Python: An Introduction

OFE 2021 Summer Workshop
Haiyong Liu
Department of Economics

Agenda

- Introduction
- Running Python
- Python Programming
  - Variables
  - Types
  - Arithmetic operators
  - Boolean logic
  - Strings
  - Printing
- Exercises

What is python?

- Object oriented language
- Interpreted language
- Supports dynamic data type
- Independent from platforms
- Focused on development time
- Simple and easy grammar
- High-level internal object data types
- Automatic memory management
- It's free (open source)!

Brief History of Python

- Invented in the Netherlands, early 90s by Guido van Rossum
- Named after Monty Python
- Open sourced from the beginning
- Considered a scripting language, but is much more
- Scalable, object oriented and functional from the beginning
- Used by Google from the beginning
- Increasingly popular

Python’s Benevolent Dictator For Life

“Python is an experiment in how much freedom programmers need. Too much freedom and nobody can read another's code; too little and expressive-ness is endangered.”

- Guido van Rossum

Language properties

- Everything is an object
- Modules, classes, functions
- Exception handling
- Dynamic typing, polymorphism
- Static scoping
- Operator overloading
- Indentation for block structure

High-level data types

- Numbers: int, long, float, complex
- Strings: immutable
- Lists and dictionaries: containers
- Other types for e.g. binary data, regular expressions, introspection
- Extension modules can define new “built-in” data types

Why learn python?

- Fun-to-use "Scripting language"
- Object-oriented
  - Highly educational
- Very easy to learn
- Powerful, scalable, easy to maintain
  - high productivity
  - Lots of libraries
- Glue language
  - Interactive front-end for FORTRAN/C/C++ code

Where to use python?

- System management (i.e., scripting)
- Graphic User Interface (GUI)
- Internet programming
- Database (DB) programming
- Text data processing
- Distributed processing
- Numerical operations
- Graphics
- And so on…

Why learn python? (cont.)

- Reduce development time
- Reduce code length
- Easy to learn and use as developers
- Easy to understand codes
- Easy to do team projects
- Easy to extend to other languages

Course Goals

- To understand the basic structure and syntax of Python programming language
- To write your own simple Python scripts.
- To serve as the starting point for more advanced training on Python coding

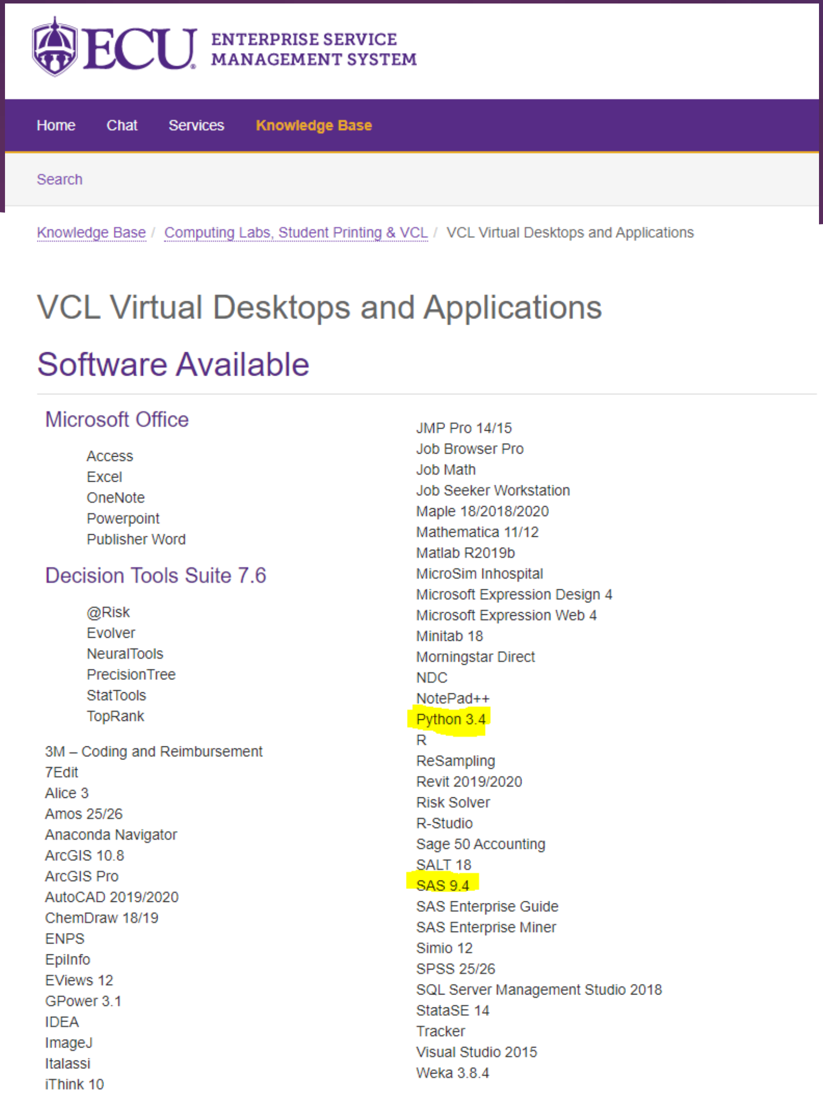
Agenda

- Introduction
- Running Python
- Python Programming
  - Data types
  - Control flows
  - Classes, functions, modules
- Hands-on Exercises

Access Python from ECU

remoteaccess.ecu.edu

https://ecu.teamdynamix.com/TDClient/1409/Portal/KB
/ArticleDet?ID=67605



Knowledge Base / Computing Labs, Student Printing & VCL / VCL Virtual Desktops and Applications

VCL Virtual Desktops and Applications

Software Available

Microsoft Office

- Access
- Excel
- OneNote
- Powerpoint
- Publisher Word

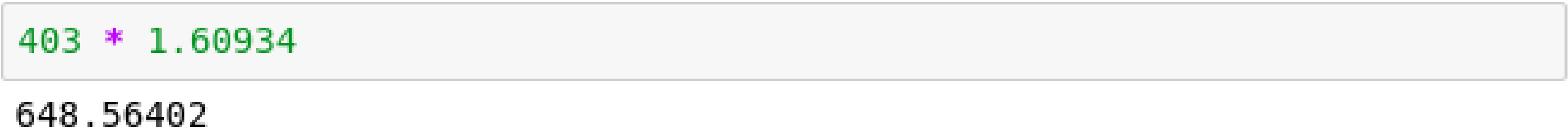
Decision Tools Suite 7.6

- @Risk
- Evolver
- NeuralTools
- PrecisionTree
- StatTools
- TopRank

3M – Coding and Reimbursement
7Edit
Alice 3
Amos 25/26
Anaconda Navigator
ArcGIS 10.8
ArcGIS Pro
AutoCAD 2019/2020
ChemDraw 18/19
ENPS
EpiInfo
EViews 12
GPower 3.1
IDEA
ImageJ
Italassi
iThink 10
JMP Pro 14/15
Job Browser Pro
Job Math
Job Seeker Workstation
Maple 18/2018/2020
Mathematica 11/12
Matlab R2019b
MicroSim Inhospital
Microsoft Expression Design 4
Microsoft Expression Web 4
Minitab 18
Morningstar Direct
NDC
NotePad++
Python 3.4
R
ReSampling
Revit 2019/2020
Risk Solver
R-Studio
Sage 50 Accounting
SALT 18
SAS 9.4
SAS Enterprise Guide
SAS Enterprise Miner
Simio 12
SPSS 25/26
SQL Server Management Studio 2018
StataSE 14
Tracker
Visual Studio 2015
Weka 3.8.4

Python as a calculator

- Let us calculate the distance between Edinburgh and London in km

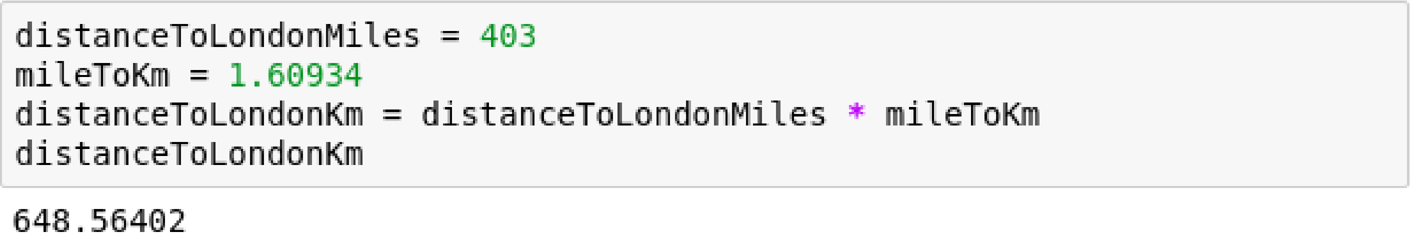


```
403 * 1.60934
```

```
648.56402
```

Variables

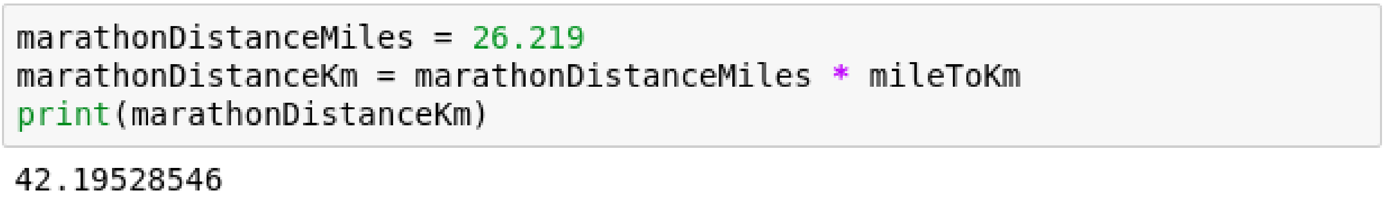
- Great calculator but how can we make it store values?
- Do this by defining variables
- Can later be called by the variable name
- Variable names are case sensitive and unique



```
distanceToLondonMiles = 403
mileToKm = 1.60934
distanceToLondonKm = distanceToLondonMiles * mileToKm
distanceToLondonKm
```

```
648.56402
```

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

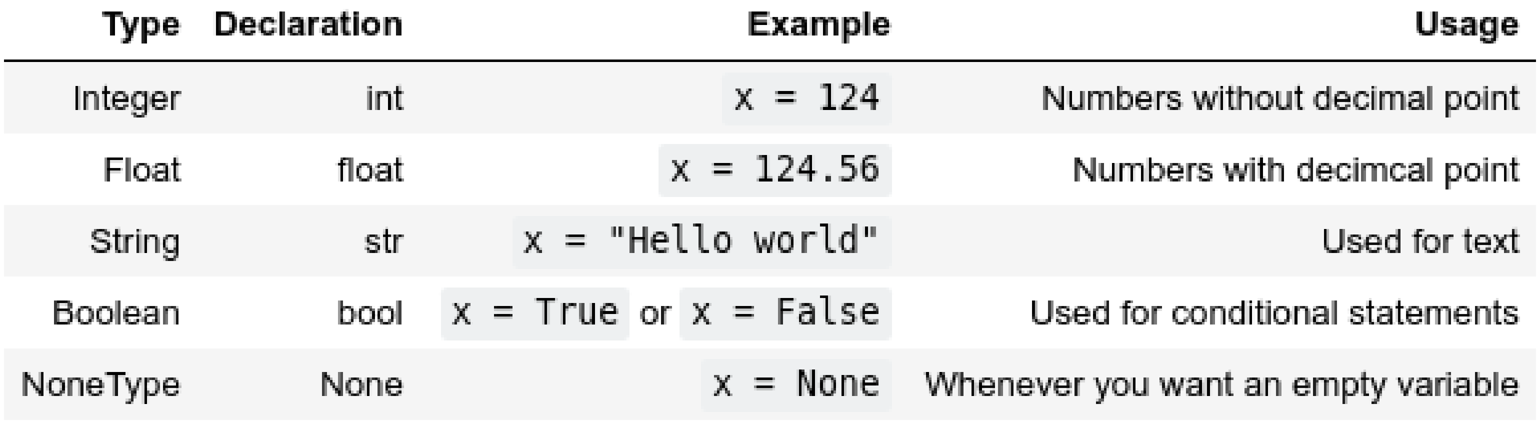


```
marathonDistanceMiles = 26.219
marathonDistanceKm = marathonDistanceMiles * mileToKm
print(marathonDistanceKm)
```

```
42.19528546
```

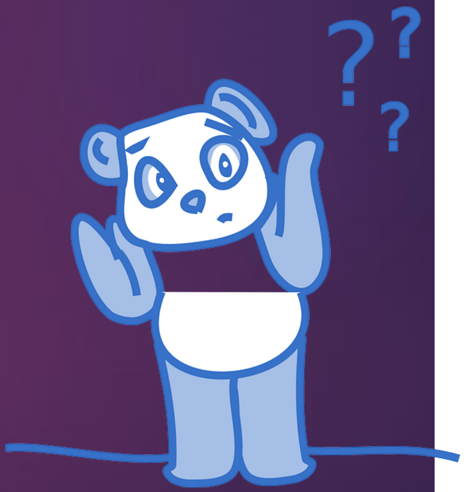
Types

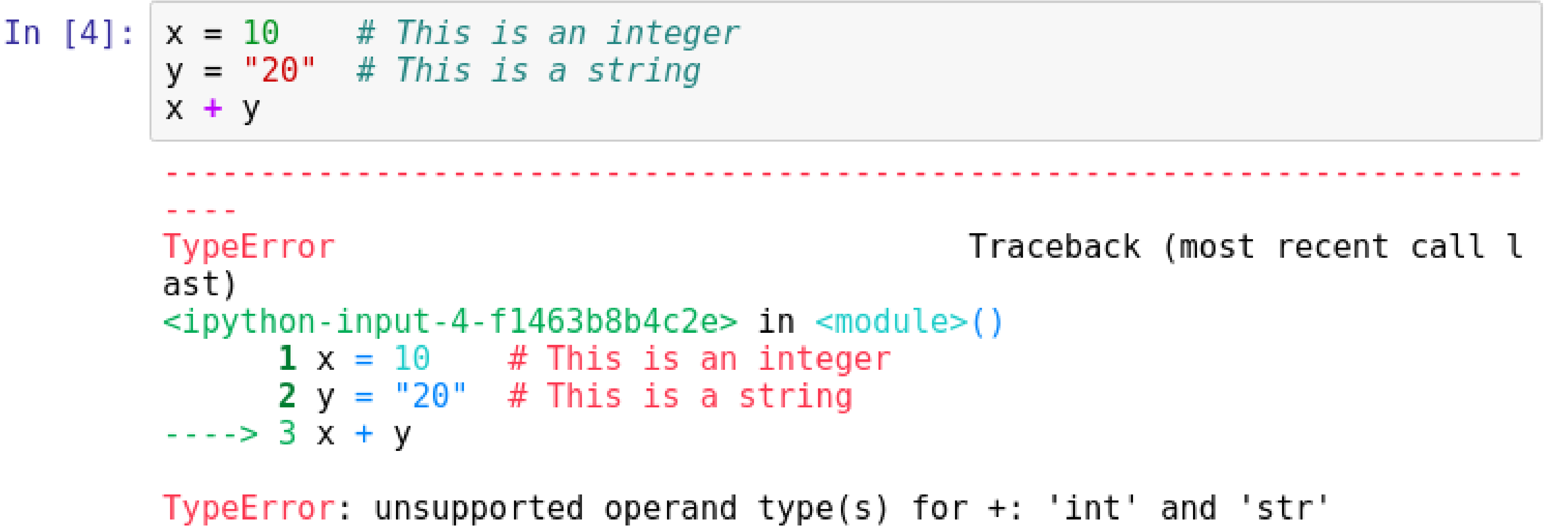
Variables actually have a type, which defines the way it is stored.



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?





```
In [4]: x = 10    # This is an integer
        y = "20"  # This is a string
        x + y
```

```
---------------------------------------------------------------------------
----
TypeError                                 Traceback (most recent call l
ast)
<ipython-input-4-f1463b8b4c2e> in <module>()
      1 x = 10    # This is an integer
      2 y = "20"  # This is a string
----> 3 x + y

TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

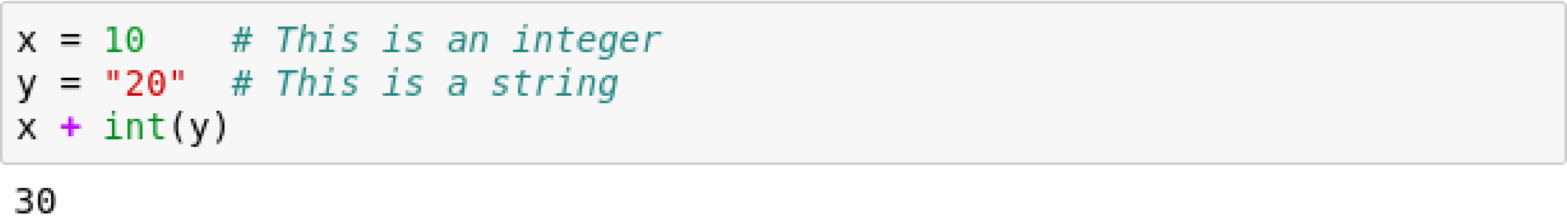
Important lesson to remember!
We can't do arithmetic operations on variables of different types. Therefore make sure that you are always aware of your variables types!

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

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

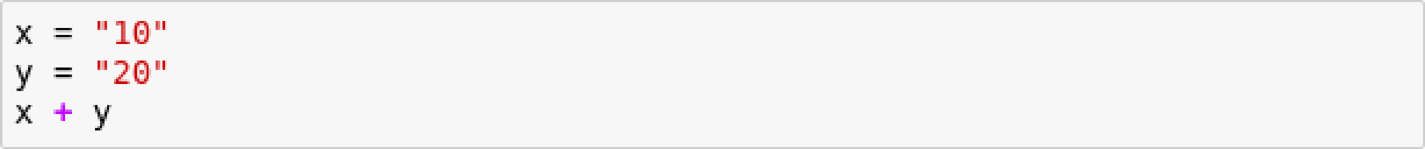


```
x = 10    # This is an integer
y = "20"  # This is a string
x + int(y)
```

```
30
```

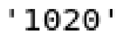
Similar methods exist for other data types: **int()**, **float()**, **str()**

Quick quiz



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

What will be the result?



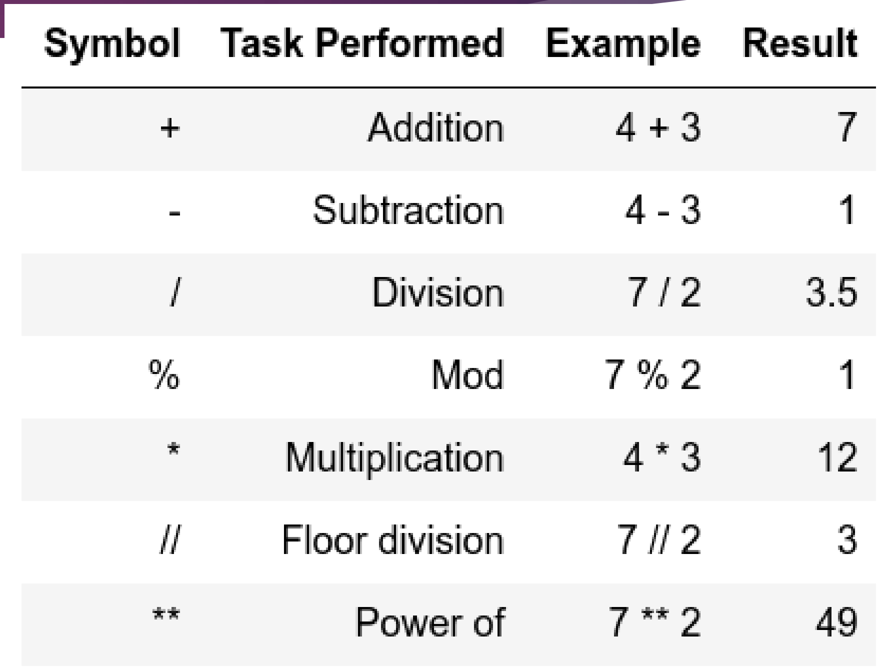
```
'1020'
```

Arithmetic operations

Similar to actual Mathematics.

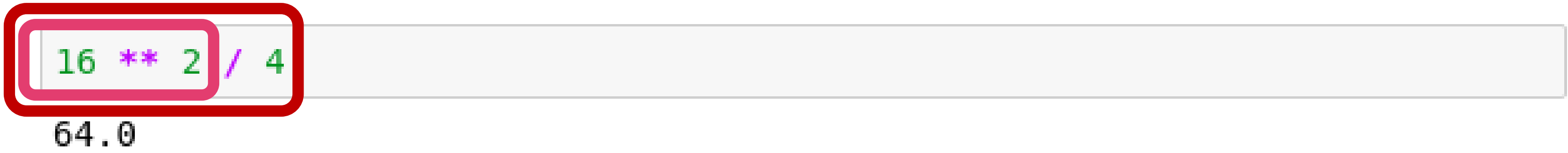
Order of precedence is the same as in Mathematics.

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

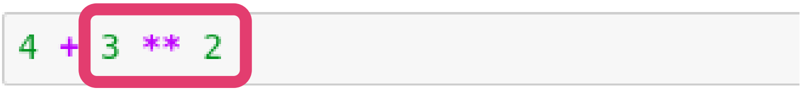
Order precedence example



```
16 ** 2 / 4
```

```
64.0
```

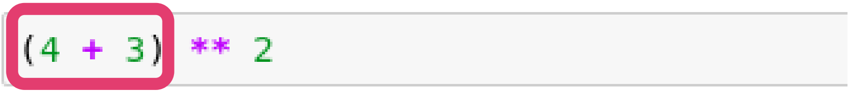
Quick quiz



```
4 + 3 ** 2
```

13

vs



```
(4 + 3) ** 2
```

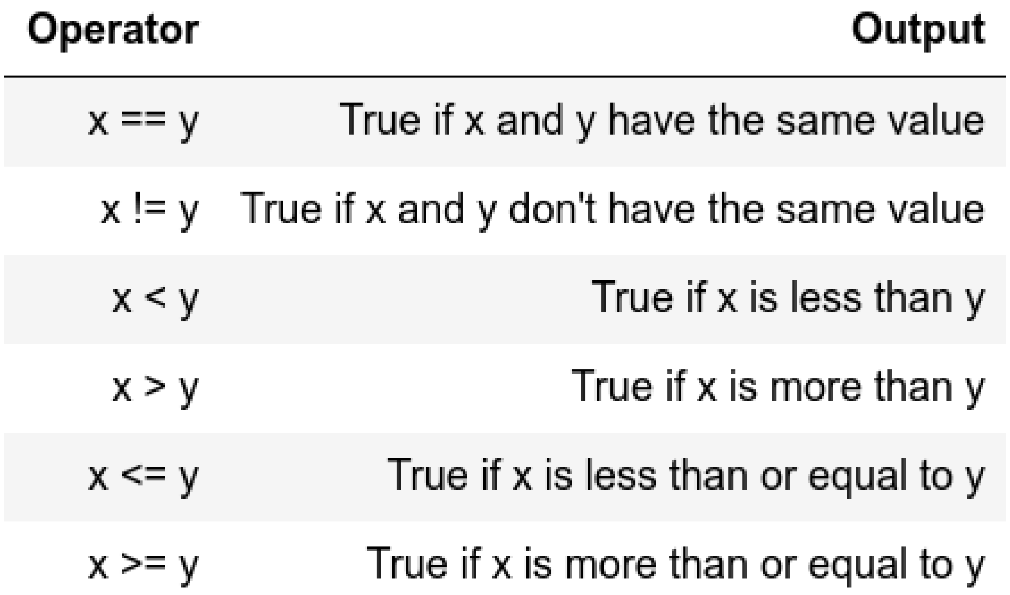
49

Comparison operators

- I.e. comparison operators
- Return Boolean values

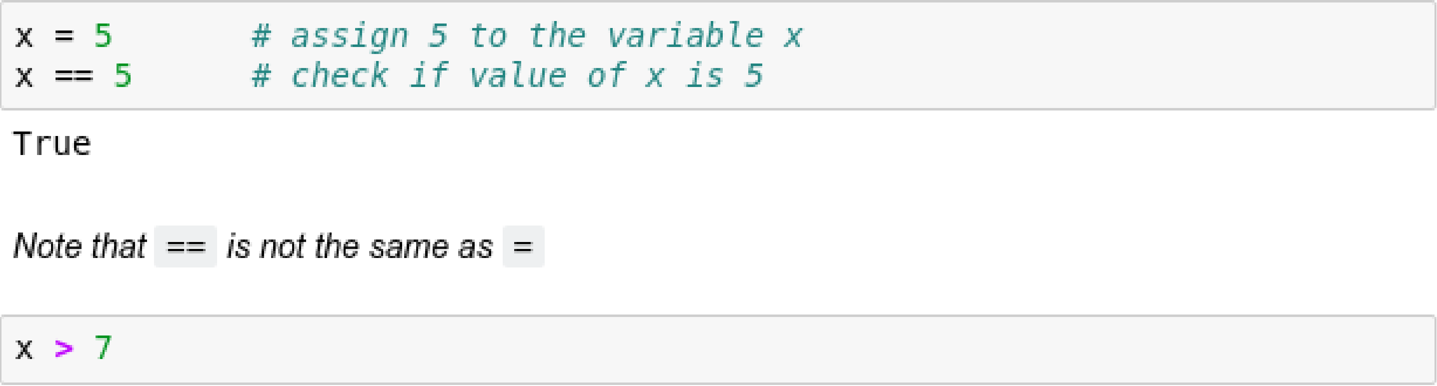
(i.e. True or False)

- 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
```

```
True
```

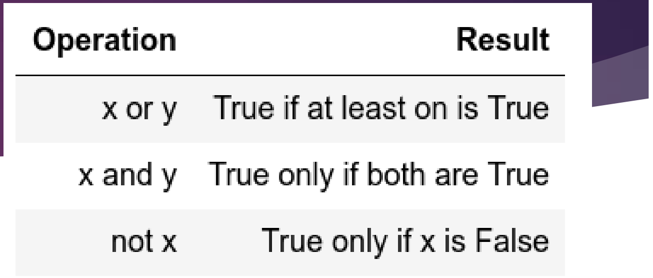
Note that `==` *is not the same as* `=`

```
x > 7
```

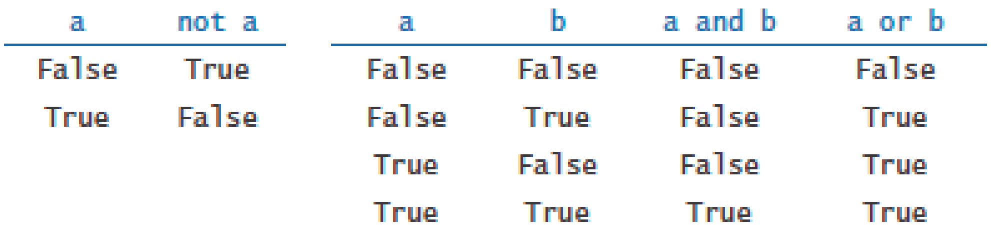
False

Logical operators

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



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

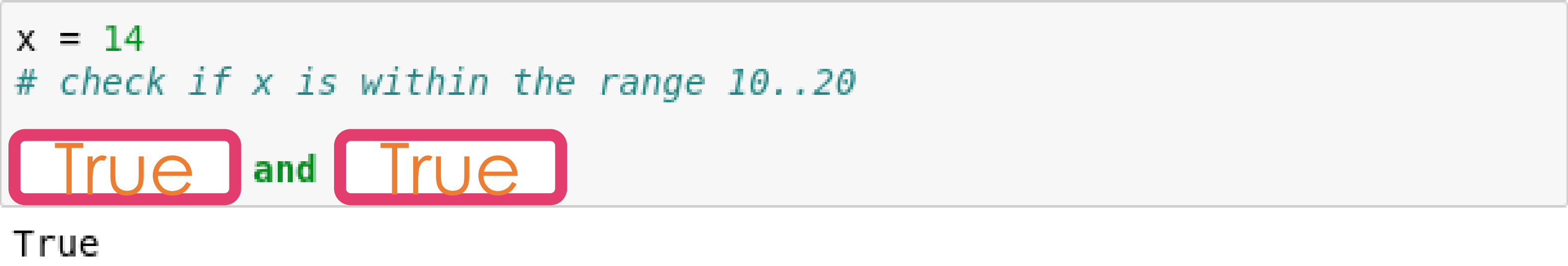


a	not a
False	True
True	False

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

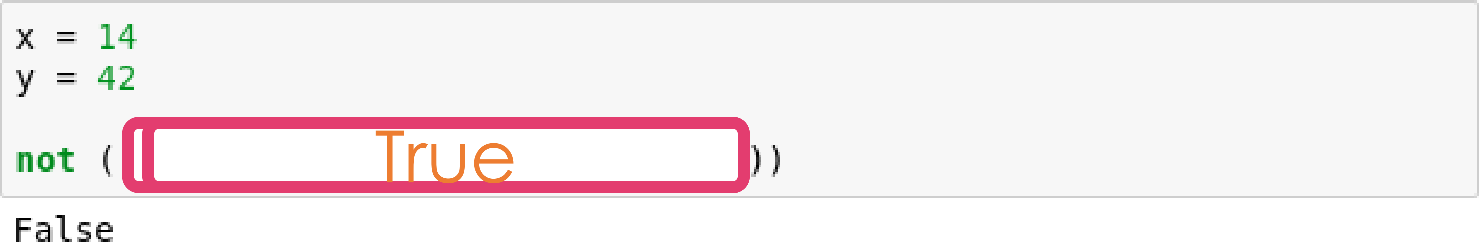
Combining both



```
x = 14
# check if x is within the range 10..20
True and True
```

```
True
```

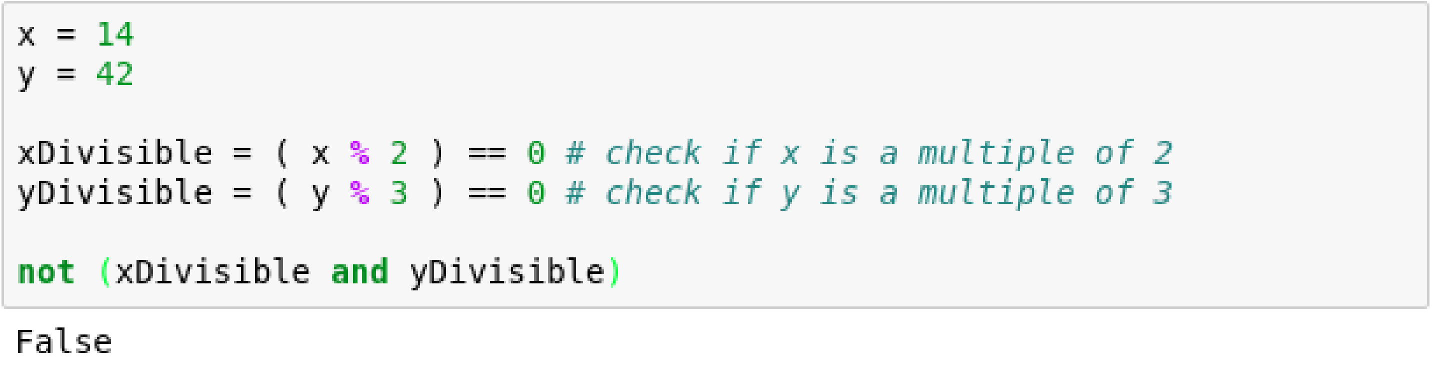
Another example



```
x = 14
y = 42

not (True))
```

```
False
```

That wasn't very easy to read was it?
Is there a way we can make it more readable?



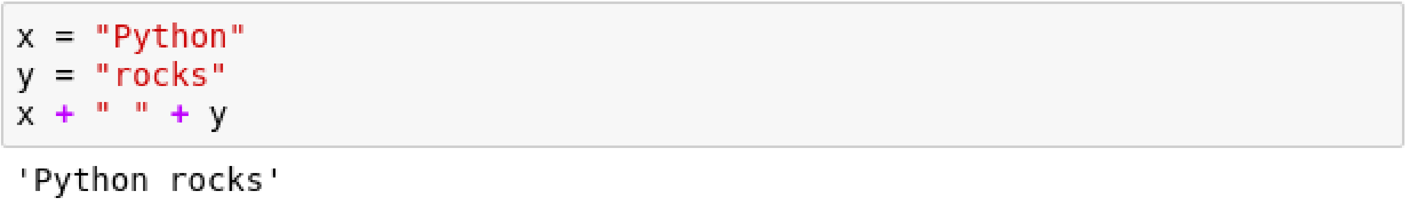
```
x = 14
y = 42

xDivisible = ( x % 2 ) == 0 # check if x is a multiple of 2
yDivisible = ( y % 3 ) == 0 # check if y is a multiple of 3

not (xDivisible and yDivisible)
```

```
False
```

Strings

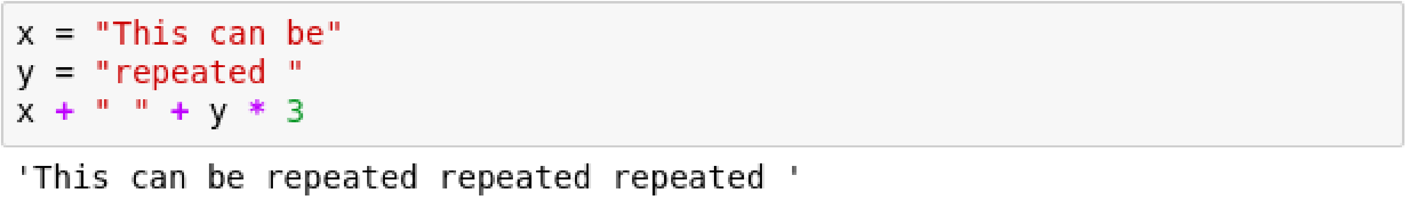
- Powerful and flexible in Python
- Can be added
- Can be multiplied
- Can be multiple lines

Strings



```
x = "Python"
y = "rocks"
x + " " + y
```

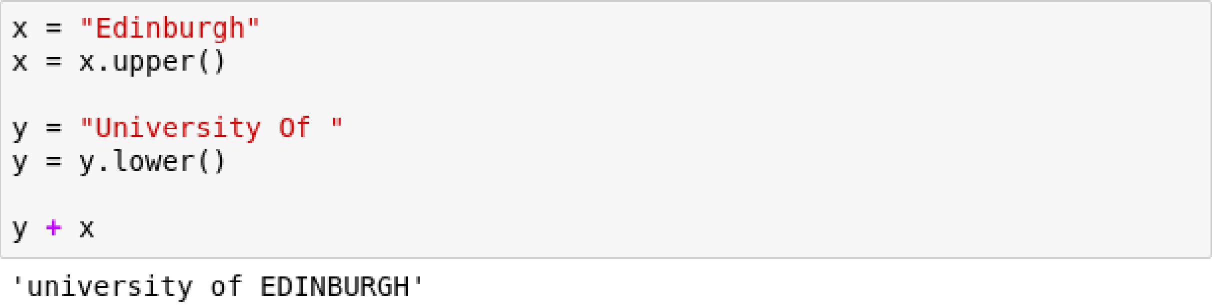
```
'Python rocks'
```



```
x = "This can be"
y = "repeated "
x + " " + y * 3
```

```
'This can be repeated repeated repeated '
```

Strings



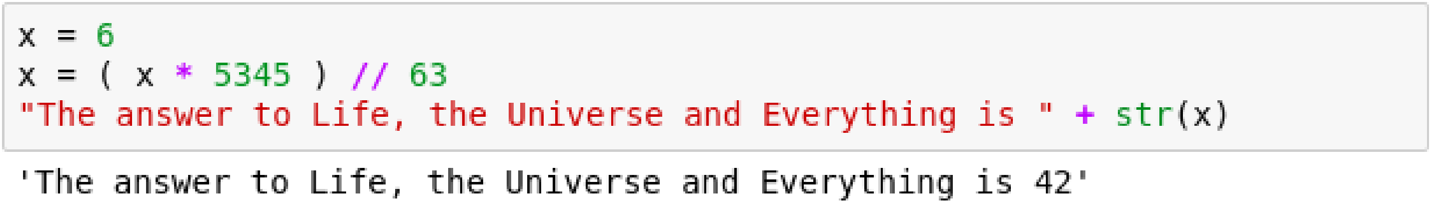
```
x = "Edinburgh"
x = x.upper()

y = "University Of "
y = y.lower()

y + x
```

```
'university of EDINBURGH'
```

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')**

Mixing up strings and numbers

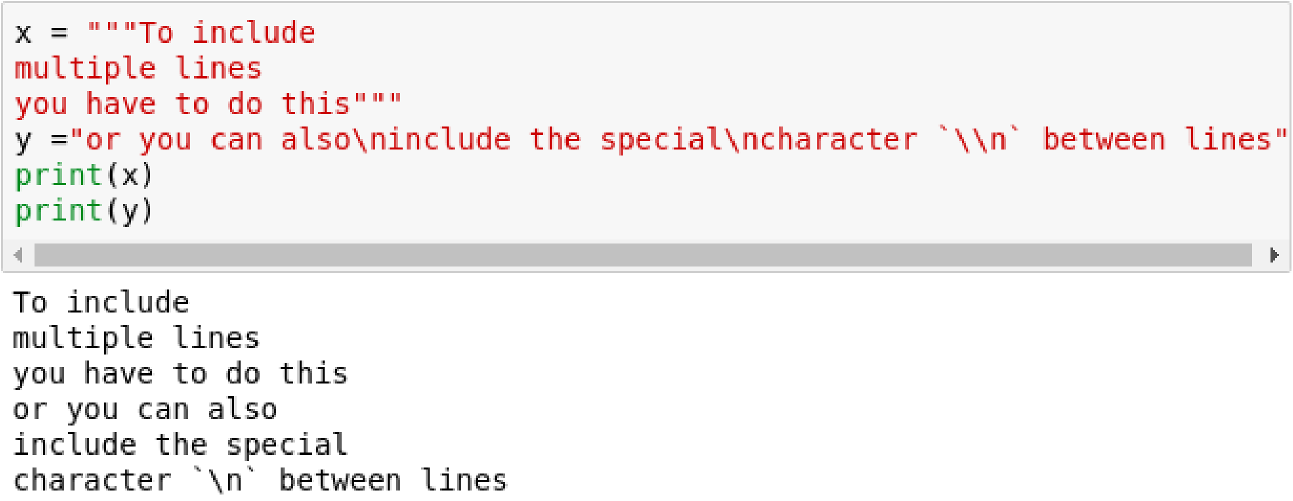
Often we would need to mix up numbers and strings.
It is best to keep numbers as numbers (i.e. int or float)
and cast them to strings whenever we need them as a string.



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

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

Multiline strings

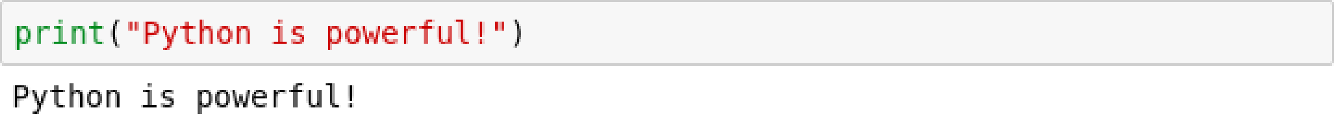


```
x = """To include
multiple lines
you have to do this"""
y ="or you can also\ninclude the special\ncharacter `\\n` between lines"
print(x)
print(y)
```

```
To include
multiple lines
you have to do this
or you can also
include the special
character `\n` between lines
```

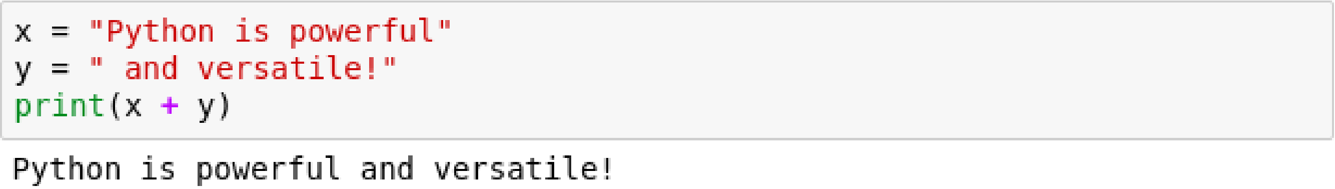
Printing

- When writing scripts, your outcomes aren't printed on the terminal.
- Thus, you must print them yourself with the print()



```python
print("Python is powerful!")
```

```
Python is powerful!
```

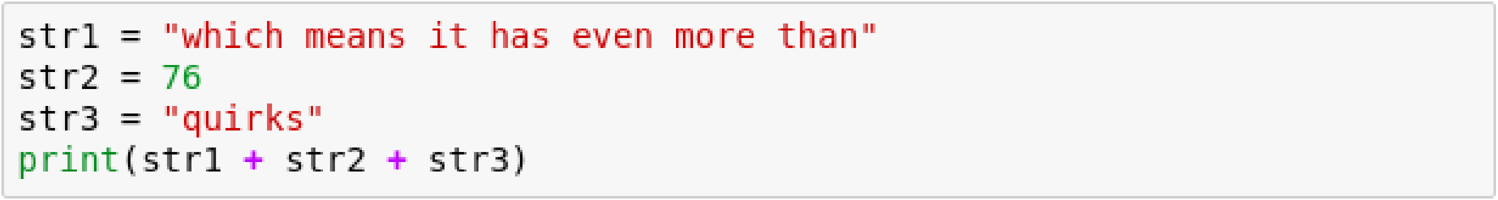


```python
x = "Python is powerful"
y = " and versatile!"
print(x + y)
```

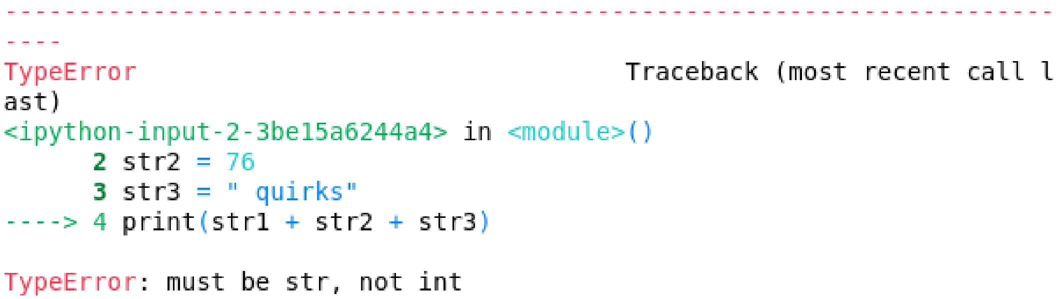
```
Python is powerful and versatile!
```

Quick quiz

Do you see anything wrong with this block?

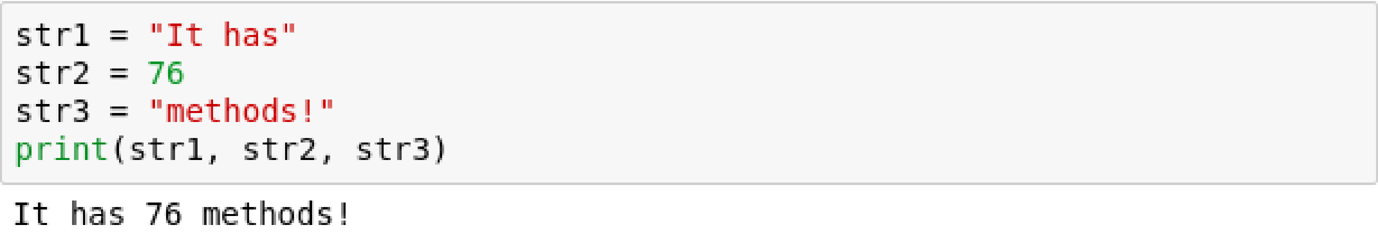


```
str1 = "which means it has even more than"
str2 = 76
str3 = "quirks"
print(str1 + str2 + str3)
```



```
---------------------------------------------------------------------------
----
TypeError                                 Traceback (most recent call l
ast)
<ipython-input-2-3be15a6244a4> in <module>()
      2 str2 = 76
      3 str3 = " quirks"
----> 4 print(str1 + str2 + str3)

TypeError: must be str, not int
```

Another more generic way to fix it



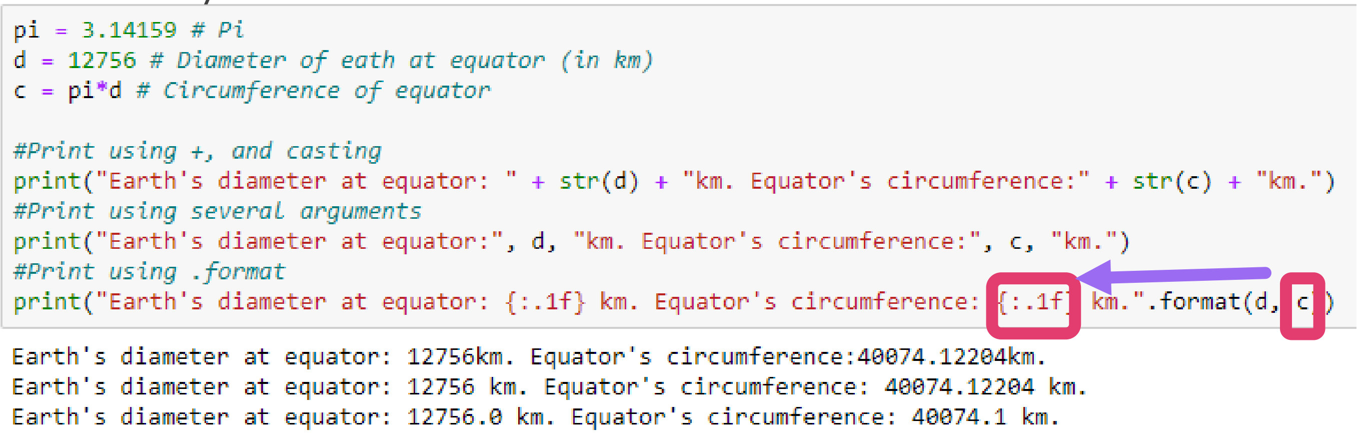
```
str1 = "It has"
str2 = 76
str3 = "methods!"
print(str1, str2, str3)
```

```
It has 76 methods!
```

If we comma separate statements in a print function we can have different variables printing!

Placeholders

- A way to interleave numbers is



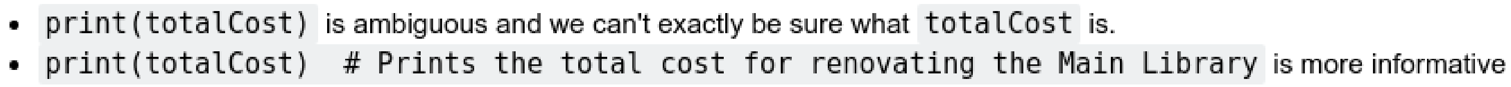
```python
pi = 3.14159 # Pi
d = 12756 # Diameter of eath at equator (in km)
c = pi*d # Circumference of equator

#Print using +, and casting
print("Earth's diameter at equator: " + str(d) + "km. Equator's circumference:" + str(c) + "km.")
#Print using several arguments
print("Earth's diameter at equator:", d, "km. Equator's circumference:", c, "km.")
#Print using .format
print("Earth's diameter at equator: {:.1f} km. Equator's circumference: {:.1f} km.".format(d, c))
```

```
Earth's diameter at equator: 12756km. Equator's circumference:40074.12204km.
Earth's diameter at equator: 12756 km. Equator's circumference: 40074.12204 km.
Earth's diameter at equator: 12756.0 km. Equator's circumference: 40074.1 km.
```

- Elegant and easy
- more in your notes

Commenting

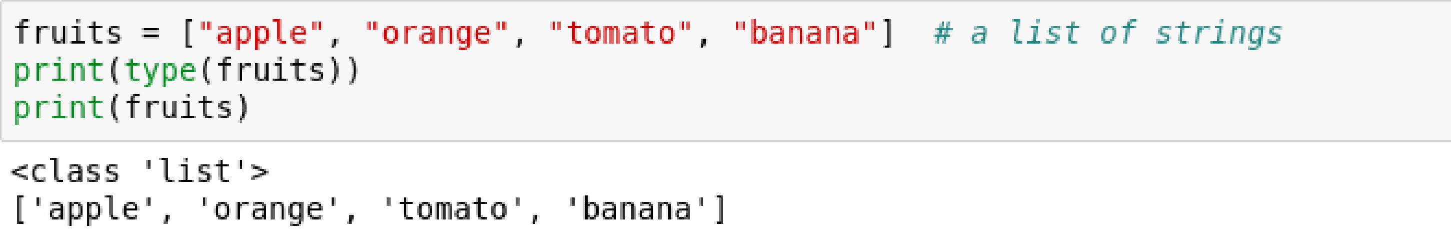
- Useful when your code needs further explanation. Either for your future self and anybody else.
- Useful when you want to remove the code from execution but not permanently
- Comments in Python are done with #



- `print(totalCost)` is ambiguous and we can't exactly be sure what `totalCost` is.
- `print(totalCost)  # Prints the total cost for renovating the Main Library` is more informative

Lists

- One of the most useful concepts
- Group multiple variables together (a kind of **container**!)

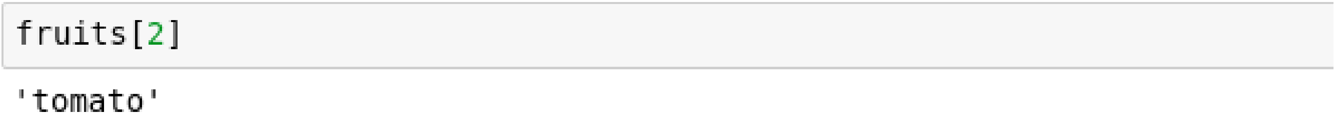


```
fruits = ["apple", "orange", "tomato", "banana"]  # a list of strings
print(type(fruits))
print(fruits)
```

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

Indexing a list

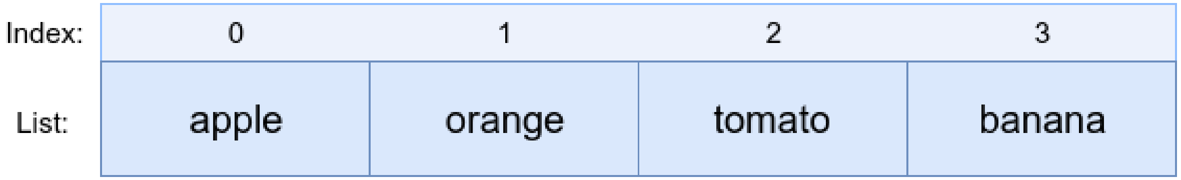
- Indexing – accessing items within a data structure



```
fruits[2]
```

```
'tomato'
```

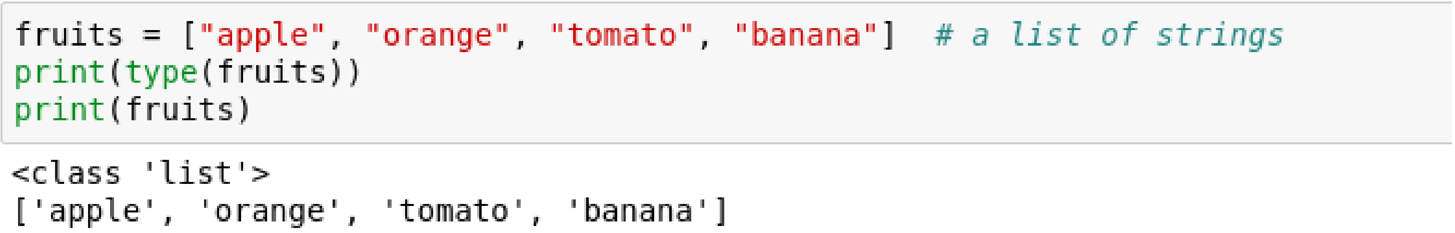
- Indexing a list is not very intuitive...
- The first element of a list has an **index 0**



Index:	0	1	2	3
List:	apple	orange	tomato	banana

Quick quiz

What will **fruits[3]** return?

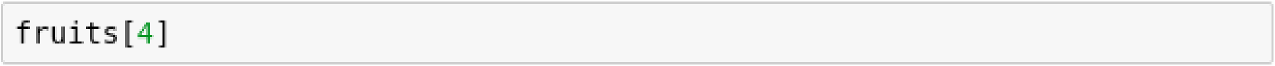


```
fruits = ["apple", "orange", "tomato", "banana"]  # a list of strings
print(type(fruits))
print(fruits)
```

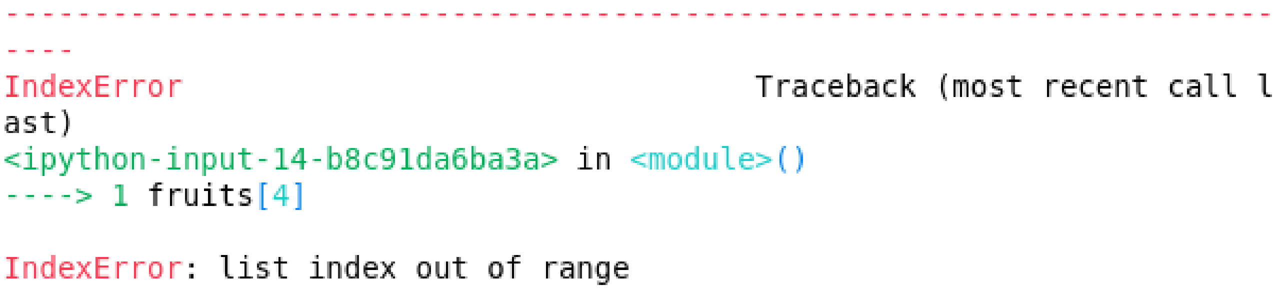
```
<class 'list'>
['apple', 'orange', 'tomato', 'banana']
```

Quick quiz

What will this return?



```
fruits[4]
```



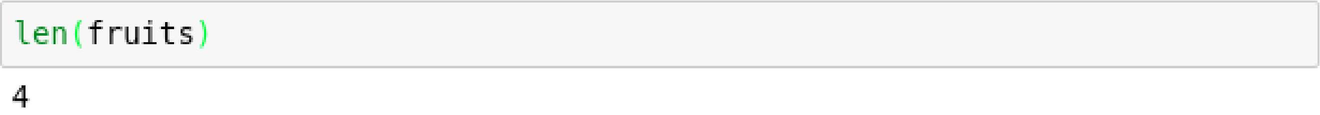
```
-----------------------------------------------------------------------
----
IndexError                                Traceback (most recent call l
ast)
<ipython-input-14-b8c91da6ba3a> in <module>()
----> 1 fruits[4]

IndexError: list index out of range
```

Data structure sizes

Make sure you are always aware of the sizes of each variable!

This can easily be done using the **len()** function.

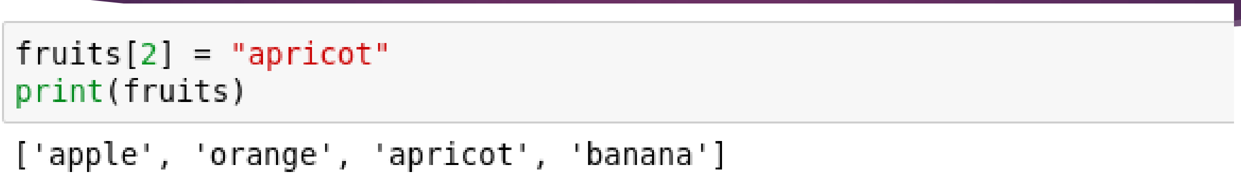
It returns the length/size of any data structure



```
len(fruits)
```

```
4
```

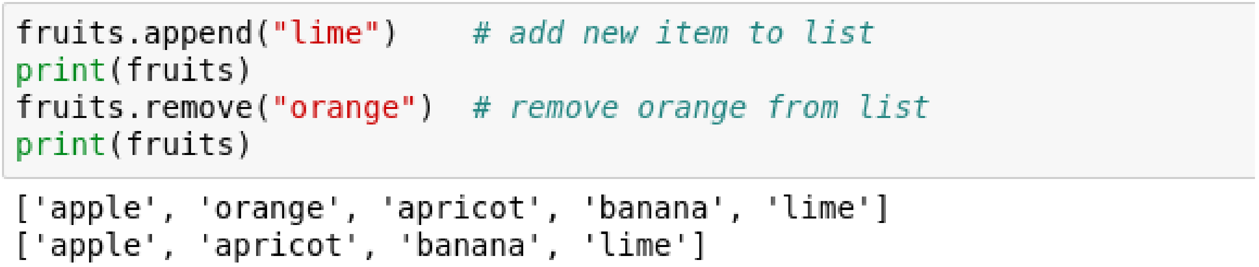
Is a tomato really a fruit?



```
fruits[2] = "apricot"
print(fruits)
```

```
['apple', 'orange', 'apricot', 'banana']
```

Furthermore, we can modify lists in various ways

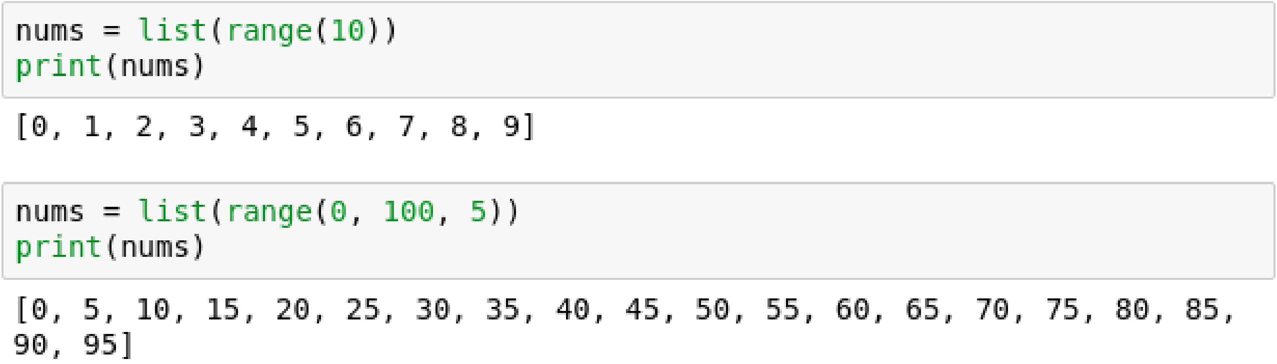


```
fruits.append("lime")    # add new item to list
print(fruits)
fruits.remove("orange")  # remove orange from list
print(fruits)
```

```
['apple', 'orange', 'apricot', 'banana', 'lime']
['apple', 'apricot', 'banana', 'lime']
```

Lists with integers

range() - a function that generates a sequence of numbers as a list



```
nums = list(range(10))
print(nums)
```

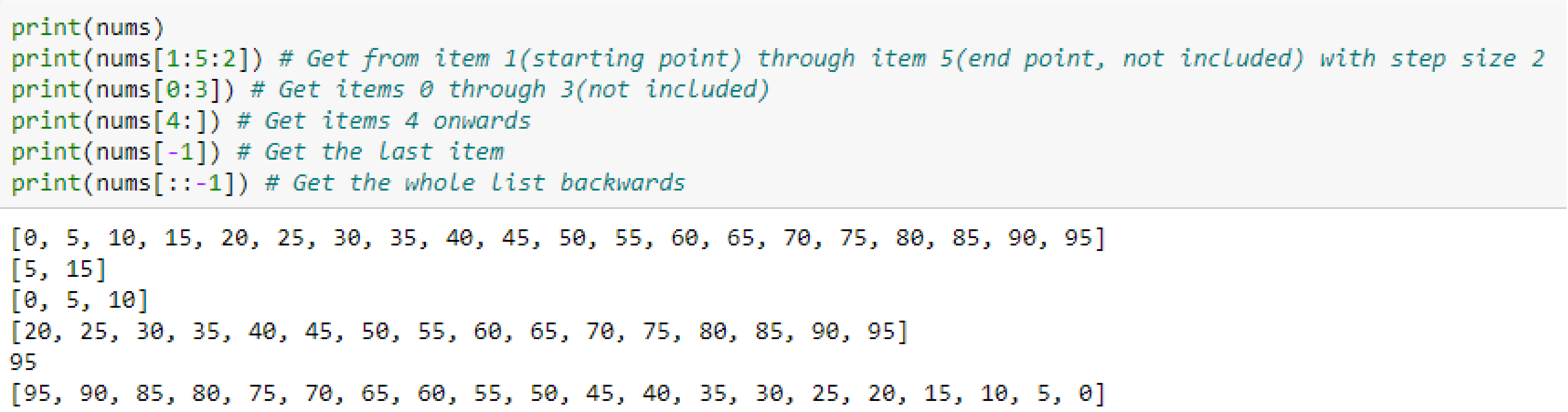
```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
nums = list(range(0, 100, 5))
print(nums)
```

```
[0, 5, 10, 15, 20, 25, 30, 35, 40, 45, 50, 55, 60, 65, 70, 75, 80, 85,
90, 95]
```

Slicing lists

- Slicing – obtain a particular set of sub-elements from a data structure.
- Very useful and flexible.

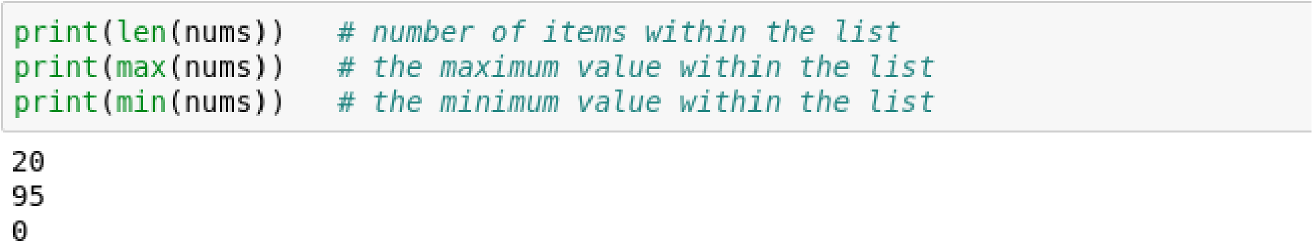


```
print(nums)
print(nums[1:5:2]) # Get from item 1(starting point) through item 5(end point, not included) with step size 2
print(nums[0:3]) # Get items 0 through 3(not included)
print(nums[4:]) # Get items 4 onwards
print(nums[-1]) # Get the last item
print(nums[::-1]) # Get the whole list backwards
```

```
[0, 5, 10, 15, 20, 25, 30, 35, 40, 45, 50, 55, 60, 65, 70, 75, 80, 85, 90, 95]
[5, 15]
[0, 5, 10]
[20, 25, 30, 35, 40, 45, 50, 55, 60, 65, 70, 75, 80, 85, 90, 95]
95
[95, 90, 85, 80, 75, 70, 65, 60, 55, 50, 45, 40, 35, 30, 25, 20, 15, 10, 5, 0]
```

Lists – helpful functions

- Makes them extremely useful and versatile

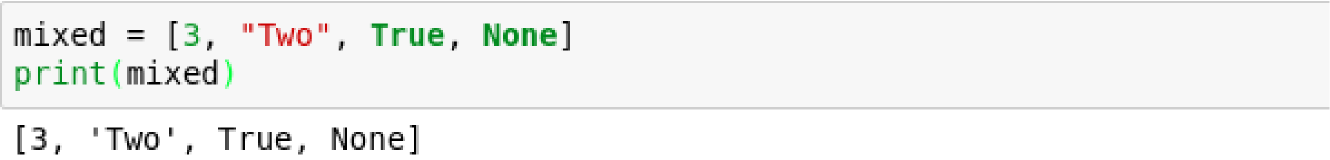


```
print(len(nums))   # number of items within the list
print(max(nums))   # the maximum value within the list
print(min(nums))   # the minimum value within the list
```

```
20
95
0
```

Lists can be of different types

- Not very useful, but possible



```python
mixed = [3, "Two", True, None]
print(mixed)
```

```
[3, 'Two', True, None]
```

Mutability

Mutable object – can be changed after creation.

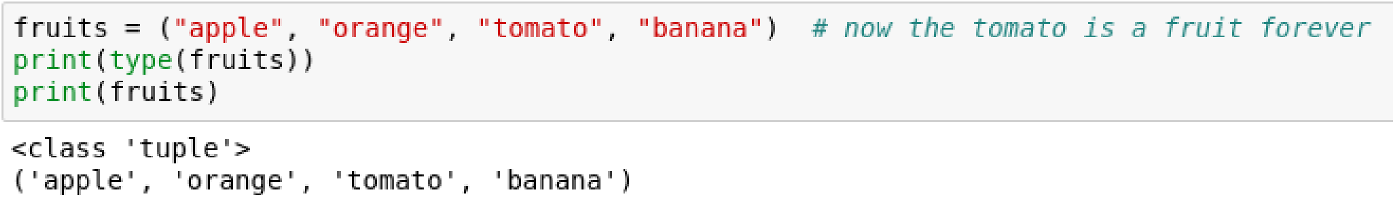
Immutable object - can **NOT** be changed after creation.

Quick quiz

- Are lists mutable?

Tuples

- Effectively lists that are immutable (I.e. can't be changed)

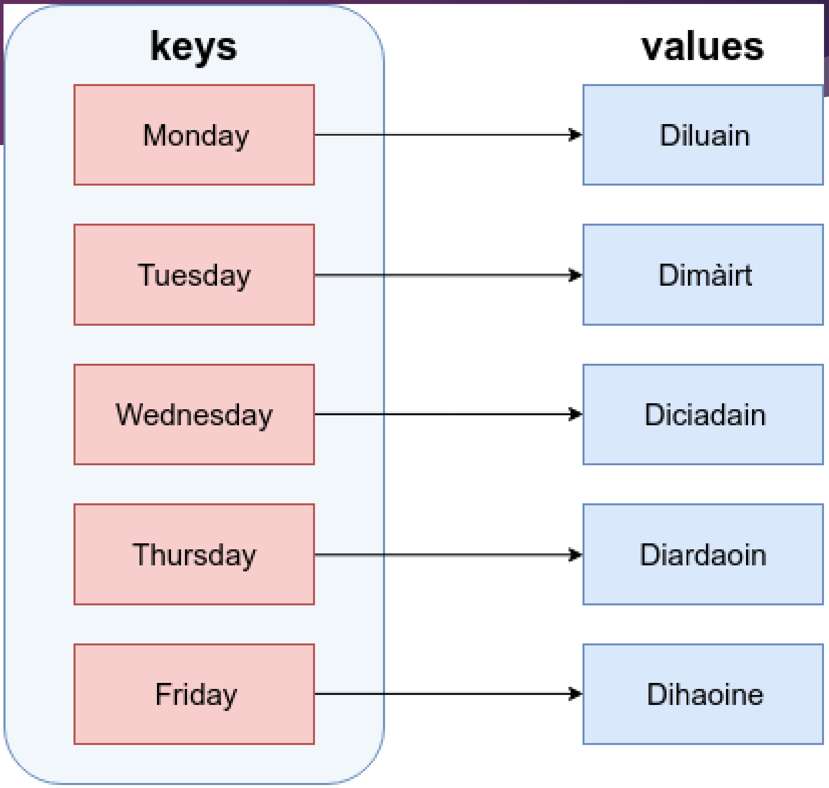


```
fruits = ("apple", "orange", "tomato", "banana")  # now the tomato is a fruit forever
print(type(fruits))
print(fruits)
```

```
<class 'tuple'>
('apple', 'orange', 'tomato', 'banana')
```

Dictionaries

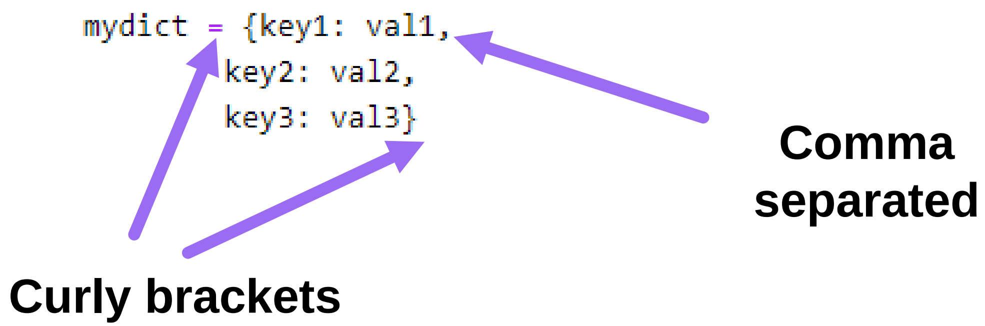
- Similar to actual dictionaries
- They are effectively 2 lists combined – keys and values
- We use the keys to access the values instead of indexing them like a list
- Each value is mapped to a unique key



keys	values
Monday	Diluain
Tuesday	Dimàirt
Wednesday	Diciadain
Thursday	Diardaoin
Friday	Dihaoine

Dictionary definition

Defined as comma separated **key : value** pairs:



```
mydict = {key1: val1,
          key2: val2,
          key3: val3}
```

Comma separated

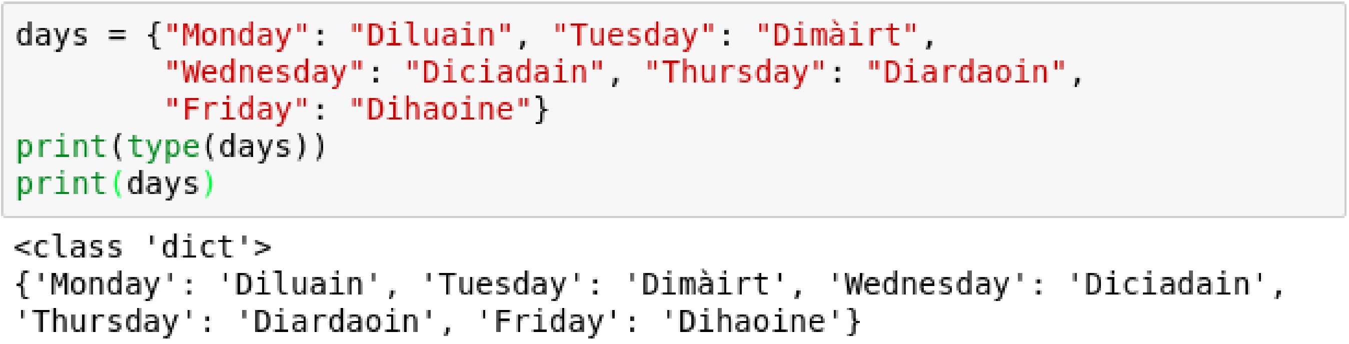
Curly brackets

Dictionary properties

- Values are mapped to a key
- Values are accessed by their key
- Key are unique and are immutable
- Values cannot exist without a key

Dictionaries

Let us define the one from the previous image

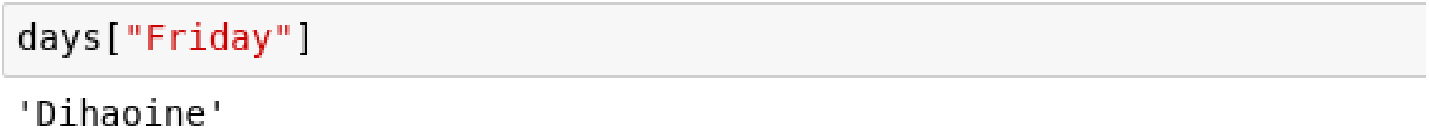


```
days = {"Monday": "Diluain", "Tuesday": "Dimàirt",
        "Wednesday": "Diciadain", "Thursday": "Diardaoin",
        "Friday": "Dihaoine"}
print(type(days))
print(days)
```

```
<class 'dict'>
{'Monday': 'Diluain', 'Tuesday': 'Dimàirt', 'Wednesday': 'Diciadain',
'Thursday': 'Diardaoin', 'Friday': 'Dihaoine'}
```

Accessing a dictionary

Values are accessed by their keys (just like a dictionary)



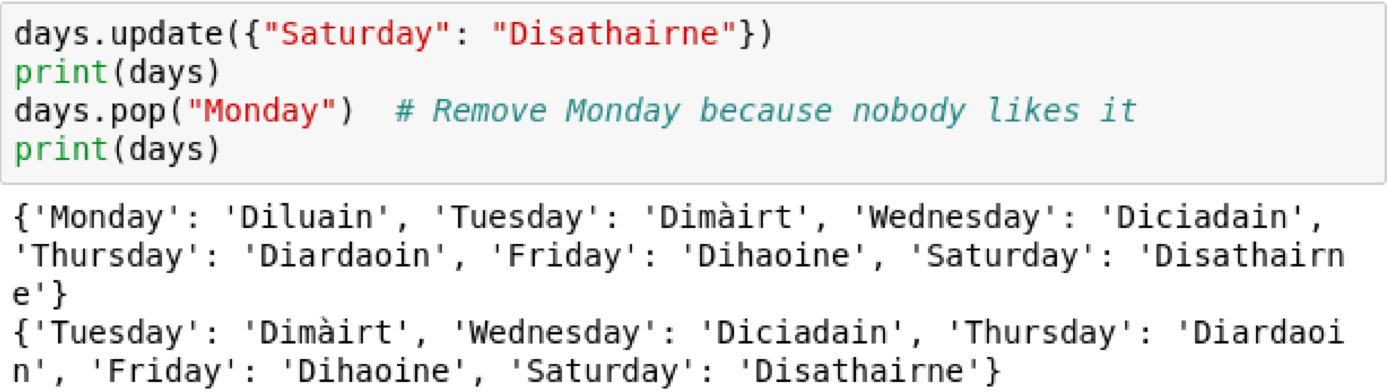
```
days["Friday"]
```

```
'Dihaoine'
```

Note that they can't be indexed like a list

Altering a dictionary

Can be done via the dictionary methods

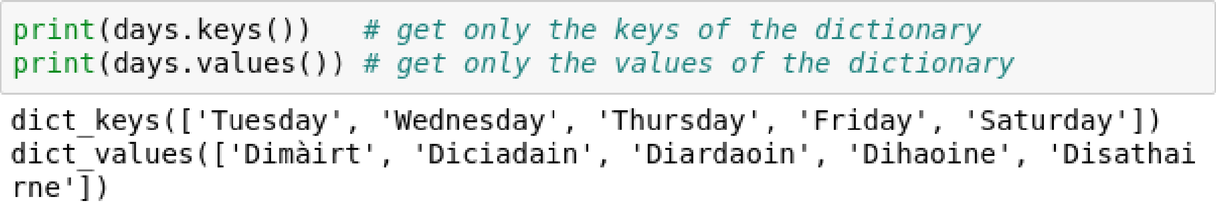


```
days.update({"Saturday": "Disathairne"})
print(days)
days.pop("Monday")  # Remove Monday because nobody likes it
print(days)
```

```
{'Monday': 'Diluain', 'Tuesday': 'Dimàirt', 'Wednesday': 'Diciadain',
'Thursday': 'Diardaoin', 'Friday': 'Dihaoine', 'Saturday': 'Disathairn
e'}
{'Tuesday': 'Dimàirt', 'Wednesday': 'Diciadain', 'Thursday': 'Diardaoi
n', 'Friday': 'Dihaoine', 'Saturday': 'Disathairne'}
```

Keys and Values

It is possible to obtain only the keys or values of a



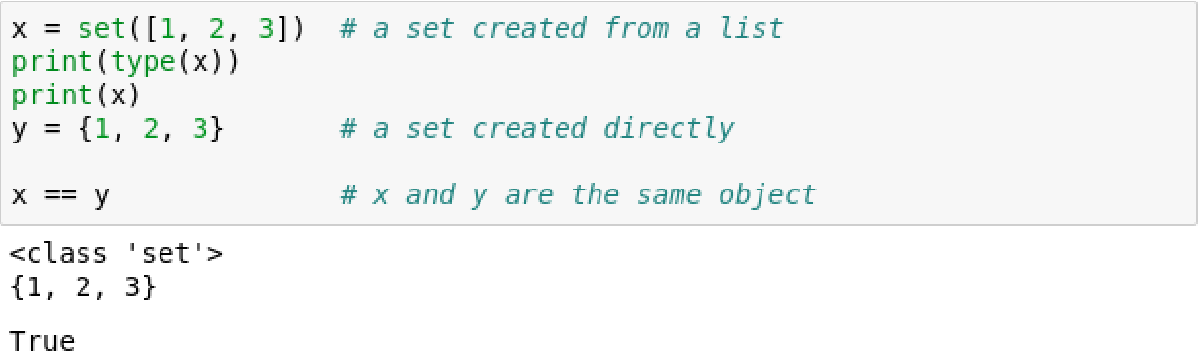
```python
print(days.keys())   # get only the keys of the dictionary
print(days.values()) # get only the values of the dictionary
```

```
dict_keys(['Tuesday', 'Wednesday', 'Thursday', 'Friday', 'Saturday'])
dict_values(['Dimàirt', 'Diciadain', 'Diardaoin', 'Dihaoine', 'Disathai
rne'])
```

This is useful for iteration.

Sets

- Effectively lists that can't contain duplicate items
- Similar functionality to lists
- Can't be indexed or sliced
- Can be created with **{}** or you can convert a list to a set

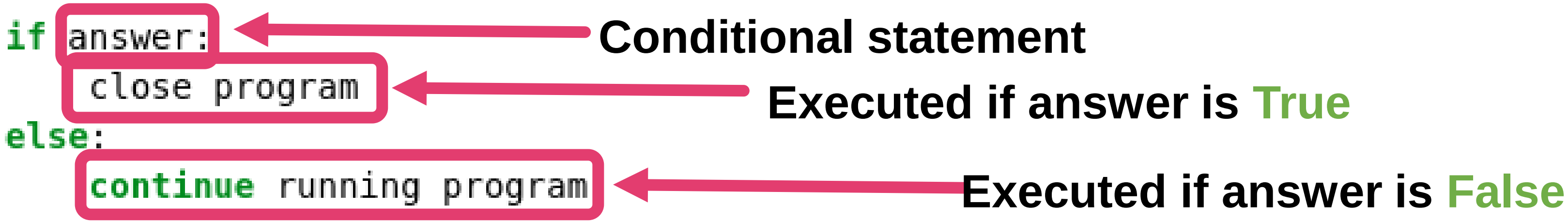


```
x = set([1, 2, 3])  # a set created from a list
print(type(x))
print(x)
y = {1, 2, 3}       # a set created directly

x == y              # x and y are the same object
```

```
<class 'set'>
{1, 2, 3}

True
```

If Else

- Fundamental building block of software



```
if answer:
    close program
else:
    continue running program
```

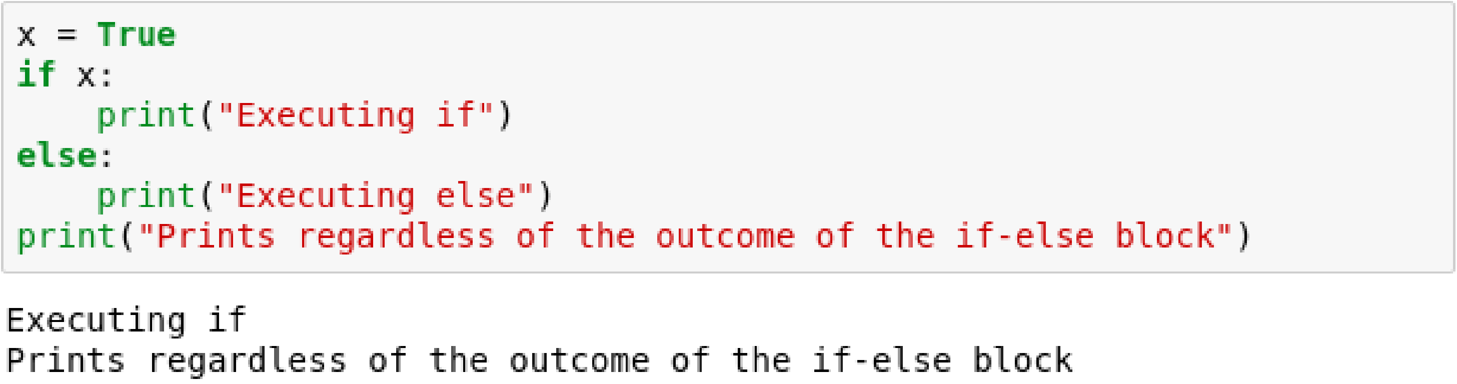
Conditional statement

Executed if answer is True

Executed if answer is False

If Else example

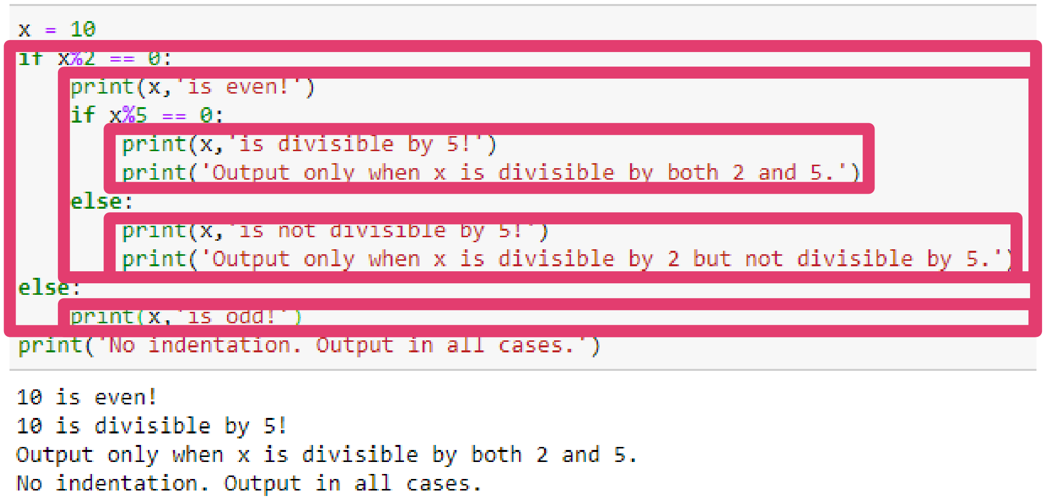
Try running the example below.
What do you get?



```
x = True
if x:
    print("Executing if")
else:
    print("Executing else")
print("Prints regardless of the outcome of the if-else block")
```

```
Executing if
Prints regardless of the outcome of the if-else block
```

Indentation matters!



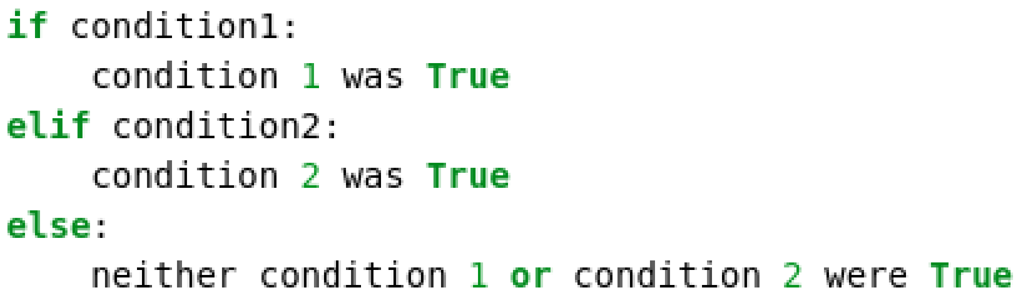
```
x = 10
if x%2 == 0:
    print(x,'is even!')
    if x%5 == 0:
        print(x,'is divisible by 5!')
        print('Output only when x is divisible by both 2 and 5.')
    else:
        print(x,'is not divisible by 5!')
        print('Output only when x is divisible by 2 but not divisible by 5.')
else:
    print(x,'is odd!')
print('No indentation. Output in all cases.')
```

```
10 is even!
10 is divisible by 5!
Output only when x is divisible by both 2 and 5.
No indentation. Output in all cases.
```

- 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

Extending if-else blocks

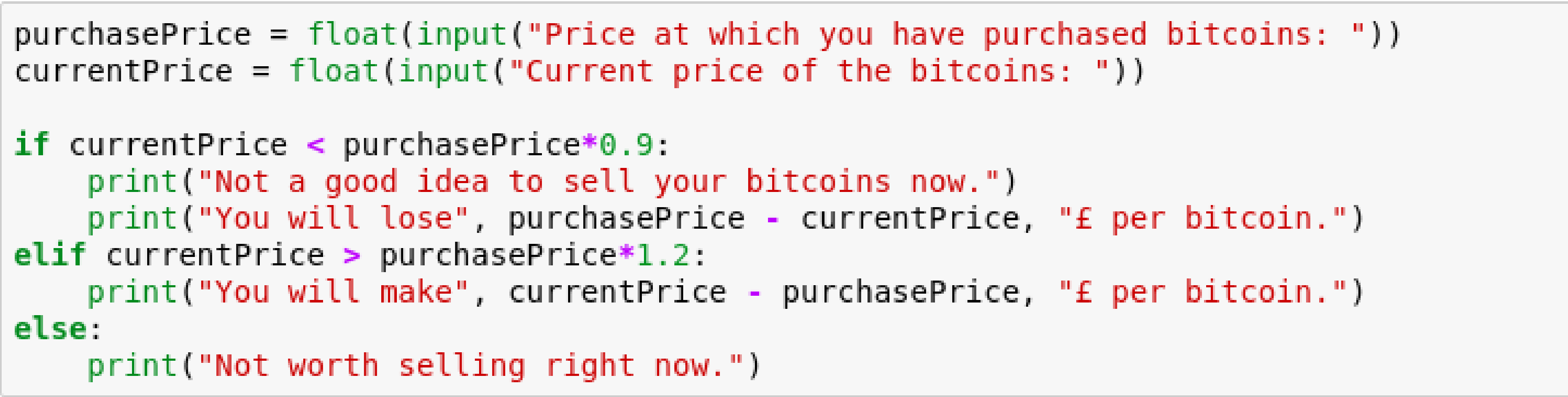
- We can add infinitely more if statements using **elif**



```
if condition1:
    condition 1 was True
elif condition2:
    condition 2 was True
else:
    neither condition 1 or condition 2 were True
```

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

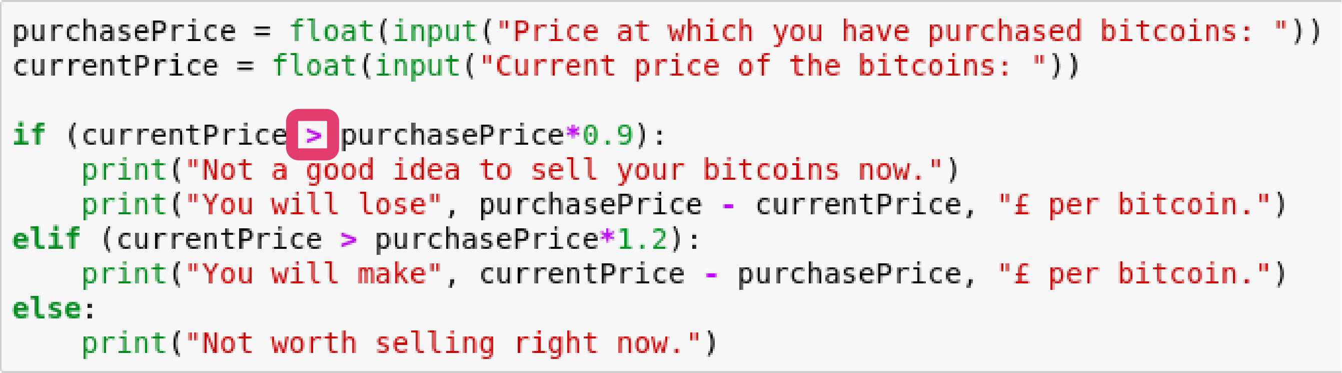
Bitcoin broker example



```
purchasePrice = float(input("Price at which you have purchased bitcoins: "))
currentPrice = float(input("Current price of the bitcoins: "))

if currentPrice < purchasePrice*0.9:
    print("Not a good idea to sell your bitcoins now.")
    print("You will lose", purchasePrice - currentPrice, "£ per bitcoin.")
elif currentPrice > purchasePrice*1.2:
    print("You will make", currentPrice - purchasePrice, "£ per bitcoin.")
else:
    print("Not worth selling right now.")
```

Quick quiz

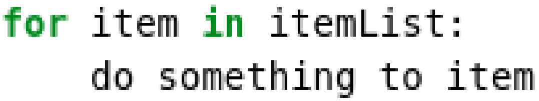
- What would happen if both conditions are True?



```
purchasePrice = float(input("Price at which you have purchased bitcoins: "))
currentPrice = float(input("Current price of the bitcoins: "))

if (currentPrice > purchasePrice*0.9):
    print("Not a good idea to sell your bitcoins now.")
    print("You will lose", purchasePrice - currentPrice, "£ per bitcoin.")
elif (currentPrice > purchasePrice*1.2):
    print("You will make", currentPrice - purchasePrice, "£ per bitcoin.")
else:
    print("Not worth selling right now.")
```

For loop

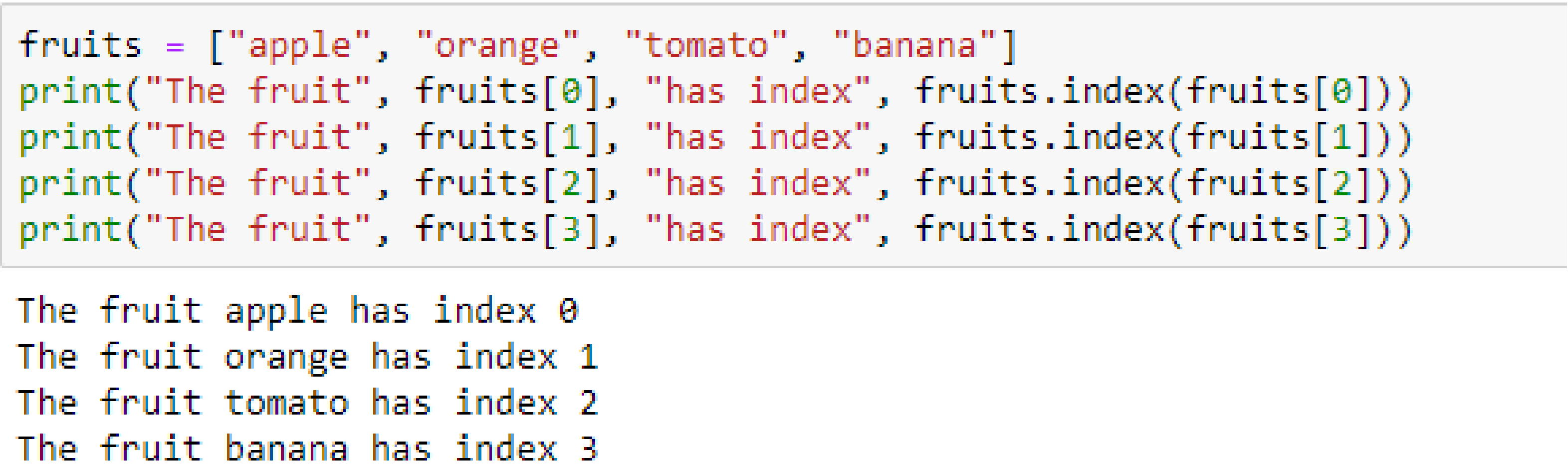
- Allows us to iterate over a set amount of variables within a data structure. During that we can manipulate each item however we want



```
for item in itemList:
    do something to item
```

- Again, indentation is important here!

Example



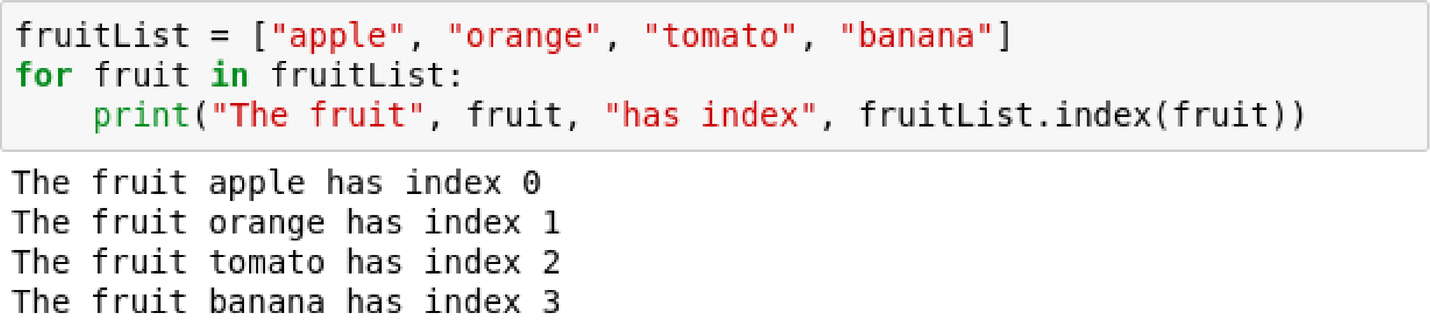
```
fruits = ["apple", "orange", "tomato", "banana"]
print("The fruit", fruits[0], "has index", fruits.index(fruits[0]))
print("The fruit", fruits[1], "has index", fruits.index(fruits[1]))
print("The fruit", fruits[2], "has index", fruits.index(fruits[2]))
print("The fruit", fruits[3], "has index", fruits.index(fruits[3]))
```

```
The fruit apple has index 0
The fruit orange has index 1
The fruit tomato has index 2
The fruit banana has index 3
```

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

For example

- Now with a for loop

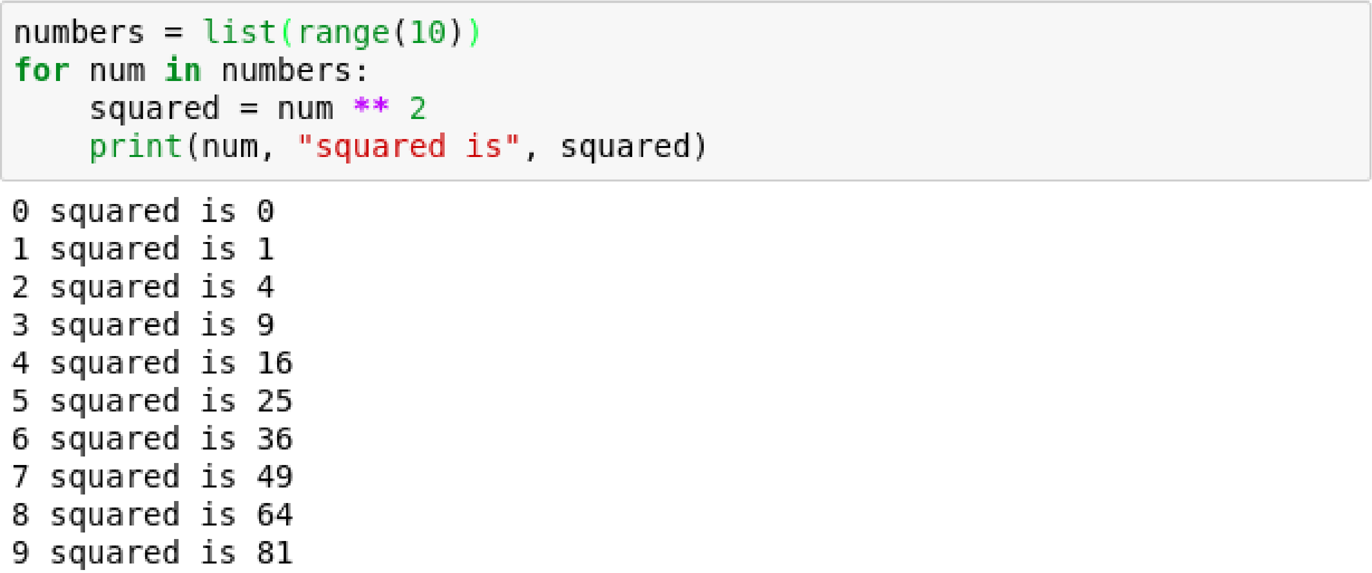


```
fruitList = ["apple", "orange", "tomato", "banana"]
for fruit in fruitList:
    print("The fruit", fruit, "has index", fruitList.index(fruit))
```

```
The fruit apple has index 0
The fruit orange has index 1
The fruit tomato has index 2
The fruit banana has index 3
```

- Saves us writing more lines
- Doesn't limit us in term of size

Numerical for loop

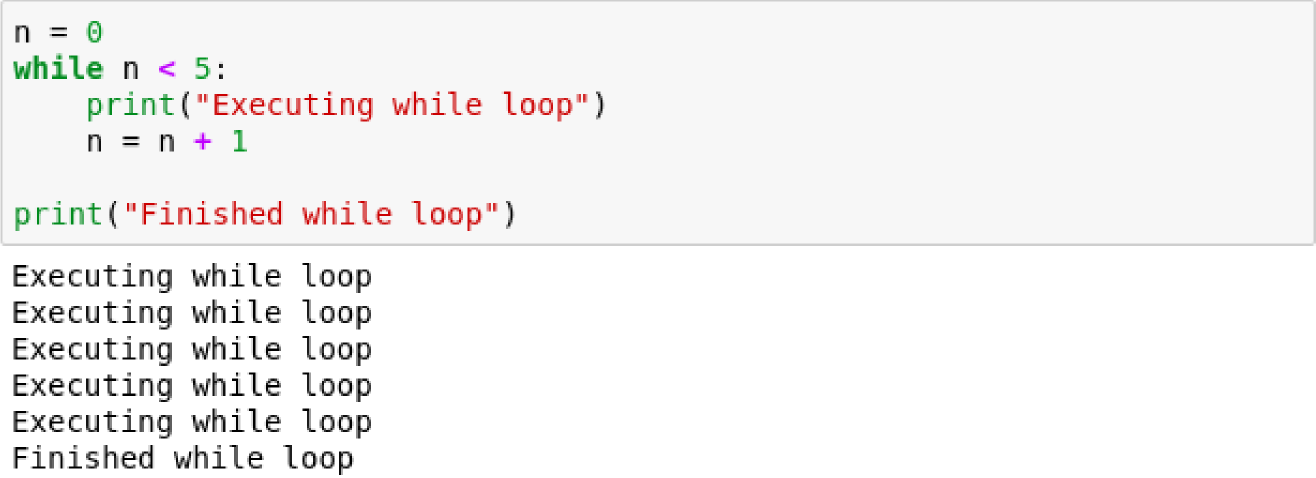


```python
numbers = list(range(10))
for num in numbers:
    squared = num ** 2
    print(num, "squared is", squared)
```

```
0 squared is 0
1 squared is 1
2 squared is 4
3 squared is 9
4 squared is 16
5 squared is 25
6 squared is 36
7 squared is 49
8 squared is 64
9 squared is 81
```

While loop

- 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



```
n = 0
while n < 5:
    print("Executing while loop")
    n = n + 1

print("Finished while loop")
```

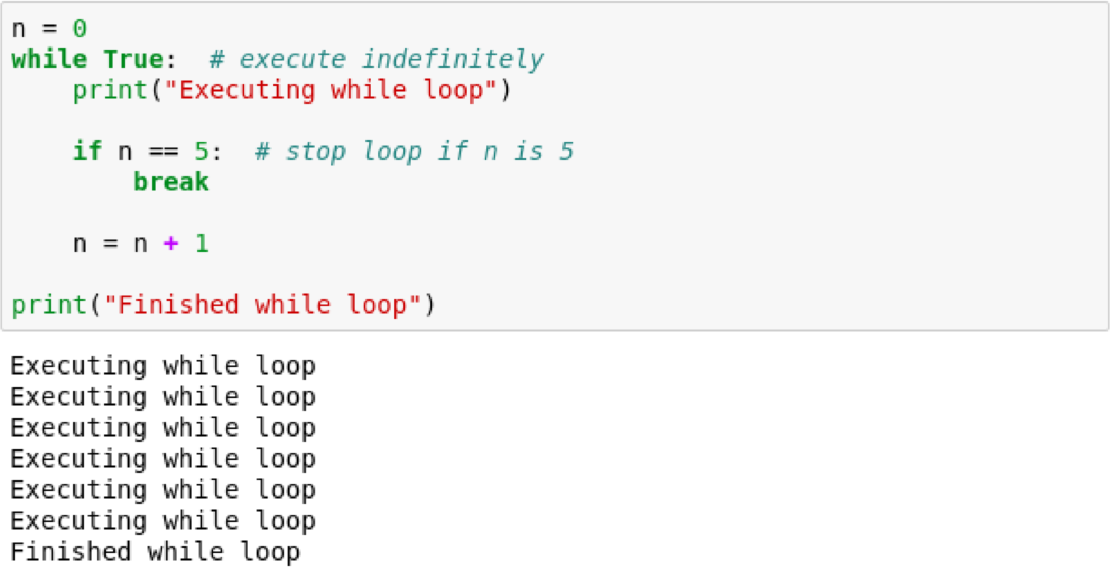
```
Executing while loop
Executing while loop
Executing while loop
Executing while loop
Executing while loop
Finished while loop
```

Break statement

- 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



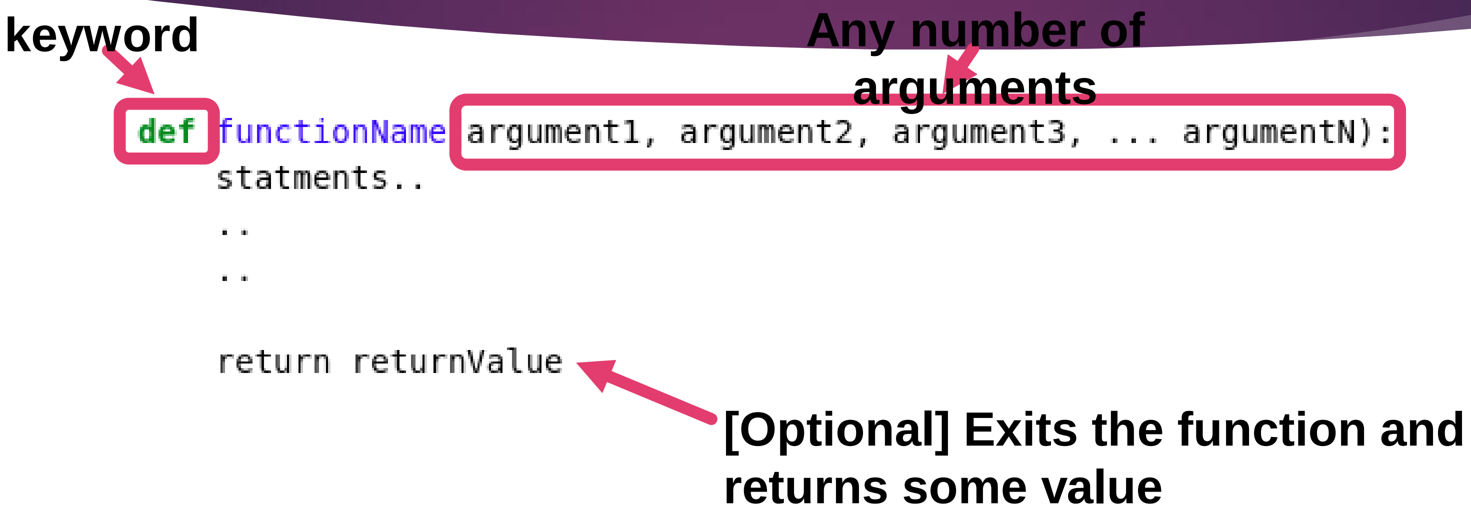
```
n = 0
while True:  # execute indefinitely
    print("Executing while loop")

    if n == 5:  # stop loop if n is 5
        break

    n = n + 1

print("Finished while loop")
```

```
Executing while loop
Executing while loop
Executing while loop
Executing while loop
Executing while loop
Executing while loop
Finished while loop
```

Functions

- Allow us to package functionality in a nice and readable way
- reuse it without writing it again
- Make code modular and readable
- Rule of thumb - if you are planning on using very similar code more than once, it may be worthwhile writing it as a reusable function.

Function declaration



keyword

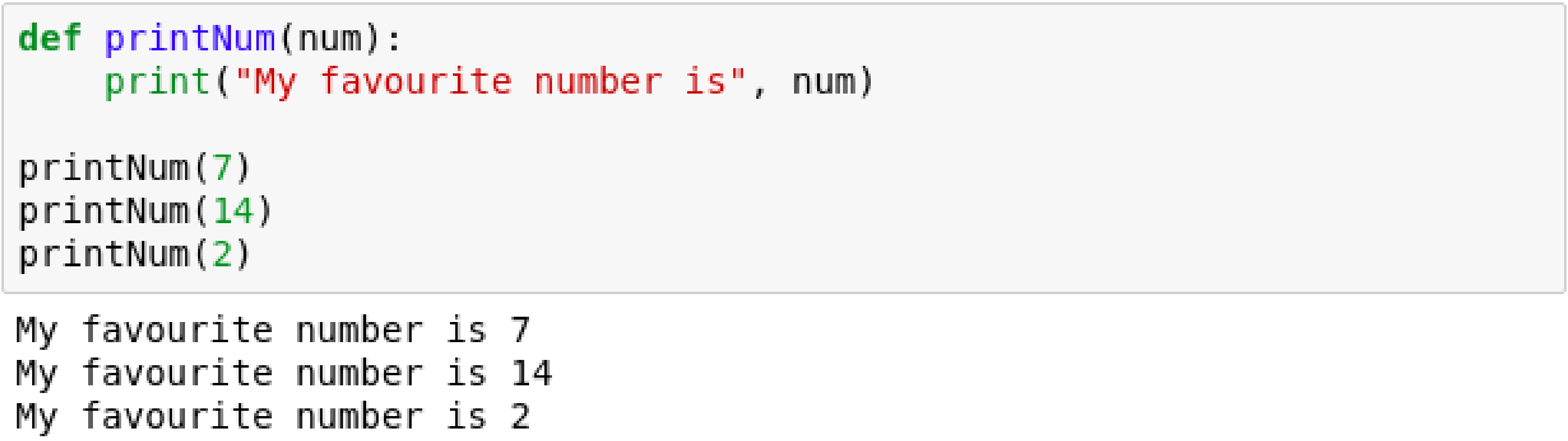
Any number of arguments

```
def functionName argument1, argument2, argument3, ... argumentN):
    statments..
    ..
    ..

    return returnValue
```

[Optional] Exits the function and returns some value

- Functions accept arguments and execute a piece of code
- Often they also return values (the result of their code)

Function example



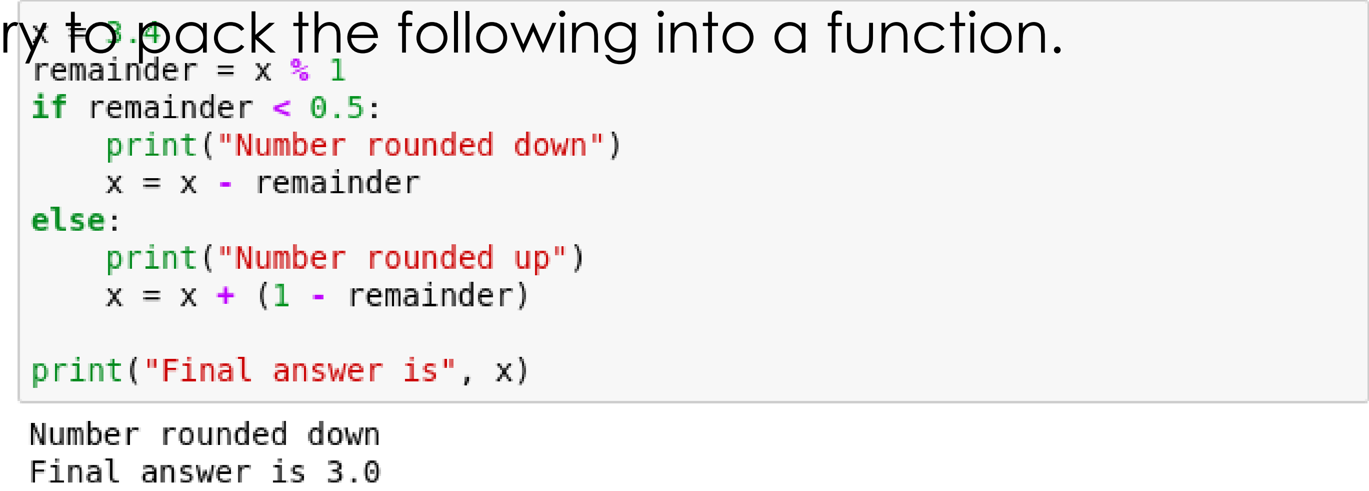
```
def printNum(num):
    print("My favourite number is", num)

printNum(7)
printNum(14)
printNum(2)
```

```
My favourite number is 7
My favourite number is 14
My favourite number is 2
```

Function example 2

We want to make a program that rounds numbers up or down.

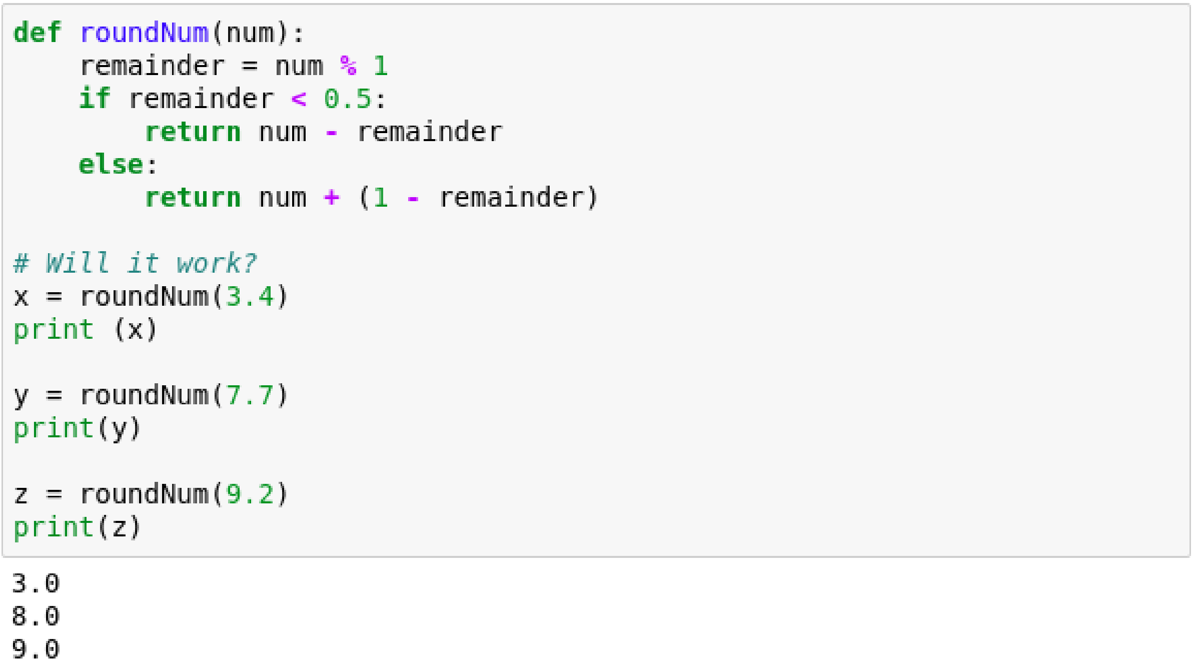
Try to pack the following into a function.



```
x = 3.4
remainder = x % 1
if remainder < 0.5:
    print("Number rounded down")
    x = x - remainder
else:
    print("Number rounded up")
    x = x + (1 - remainder)

print("Final answer is", x)
```

```
Number rounded down
Final answer is 3.0
```

Function example 2

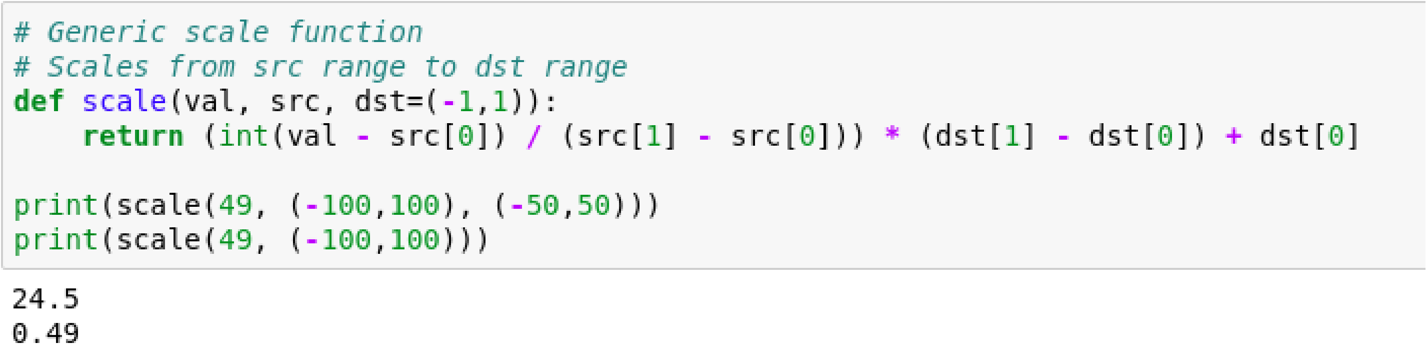


```python
def roundNum(num):
    remainder = num % 1
    if remainder < 0.5:
        return num - remainder
    else:
        return num + (1 - remainder)

# Will it work?
x = roundNum(3.4)
print (x)

y = roundNum(7.7)
print(y)

z = roundNum(9.2)
print(z)
```

```
3.0
8.0
9.0
```

Function example 3

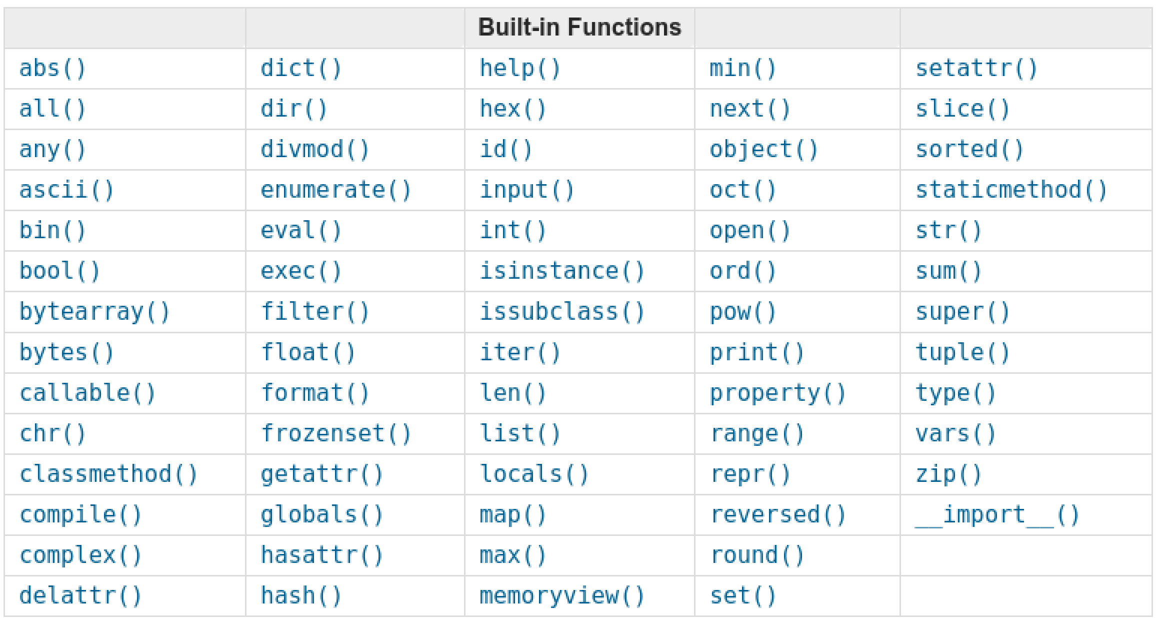
$$(val - src[0]) \times \frac{dst[1] - dst[0]}{src[1] - src[0]} - dst[0]$$



```
# Generic scale function
# Scales from src range to dst range
def scale(val, src, dst=(-1,1)):
    return (int(val - src[0]) / (src[1] - src[0])) * (dst[1] - dst[0]) + dst[0]

print(scale(49, (-100,100), (-50,50)))
print(scale(49, (-100,100)))
```

```
24.5
0.49
```

Python built-in functions



		Built-in Functions		
abs()	dict()	help()	min()	setattr()
all()	dir()	hex()	next()	slice()
any()	divmod()	id()	object()	sorted()
ascii()	enumerate()	input()	oct()	staticmethod()
bin()	eval()	int()	open()	str()
bool()	exec()	isinstance()	ord()	sum()
bytearray()	filter()	issubclass()	pow()	super()
bytes()	float()	iter()	print()	tuple()
callable()	format()	len()	property()	type()
chr()	frozenset()	list()	range()	vars()
classmethod()	getattr()	locals()	repr()	zip()
compile()	globals()	map()	reversed()	__import__()
complex()	hasattr()	max()	round()	
delattr()	hash()	memoryview()	set()	

To find out how they work: https://docs.python.org/3.3/library/functions.html

Running Python Programs Interactively

Suppose the file `script.py` contains the following lines:

```
print 'Hello world'
x = [0,1,2]
```

Let's run this script in each of the ways described on the last slide:

- python

```
>>> import script  # DO NOT add the .py suffix.  Script is a module here
>>> x
    Traceback (most recent call last):
    File "<stdin>", line 1, in ?
    NameError: name 'x' is not defined
>>> script.x      # to make use of x, we need to let Python know which
           #module it came from, i.e. give Python its context
    [0,1,2]
```

File naming conventions

- python files usually end with the suffix `.py`
- but executable files usually don't have the `.py` extension
- modules (later) should <u>always</u> have the `.py` extension

References

- Python Homepage
  - http://www.python.org
- Python Tutorial
  - http://docs.python.org/tutorial/
- Python Documentation
  - http://www.python.org/doc
- Python Library References
  - http://docs.python.org/release/2.5.2/lib/lib.html
- Python Add-on Packages:
  - http://pypi.python.org/pypi