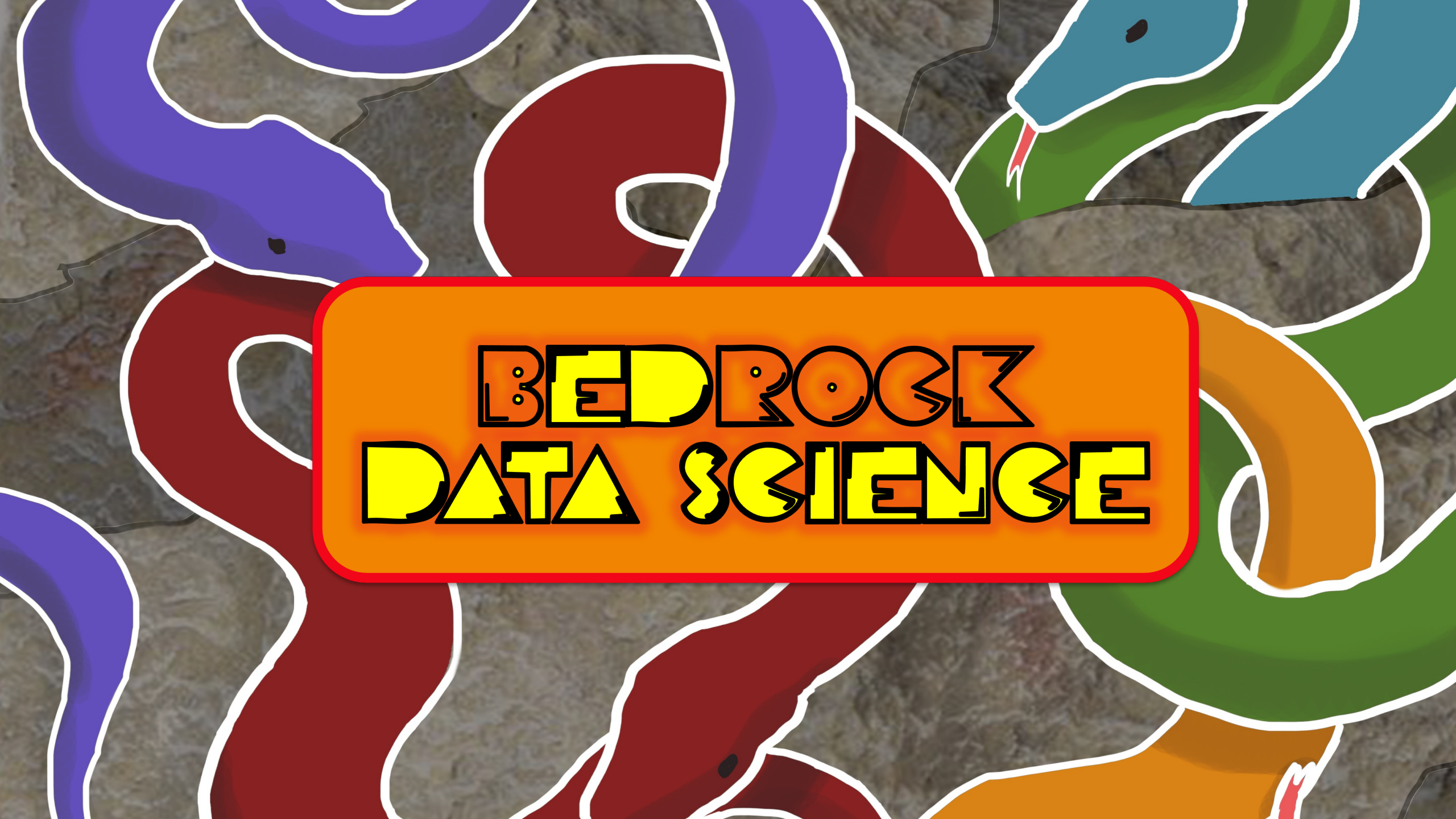

BEDROCK
DATA SCIENCE

Previously on BRDS

RECAP: Dictionaries in Python

In Python, dictionaries are **ordered** collection of **mutable objects with immutable keys.**
Elements within a dictionary have a key & value.

$$Dict = \{Key_1:Value_{1,\ldots,}\ Key_n:Value_n\}$$

```
>>> dictA = {'India':'New Delhi', 'USA':'Washington
DC','Germany':'Berlin','Sri Lanka':'Colombo’}

>>> dictB = {'Apples':1, 'Pineapple':4, 'Grapes’:3}

>>> print(dictA)
{'India': 'New Delhi', 'USA': 'Washington DC', 'Germany': 'Berlin',
'Sri Lanka': 'Colombo’}

>>> print(dictB)
{'Apples': 1, 'Pineapple': 4, 'Grapes': 3}
```

DictA

Key - Type	Value - Type
India<str>	New Delhi<str>
USA<str>	Washington DC<str>
Germany<str>	Berlin<str>

DictB

Key - Type	Value - Type
Apples<str>	1 <int>
Pineapple<str>	4 <int>
Grapes <str>	3 <int>

RECAP: List Comprehension

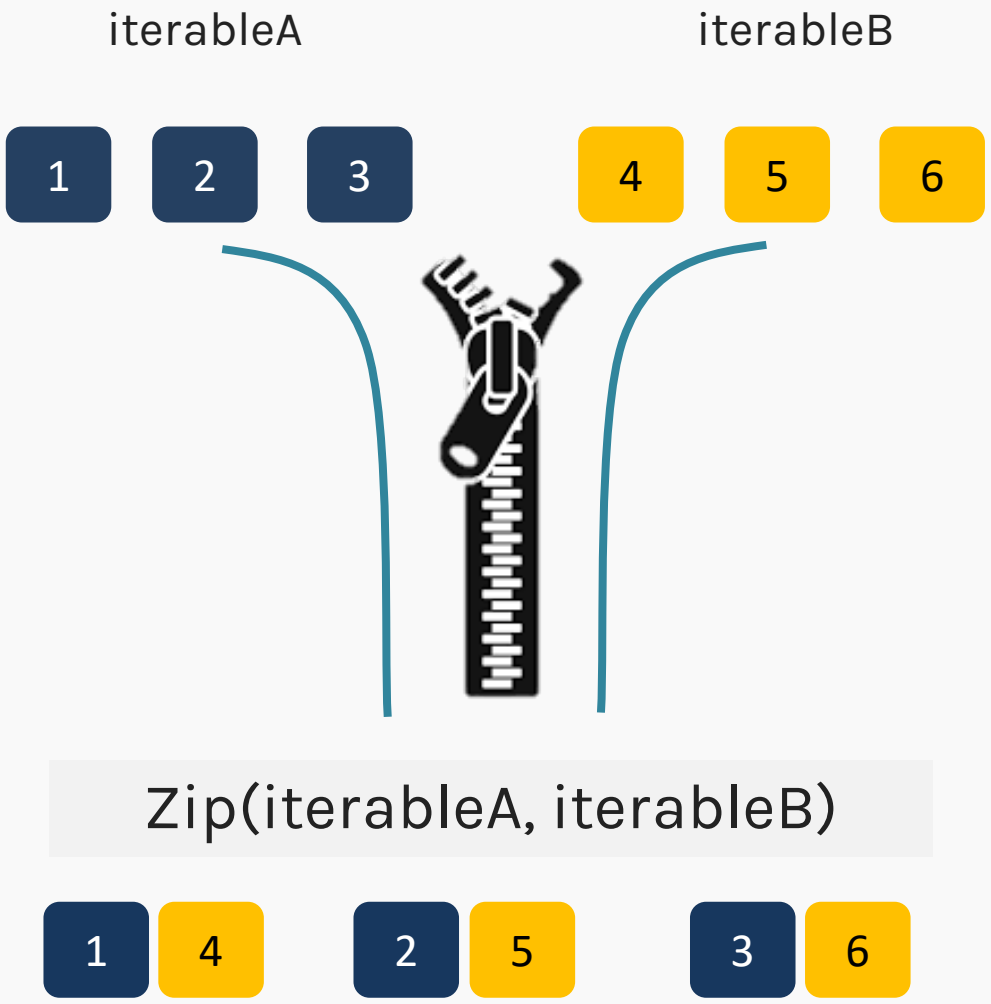
List Comprehensions: A Pythonic way for making lists and loops.

```
List = [expression for item in iterable]
```

```
List = [expression for item in iterable if conditional]
```

```
List = [expression1 (if conditional) else expression2 for
                    item in iterable]
```

Zip



Python's zip() function creates an iterator that will aggregate elements from two or more iterables.

```
>>> letters = ['a','b','c','d']
>>> numbers = [1,2,3,4]

>>> for letter, number in
zip(letters, numbers):
...     print(letter,number)
'a' 1
'b' 2
'c' 3
'd' 4
```

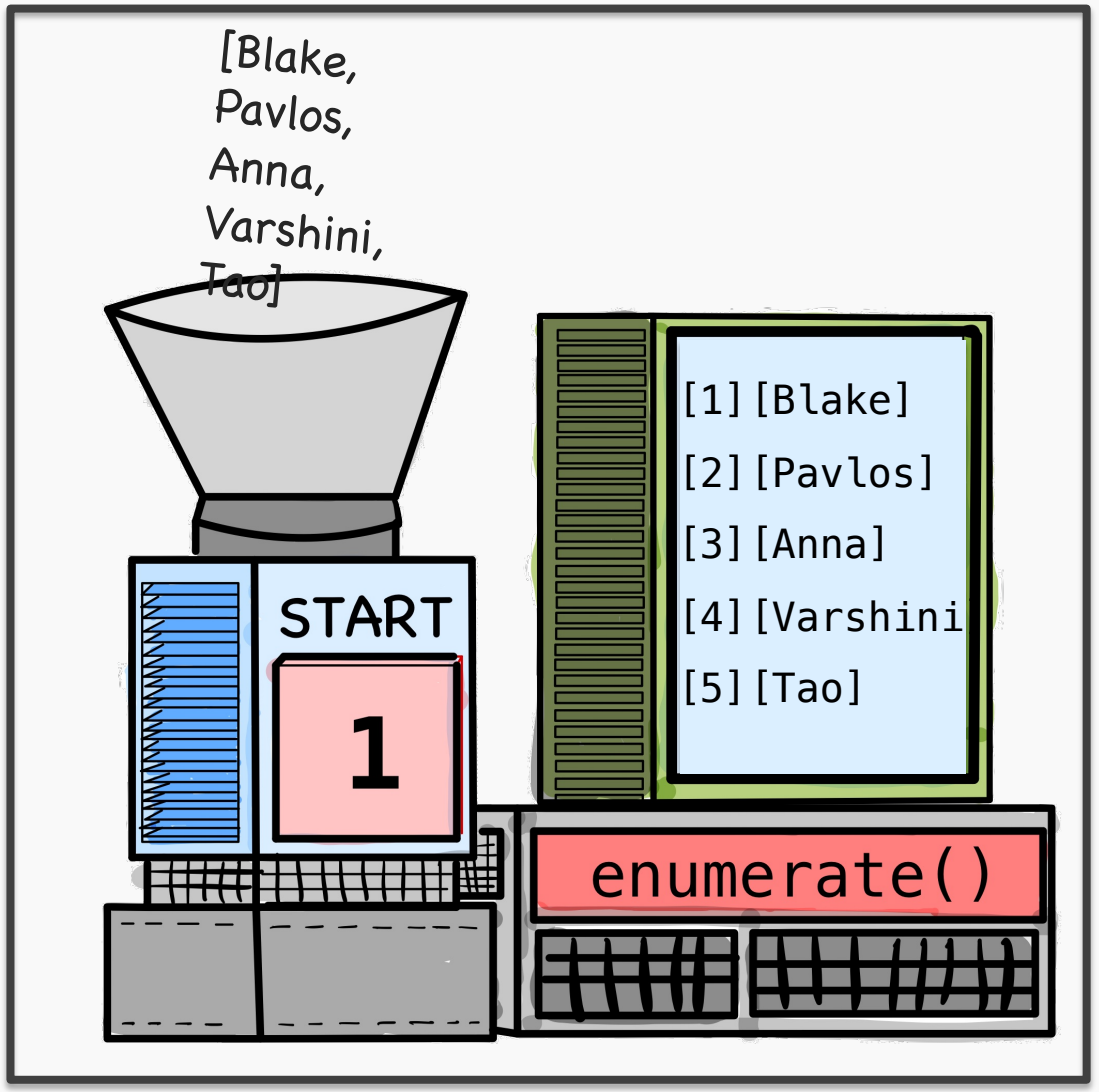
Enumerate

```
SYNTAX: for count,value in enumerate(values):
   ...:      Do something
```

- When you use enumerate(), the function gives you back *two* loop variables:
  - The **count** of the current iteration
  - The **value** of the item at the current iteration

- The use of two loop variables i.e count and value, is an example of argument unpacking.

- You can adjust the start count of the index using the **`start`** argument.



Lists

Dictionaries

Strings

Functions

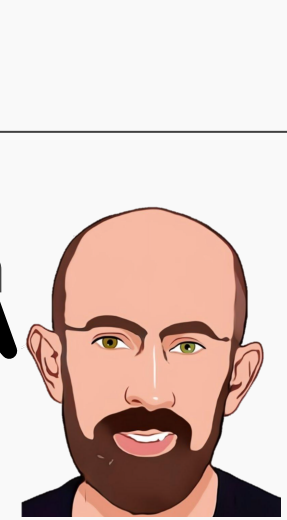
Classes

Functions

Outline

- Introduction to functions
  - Motivation
  - Defining function
  - Inputs and return
- Python function essentials
  - Built-in functions
  - Custom functions
  - Namespaces
- Advanced topics
  - Lambda functions
  - Functions within functions
  - Function decorators

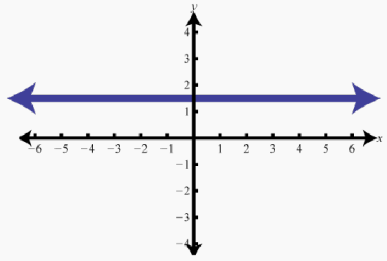
What is a function?



- A function is the same concept as was taught in maths, a unique mapping.

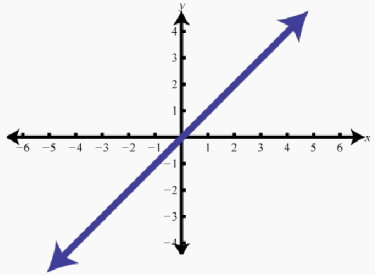
Constant Function

$f(x) = c$



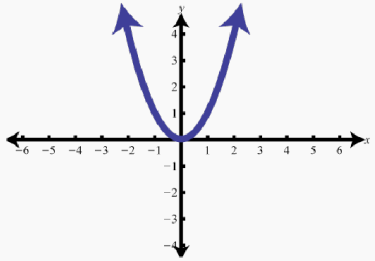
Identity Function

$f(x) = x$



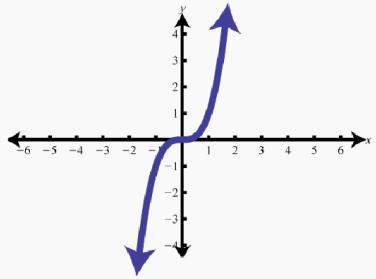
Squaring Function

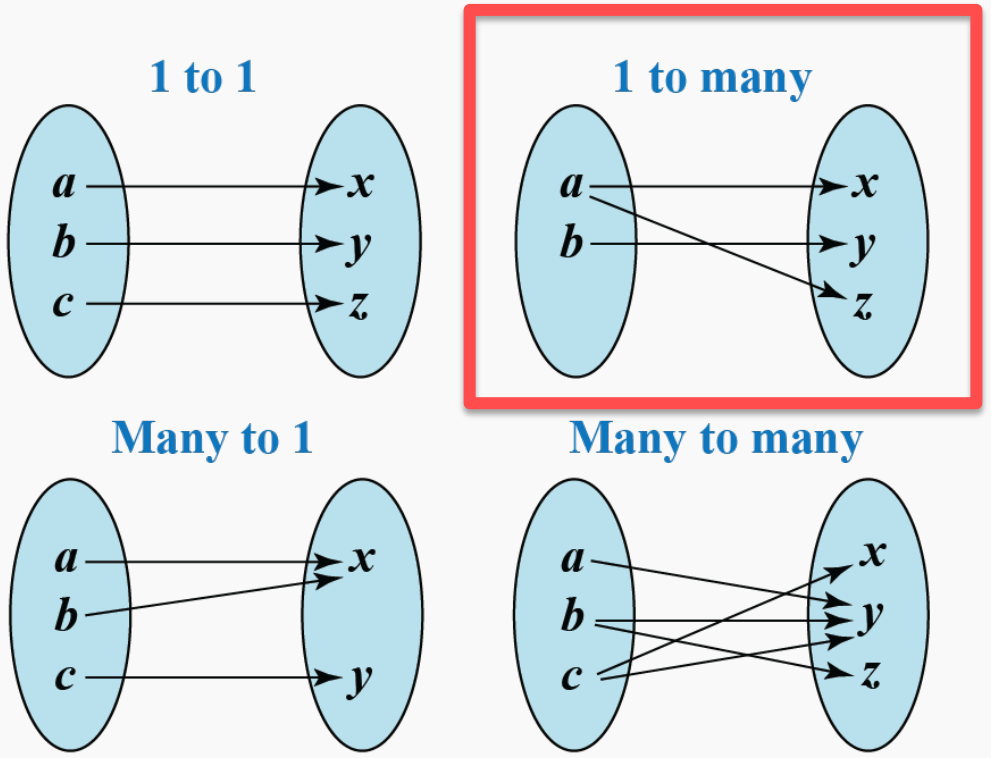
$f(x) = x^2$



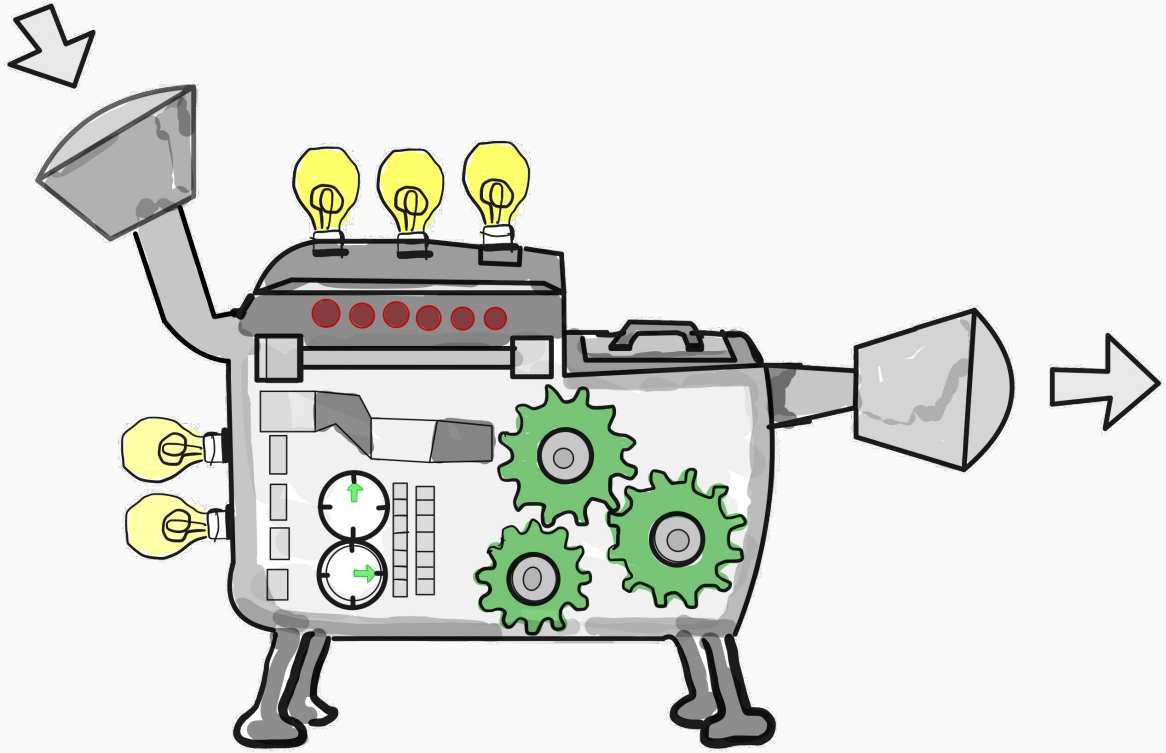
Cubing Function

$f(x) = x^3$

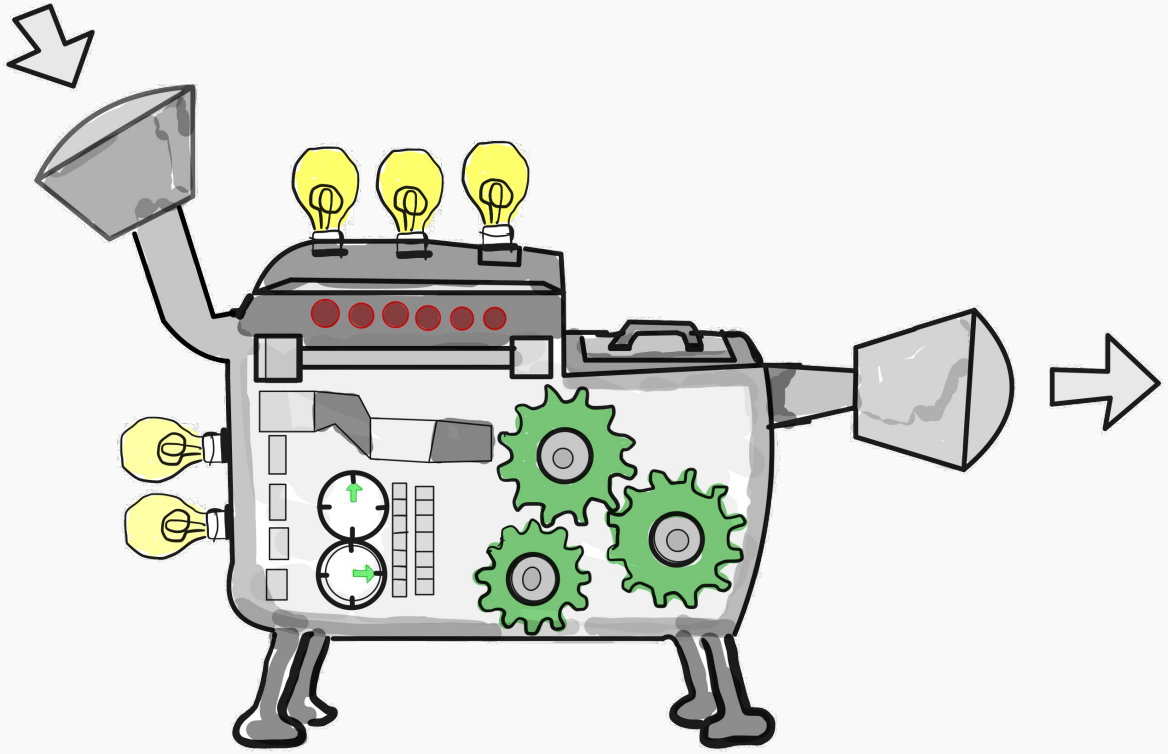


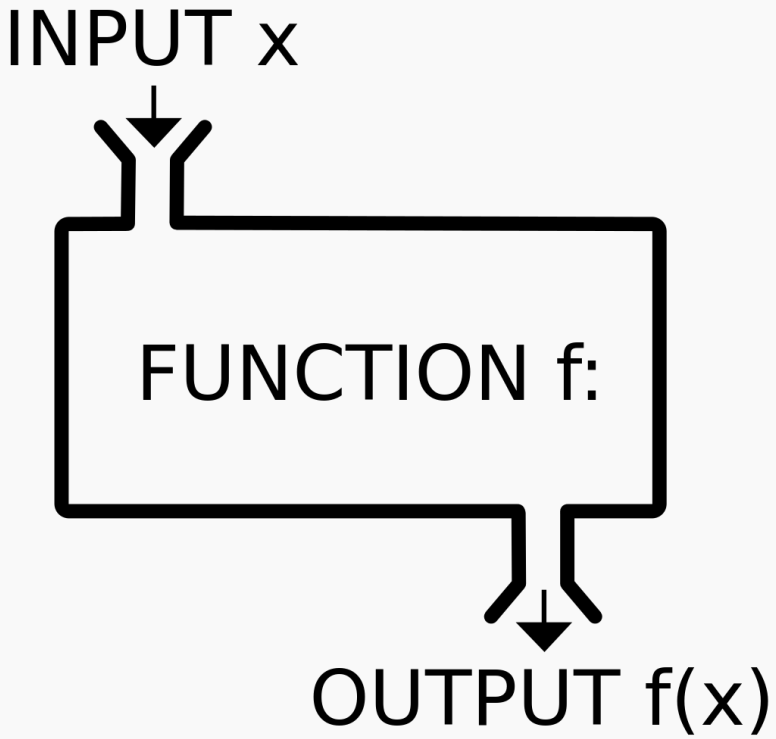


What is a function?



What is a function?





Built-in Functions

built-in Functions

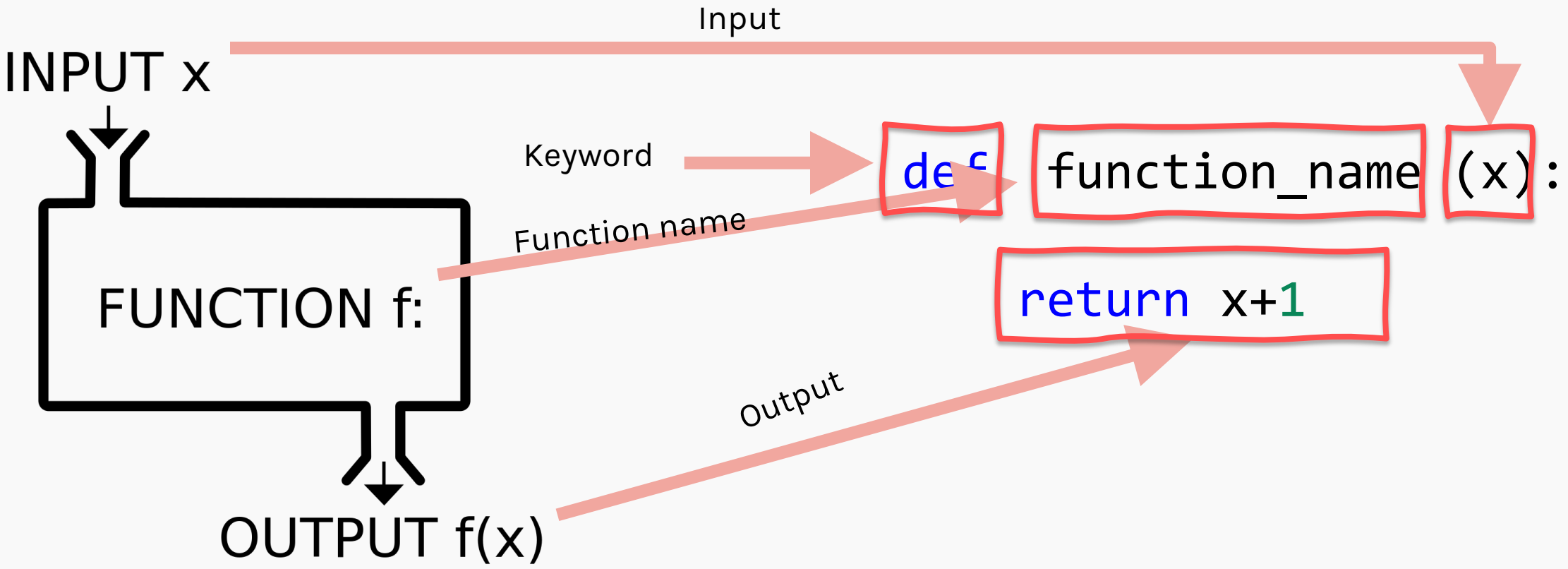
Function	Description
input()	Reads a line from the input
id()	Address of the object in memory
abs()	Returns the absolute value of a number
all()	Returns True if all items in an iterable object are true
any()	Returns True if any item in an iterable object is true
print()	displays a string to the console
bin()	Returns the binary version of a number
bool()	Returns the boolean value of the specified object
chr()	Returns a character from the specified Unicode code.

You can find more in-built functions here: https://docs.python.org/3/library/functions.html

Anatomy of a function

Anatomy of a function

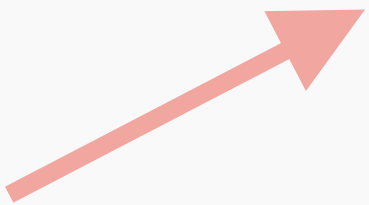
- In code a function works in similar manner, follow the following syntax in analogy to the ‘maths’ functions



Why use functions?

Reason #477: To avoid repetition

```
sum = 1+2+3
mean = sum/3

sum = 4+5+6
mean = sum/3

...
```



```
def mean(a,b,c):
    sum = a + b + c
    return sum/3
```

```
mean(1,2,3)
mean(4,5,6)
...
```

Reason #722: As a more general approach

```
def sum(lst):
    sum=0
    for i in lst:
        sum = sum+i
    return sum
```

```
def mean(lst):
    return sum(lst)/len(lst)
```

```
mean([1,2,3,4,56])
mean([6,5,7])
mean([7,45,23,11,64,86,35])
```

Input arguments to a function

- Pass input arguments while declaring the function and use them within the function.

```
In [2]: def quadratic(a,b,c):
   ...:     root1 = (-b + (b**2 - 4*a*c)**0.5)/(2*a)
   ...:     root2 = (-b - (b**2 - 4*a*c)**0.5)/(2*a)
   ...:     return (root1,root2)
   ...:

In [3]: quadratic(1,1,1)
Out[3]: (0.6180339887498949, -1.618033988749895)

In [4]: quadratic(1,-1,1)
Out[4]: (1.618033988749895, -0.6180339887498949)
```

Function declaration

Function call

Function declaration

The input arguments to be used in the function

```
In [2]: def quadratic(a,b,c):
   ...:     root1 = (-b + (b**2 - 4*a*c)**0.5)/(2*a)
   ...:     root2 = (-b - (b**2 - 4*a*c)**0.5)/(2*a)
   ...:     return (root1,root2)
   ...:

In [3]: quadratic()
---------------------------------------------------------------------
TypeError                                 Traceback
(most recent call last)
<ipython-input-7-9b94de133a5a> in <module>
----> 1 quadratic()


TypeError: quadratic() missing 3 required positional
arguments: 'a', 'b', and 'c'
```

Function declaration

Remember to put the default arguments after the non-default ones

```
In [2]: def quadratic(a=1,b=0,c=-1):
   ...:     root1 = (-b + (b**2 - 4*a*c)**0.5)/(2*a)
   ...:     root2 = (-b - (b**2 - 4*a*c)**0.5)/(2*a)
   ...:     return (root1,root2)
   ...:

In [9]: quadratic()
Out[9]: (1.0, -1.0)
```

Set a default value, while declaring input arguments

In case no value is passed to the function it will consider the default value

Function call

```
In [9]: quadratic()
Out[9]: (1.0, -1.0)
```

default values

```
In [10]: quadratic(1,0,-4)
Out[10]: (2.0, -2.0)
```

positional arguments

Remember to put the keyword arguments after the positional ones

```
In [11]: quadratic(c=-4,b=0,a=1)
Out[11]: (2.0, -2.0)
```

keyword arguments

While calling a function, you can pass arguments in order, or pass them by keyword

Summary: Functions 101

- Pass input arguments while declaring the function and use them within the function.

```
def sum(a,b,c):
```

Function call

```
sum(1,2,3)
```

- Set a default value, wh… case no value is passe… default value

Remember to put the default arguments after the non-default ones

You can change the default argument as a normal call results(2,10,'home')

```
def results(matches, goals, match_type='away'):
```

Function call

```
results(2,10)
```

- Set an expected type(Fix return type as well). The expected types are more as a reference as they are not enforced.

```
def results(matches : int, goals : int, match_type='away') ->str:
```

Function call

```
results(2.5,11.5)
```

Summary: Functions 101

- Keyword arguments, to assign the input argument of function using it's name

Function call → `results(goals = 10, matches = 5)`

Function call → `results(matches = 5, goa`

- Unknown number of arguments

```
def sum(*args):
    return arg[2]-arg[1]/arg[0]
```

Function call → `sum(17,2,3,4,1,2)`

You can pass more but not less than 3 inputs

`sum(17,2)` ✖

- Unknown number of keyword arguments

```
def goalpermatch(**kwargs):
    gpm = kwargs['goals']/kwargs['matches']
    return gpm
```

Function call → `goalpermatch(goals=2,matches=1)`

You have to pass atleast 'goals' and 'matches' keywords

More * and **

- * and ** can also be used while calling a function.
- * unpacks element from a list/tuple/iterable

```
def root(a,b,c):
    root = (-b + (b**2 - 4*a*c)**0.5)/2*a
    return root
lst = [5,6,7]
root(*lst)
>>>
(-14.999999
```

- ** unpacks elemen assign keys to keywords

Useful when order of arguments is not known while calling the function

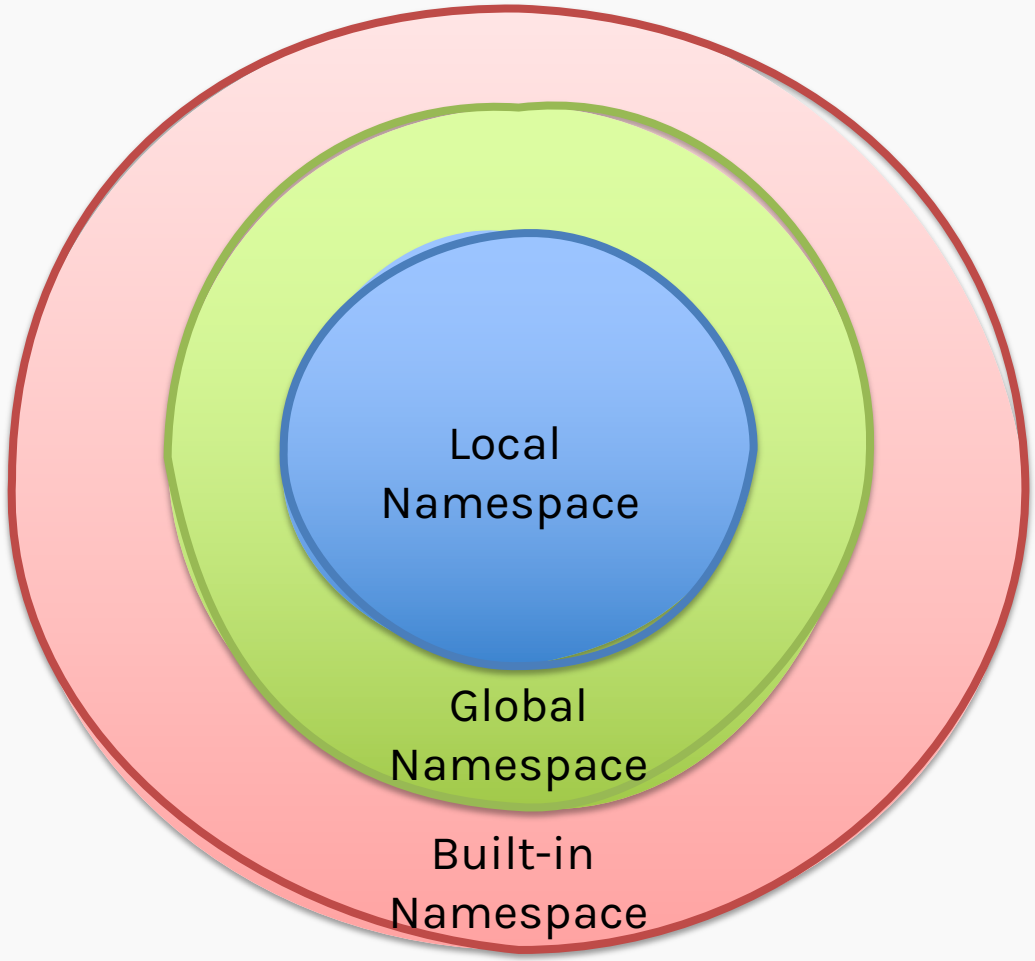
```
dict = {'b':6,'c':7,'a':5}
root(**dict)
>>>
(-14.999999999999998+25.495097567963924j)
```

More on * here

Function Scope

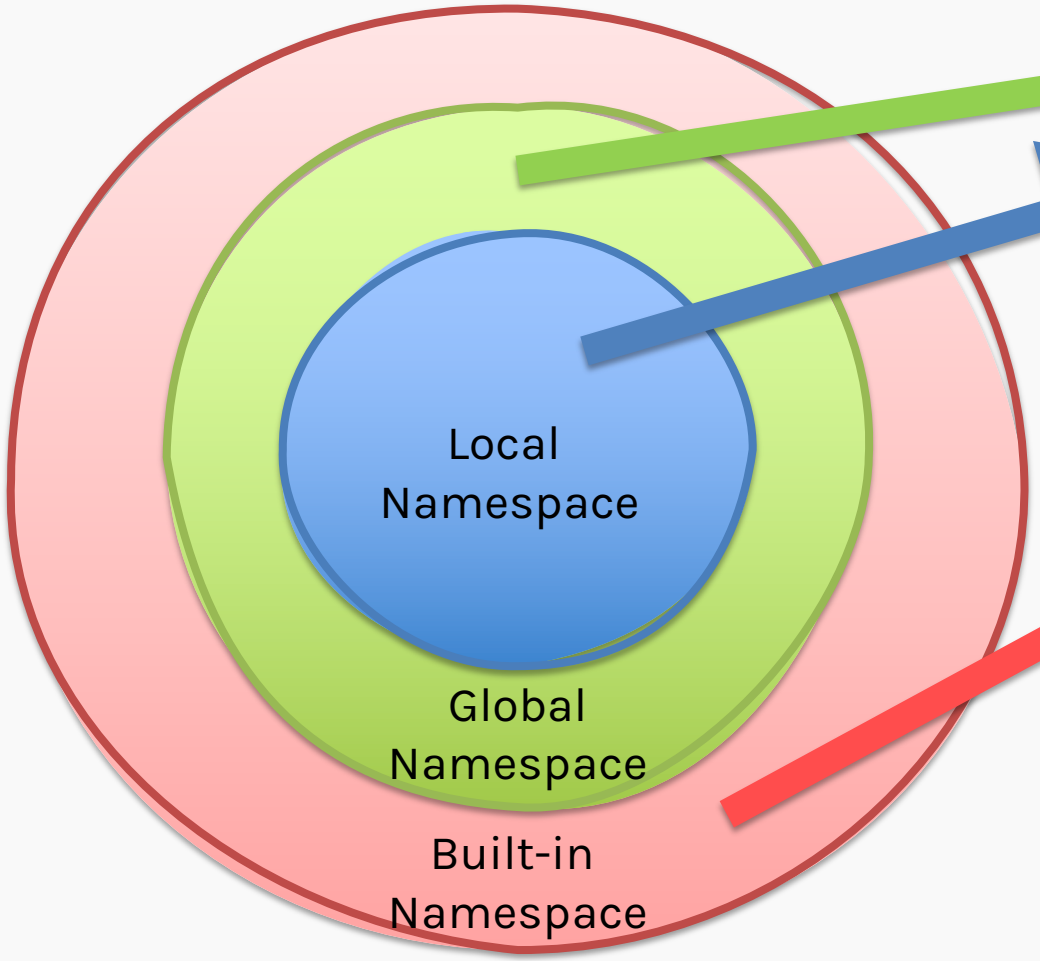
Namespaces

- Namespaces are boundaries or scopes defined for each **object** in python.
- Built-in namespaces can be called anywhere and hence have the largest scope.
- Global variables can be accessed by all functions.
- Local namespaces can be accessed **only** with the function scope.



Broad classification of namespaces in python

Namespaces




```
def square(x):
    newvar = x**2
    return newvar
square_list = list()
for number in range(10):
    square_list.append(square(number))
square_list
>>>
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Scope of a function variable

- A variable declared within the function has its scope **limited** to the function only.
- The input arguments of a function are also limited to the function only.
- **Return** the value of the variable which you want to preserve after the function execution.

```
x = 20
def my_func():
    x = 10
    print("Value inside function:",x)

print("Value before calling function:",x)
my_func()
print("Value after calling function:",x)
>>>

Value before calling function: 20
Value inside function: 10
Value after calling function: 20
```

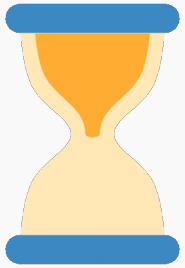
NOTE: *Variables declared in the global scope can be accessed within functions (Check 'enclosing scope')*

Scope of a function variable

```
x = 20
def my_func():
    x = 10
    print("Value inside function:",x)

print("Value before calling function:",x)
my_func()
print("Value after calling function:",x)
>>>

Value before calling function: 20
Value inside function: 10
Value after calling function: 20
```

```
x = 20
def my_f
    x =
    print("Value inside function:",x)
    return x
print("Value before calling function:",x)
x = my_func(x)
print("Value after calling function:",x)

>>>
Value before calling function: 20
Value inside function: 200
Value after calling function: 200
```

Remember to take x as an input, modify it and return it through the function



Digestion Time

Common Pitfalls

- Reserved names will throw an error if you try to use them as variable names.
- However, object names and built-in functions **will not** throw an error.
- Don't use the same variable name as an object in the accessible namespace.
- Take special care in avoiding writing over built-in objects.

```
>>>abs = 5
>>>abs(-10)

TypeError
Traceback (most recent call last)
<ipython-input-1-9ae574a2cf83> in
<module>
      1 abs = 5
----> 2 abs(-10)

TypeError: 'int' object is not
callable
```

101 ways to ruin your built-in functions