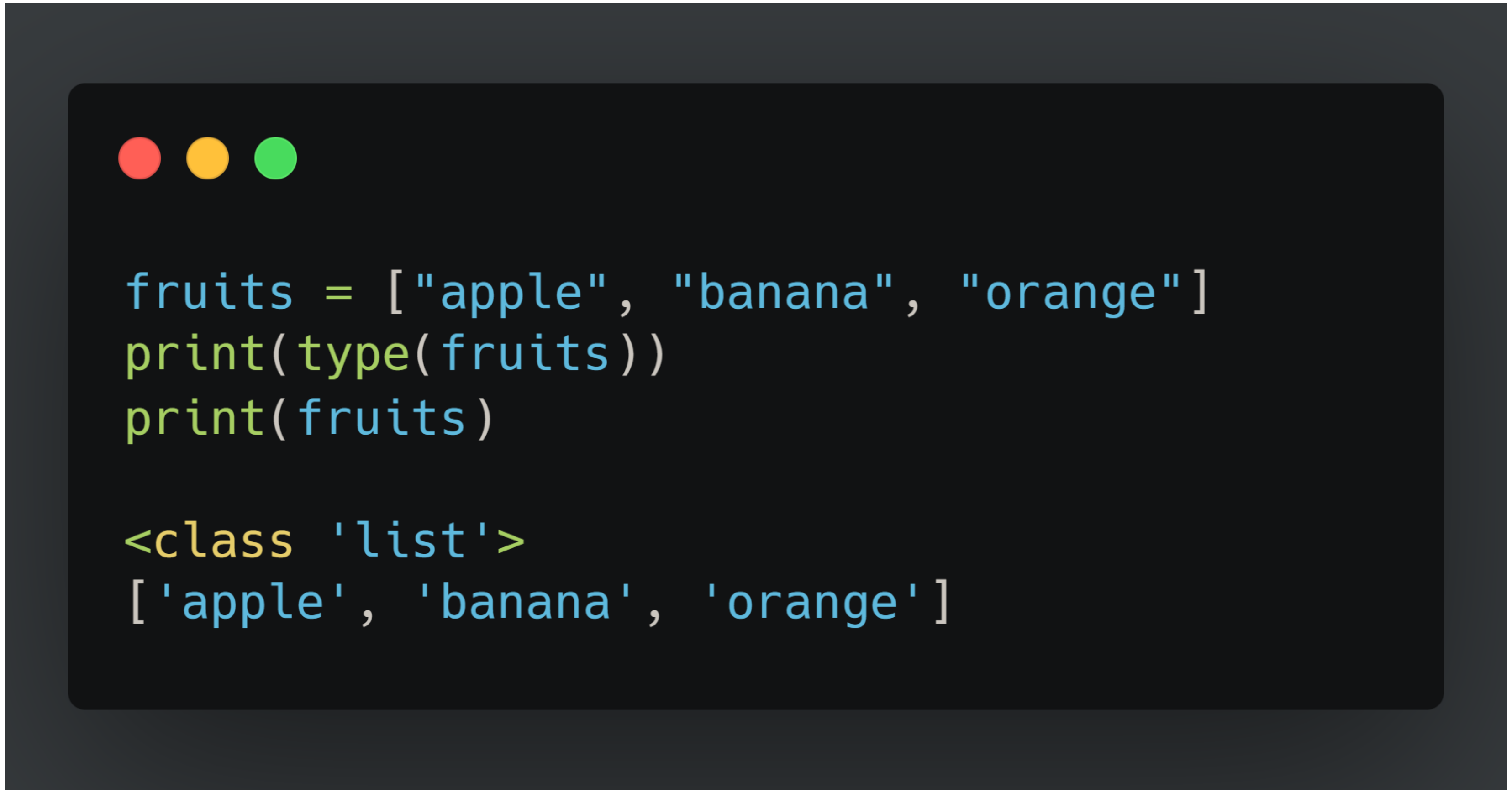
Quick quiz

How many times are we going to execute the while loop?



```
x = 4
i = 1
while x < 256:
  print(i)
  i = i + 1
  x = x * 2

print("End of program")
```

List

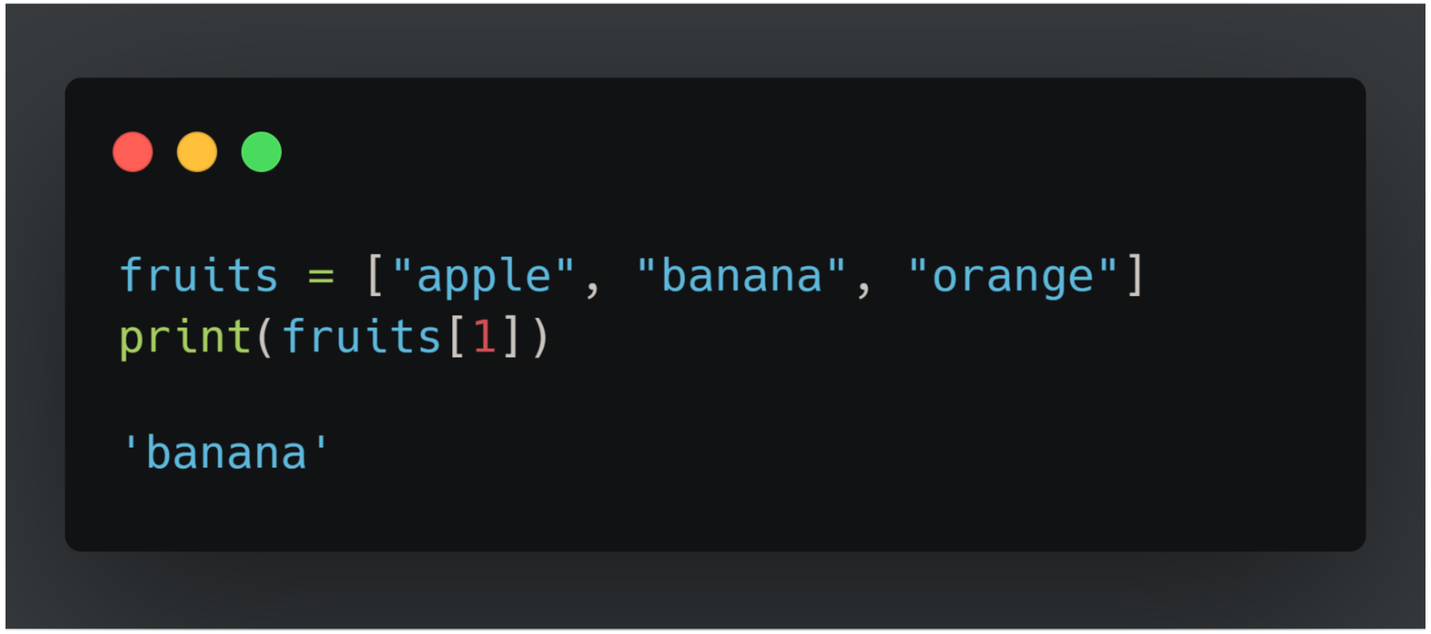


```
fruits = ["apple", "banana", "orange"]
print(type(fruits))
print(fruits)

<class 'list'>
['apple', 'banana', 'orange']
```

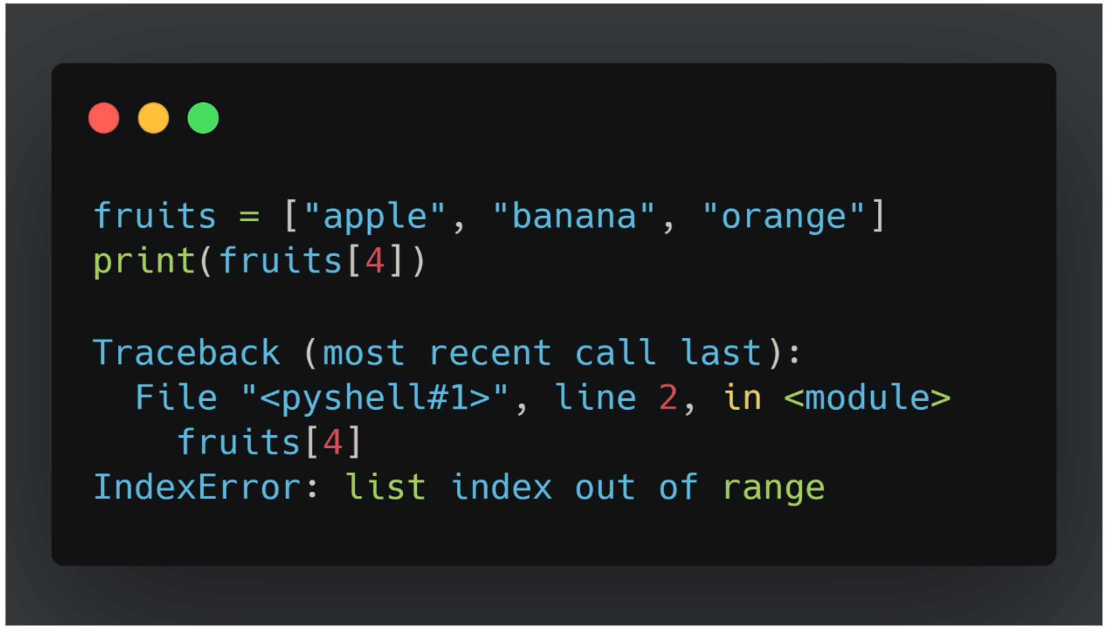
List

- We can access each of the individual element in the array using the subscript notation

- Index of lists start from 0



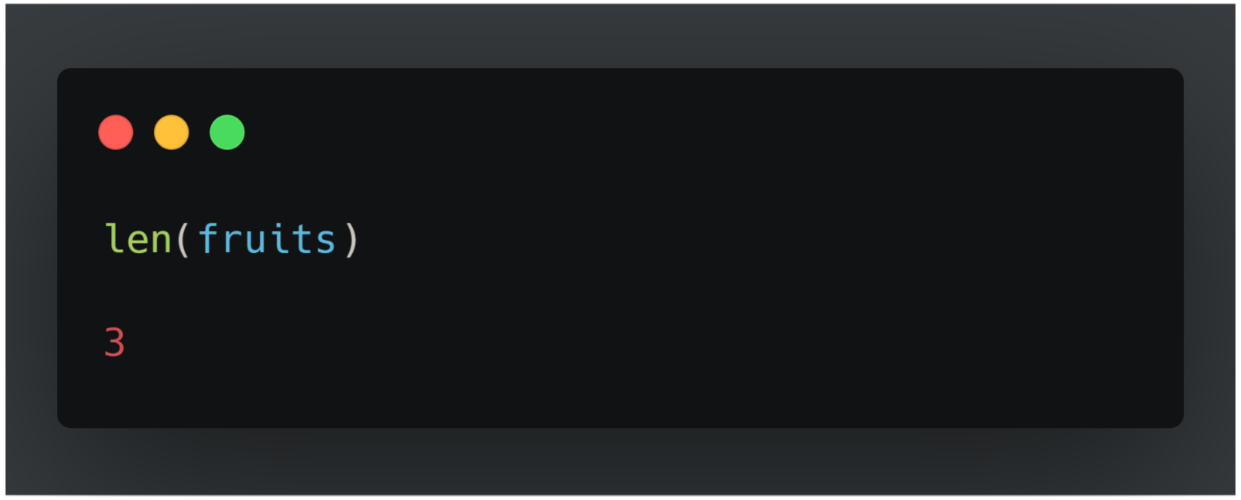
```
fruits = ["apple", "banana", "orange"]
print(fruits[1])

'banana'
```



```
fruits = ["apple", "banana", "orange"]
print(fruits[4])

Traceback (most recent call last):
  File "<pyshell#1>", line 2, in <module>
    fruits[4]
IndexError: list index out of range
```

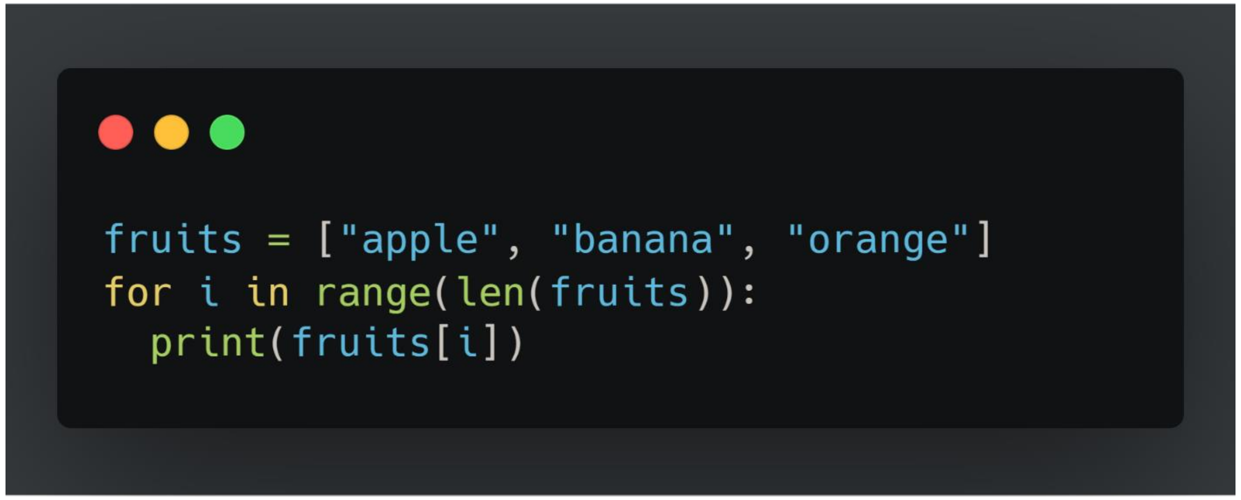
Example

- Say we want to go over a list and print each item along with its index



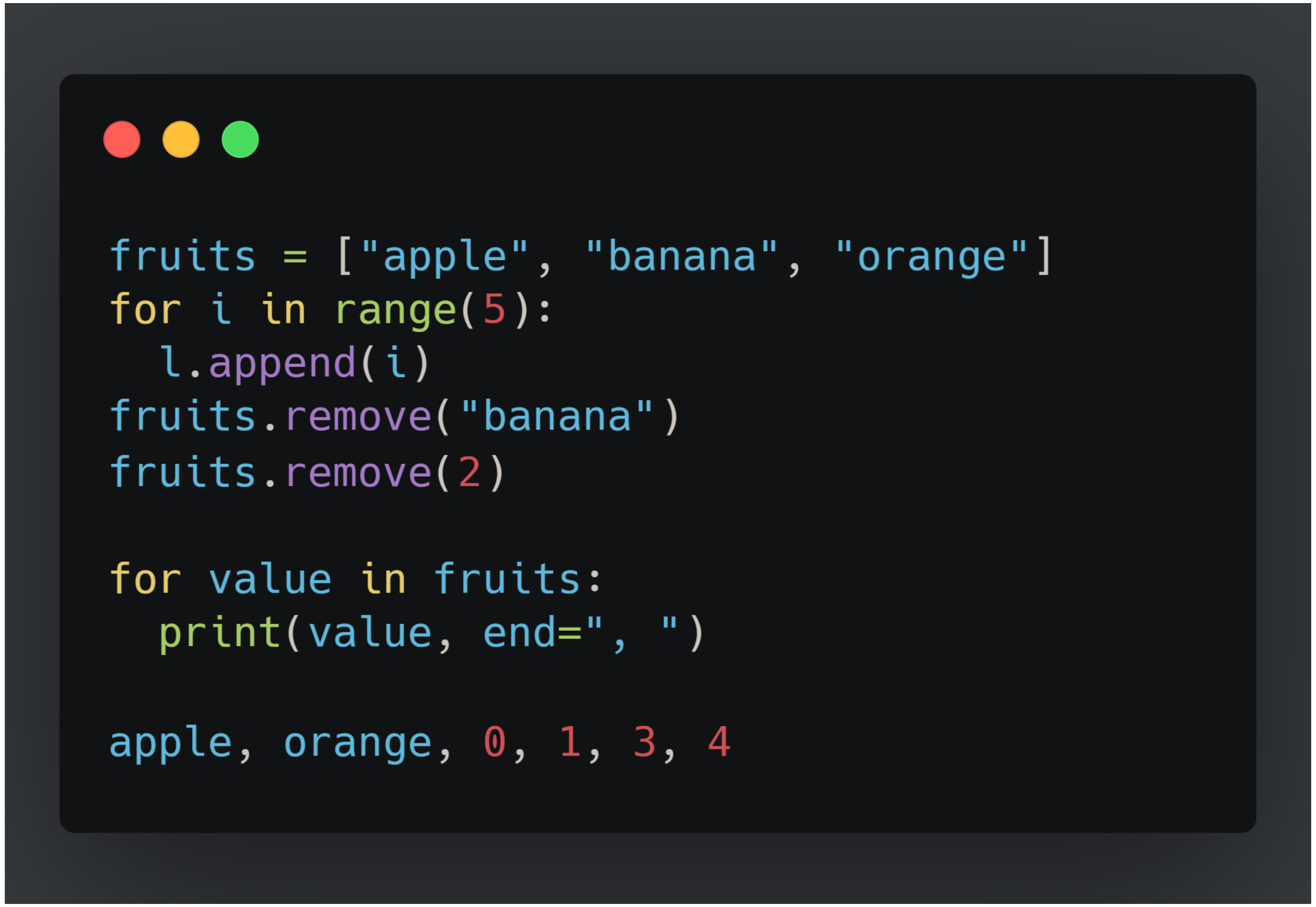
```
len(fruits)

3
```

- What if we have much more than 4 items in the list, say, 1000?



```
fruits = ["apple", "banana", "orange"]
for i in range(len(fruits)):
  print(fruits[i])
```

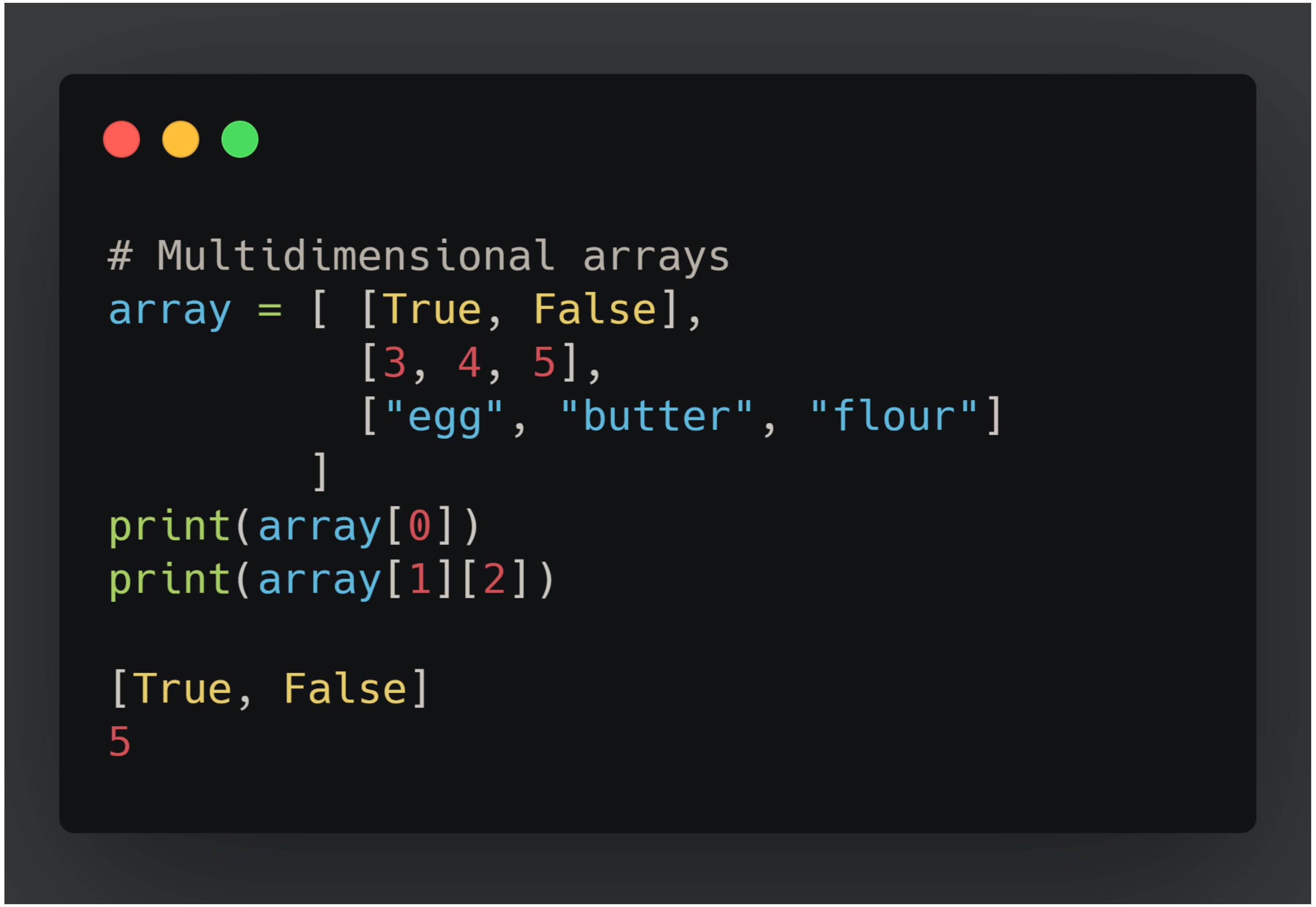
More Examples



```
fruits = ["apple", "banana", "orange"]
for i in range(5):
  l.append(i)
fruits.remove("banana")
fruits.remove(2)

for value in fruits:
  print(value, end=", ")

apple, orange, 0, 1, 3, 4
```

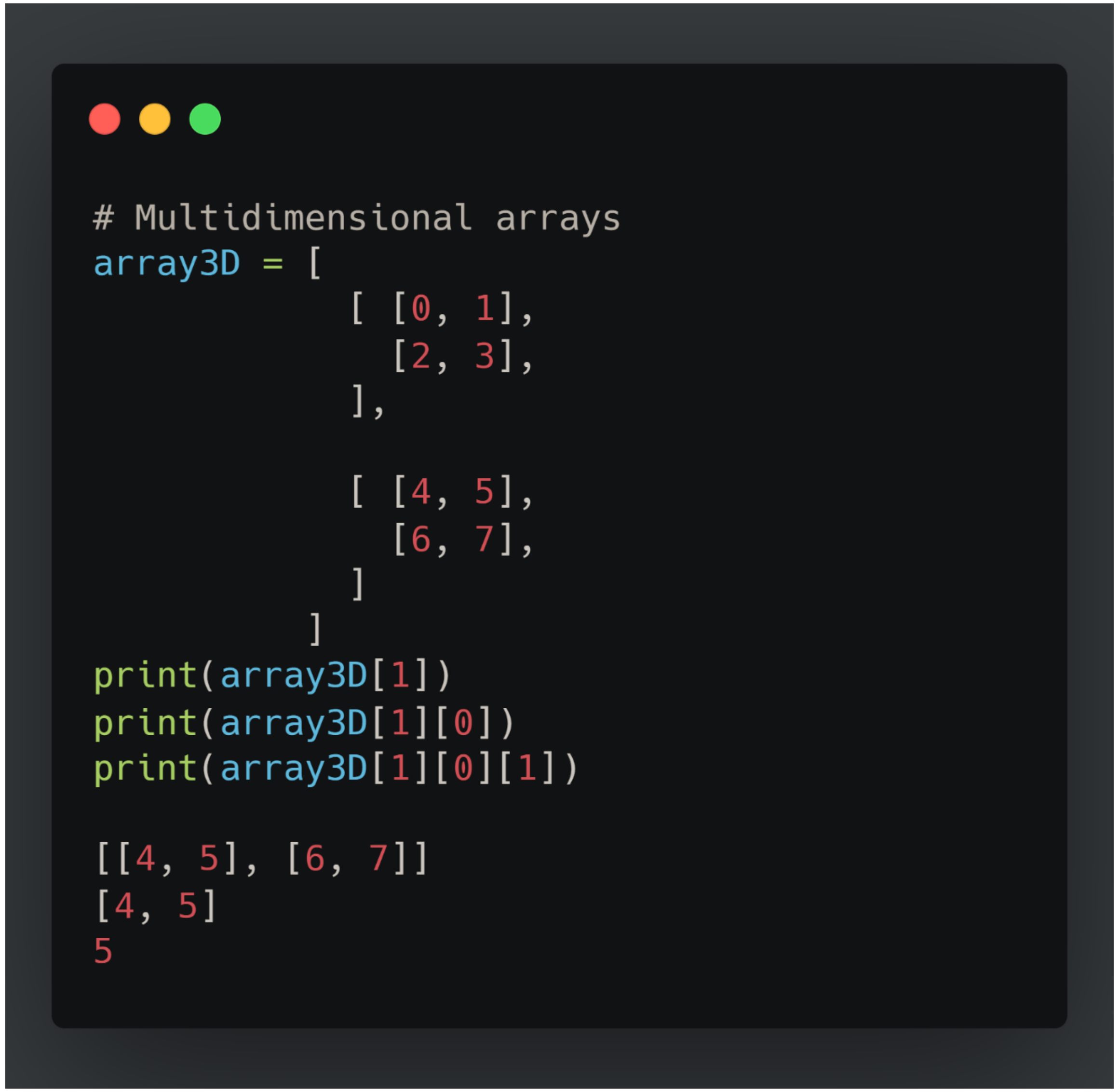
List

Multidimensional



```
# Multidimensional arrays
array = [ [True, False],
          [3, 4, 5],
          ["egg", "butter", "flour"]
        ]
print(array[0])
print(array[1][2])

[True, False]
5
```

3D arrays



```
# Multidimensional arrays
array3D = [
            [ [0, 1],
              [2, 3],
            ],

            [ [4, 5],
              [6, 7],
            ]
          ]
print(array3D[1])
print(array3D[1][0])
print(array3D[1][0][1])

[[4, 5], [6, 7]]
[4, 5]
5
```

Thank You