

***Rotman***

# FUNCTIONS

February 8, 2022 Prepared by Niti  
TDMDAL & FinHUB




Rotman School of Management  
UNIVERSITY OF TORONTO

# Functions

```
numlist = [4, 8, 10, 15]
```

1. What if we want to add 1 to **each item** of this list **if greater than 5**? → Conditional and iteration!

2. What if we want to be able to do it for **many other lists**? → Copy paste and change the name of list



```
for i in nextlist:  
    if i > 5:  
        print(i+1)  
    else:  
        print(i)
```

→ Actually, it is way more efficient to **create a function** !

# Functions

```
def sixormore(alist):  
    for i in alist:  
        if i > 5:  
            print(i+1)  
        else:  
            print(i)
```

```
numlist = [4,8,10,15]  
sixormore(numlist)
```

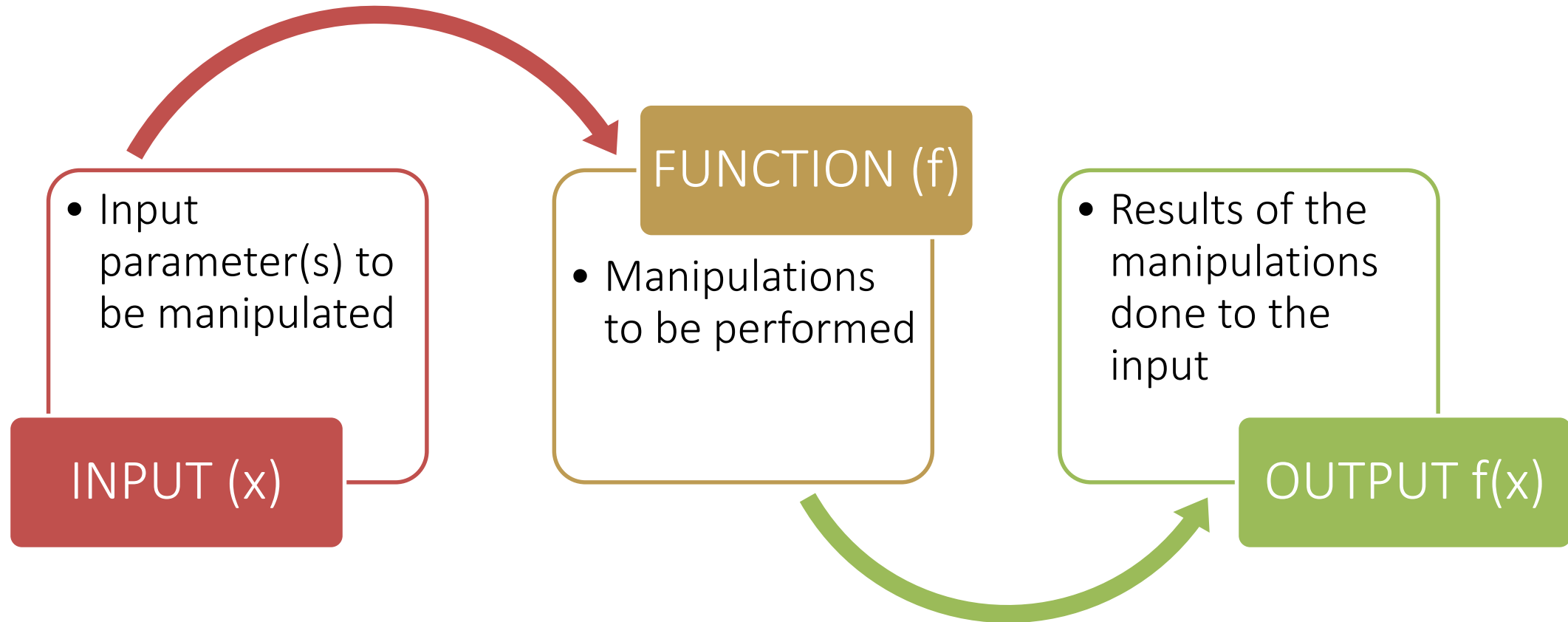
→ 4  
9  
11  
16

```
nextlist = [1,3,4]  
sixormore(nextlist)
```

→ 1  
3  
4

```
alist = [9,11,15]  
sixormore(alist)  
→ 10  
12  
16
```

# Functions



# Functions

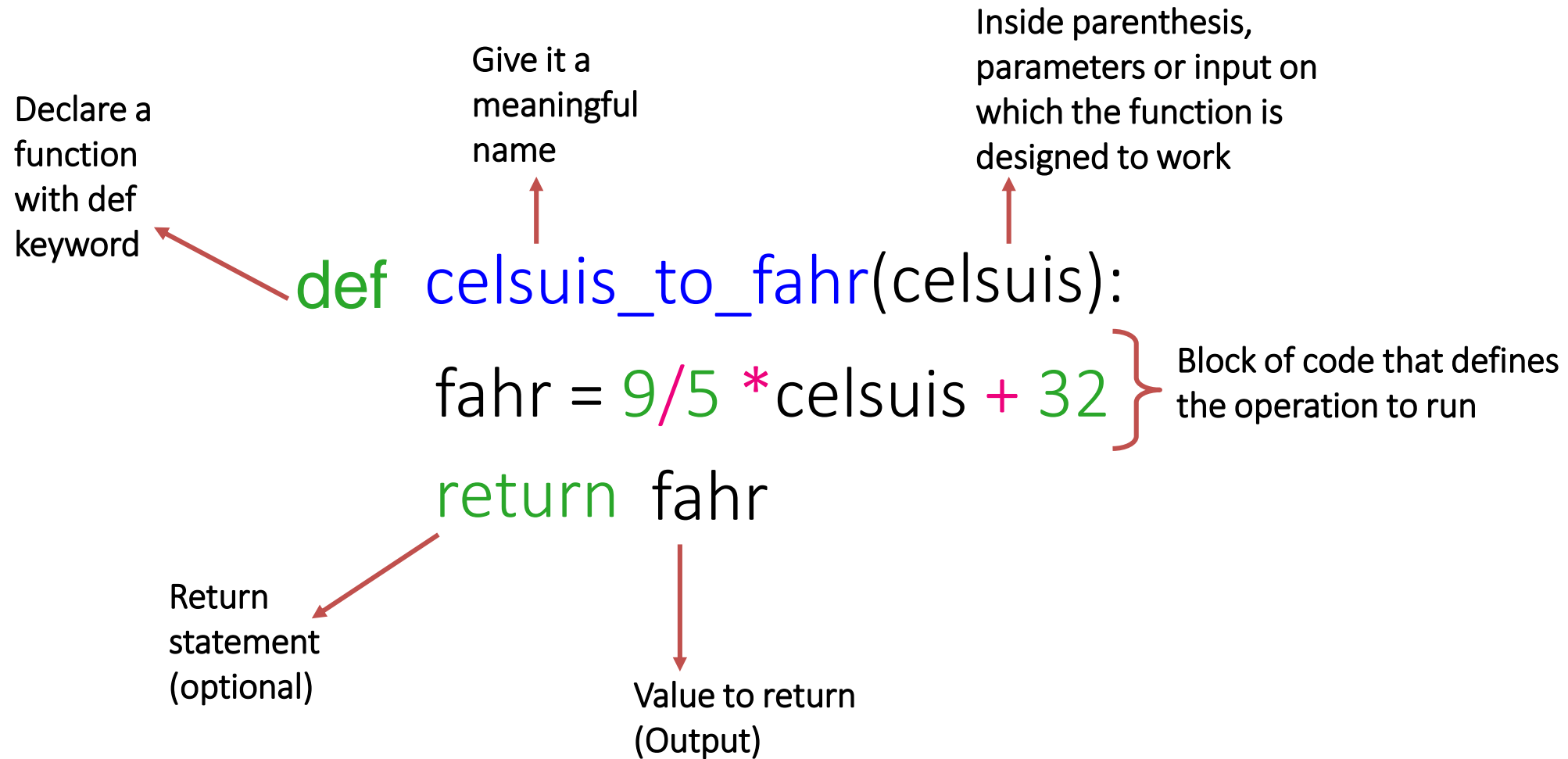
- An efficient way of programming.
- A block of code attached to a name, which runs when that name is called.
- Instead of repeating loops on different lists, functions make a loop generic enough to work on similar lists.
- Helps breakdown a large program into smaller manageable sections.
- All functions, whether created by the user or available in Python follow the same pattern.
- Once a function is defined it must be called, otherwise nothing happens!

# Functions

1. User-defined functions
2. Built-in functions
3. Methods
4. Lambda functions
5. Third-party packages

# User-defined Functions

# Functions: User-defined Functions





# Functions: User-defined Functions

## • Global vs Local Variables

Variables	Global	Local
Created in	jupyter notebook	inside a function
Stored in	notebook's memory	function's memory

- Memory block of function is **not shared** with that of the notebook. Thus, variables of a given name in notebook and variables with the **same names** inside functions are **separate** variables. Variables inside functions are accessed only when the function is in use.
- Separating memory block is efficient for large projects that require numerous variables and functions, and have multiple users, who can all program independently without having to worry about variable names.

# Functions: User-defined Functions

## • Global vs Local Variables

```
In [1]: ▶ def morethan140(alist):  
  
    result = [i for i in alist if i>140]  
  
    if len(result)<1:  
        print(f'There are no values in input list greater than 140')  
    else:  
        print(f'Value of local variable result is: {result}')        return result
```

```
In [2]: ▶ prices = [126, 147, 387]  
    result = 'global variable'  
  
    print(morethan140(alist=prices))  
    print(f'Value of global variable result is: {result}')
```

Value of local variable result is: [147, 387]  
[147, 387]

Value of global variable result is: global variable

Locally i.e. inside the  
function, result is a list

Globally i.e. in  
jupyter notebook,  
result is a string

# Built-in Functions

# Functions: Built-in Functions

- Python interpreter has a number of functions and types built into it that are always available.
- `print()` is an example of built-in function. It prints the given object to the standard output device (screen) or to the text stream file.
- [Here](#) is the list of Python's built-in functions.
- [Here](#) is a list of examples of use cases of Python's built-in functions.

```
numlist = [4, 8, 10, 15]
```

```
type(numlist)  
→ list
```

```
len(numlist)  
→ 4
```

```
sum(numlist)  
→ 37
```

# Functions: Built-in Functions

## • Parameters

- When defining functions, parameters are the names listed within the parenthesis.
- When calling functions, parameters are replaced by the input value, known as argument.
- Parameters make functions generic – functions can be called on different data of the **same data type**.
- Parameters are temporary variables (like the looping variable).
- A functions can have one or more parameters.
- Not all functions have parameters.

```
numlist = [4, 8, 10, 15]
```

```
len(numlist)  
→ 4
```

numlist is the argument.  
What is the parameter?

len(**s**)

Return the length (the number of items) of an object. The argument may be a sequence (such as a string, bytes, tuple, list, or range) or a collection (such as a dictionary, set, or frozen set).

s is the parameter

# Functions: Built-in Functions

- **Parameters' order**

- The order in which arguments are passed must line up with the order of parameters listed in function definition.
- You can also explicitly assign values for every parameter but it is not necessary.

- **Parameters with default value**

- Parameters can be associated with a default value when values are likely to stay the same.
- Parameters with default value always go **after** parameters with non-default values.

- **Parameters with optional value**

- Parameters can take optional arguments, which are used in operation inside a function only when they are defined.

# Methods

# Functions: Methods

- Functions that are attached to specific class of objects.
- Methods are accessed using the dot expression.
- Methods available to an object can be viewed using "dir" function.

```
b = 'Hello World!'
```

```
b.upper()  
→ 'HELLO WORLD!'
```

methods available  
to string objects  
only

```
b.isnumeric()  
→ False
```


```
b.count('l')  
→ 3
```



# Functions: Methods

- How are methods supposed to work?
- There are documentations available with information on how a given method is intended to work.
- [Python's official documentation for methods of list object](#)
- [Easy-to-read documentation provided by w3schools.](#)

```
numlist=[4,8,10,15]  
numlist.append(16)  
numlist  
→ [4,8,10,15,16]
```



.append is a method  
available to objects of  
class list only

# Lambda Functions

# Functions: Lambda Functions

- Nameless one line functions
- Begin with the **lambda** keyword
- Have arguments and a expression separated by a colon
- Any number of arguments are allowed but only one expression!
- Best used with another function.

```
lambda arguments : expression
```

```
lambda x : x**2
```

```
lambda x, y : x*y
```

# Third Party Packages

# Functions: Third Party Packages

- **Modules**

- Simply put, modules are files with Python scripts.
- Modules can include functions, variables and classes.
- As the length of our Python code grows, it becomes lengthy and difficult to manage.
- Modules are basically large pieces of code saved into smaller more manageable forms.
- Items inside a module can be accessed using import statement.
- You can also create your own modules.

# Functions: Third Party Packages

## • Python Modules

- Python provides in-house support for some packages/libraries, called [Python Modules](#).
- There is **no** need to install them individually.
- Simply import them and start using the functions inside these modules.

```
In [1]: ► import datetime as dt
```

```
In [2]: ► date_str = '2021-01-06'  
date_datetime = dt.datetime.strptime(date_str, '%Y-%m-%d')  
print(date_datetime)  
  
2021-01-06 00:00:00
```

```
In [3]: ► date_datetime.year
```

```
Out[3]: 2021
```

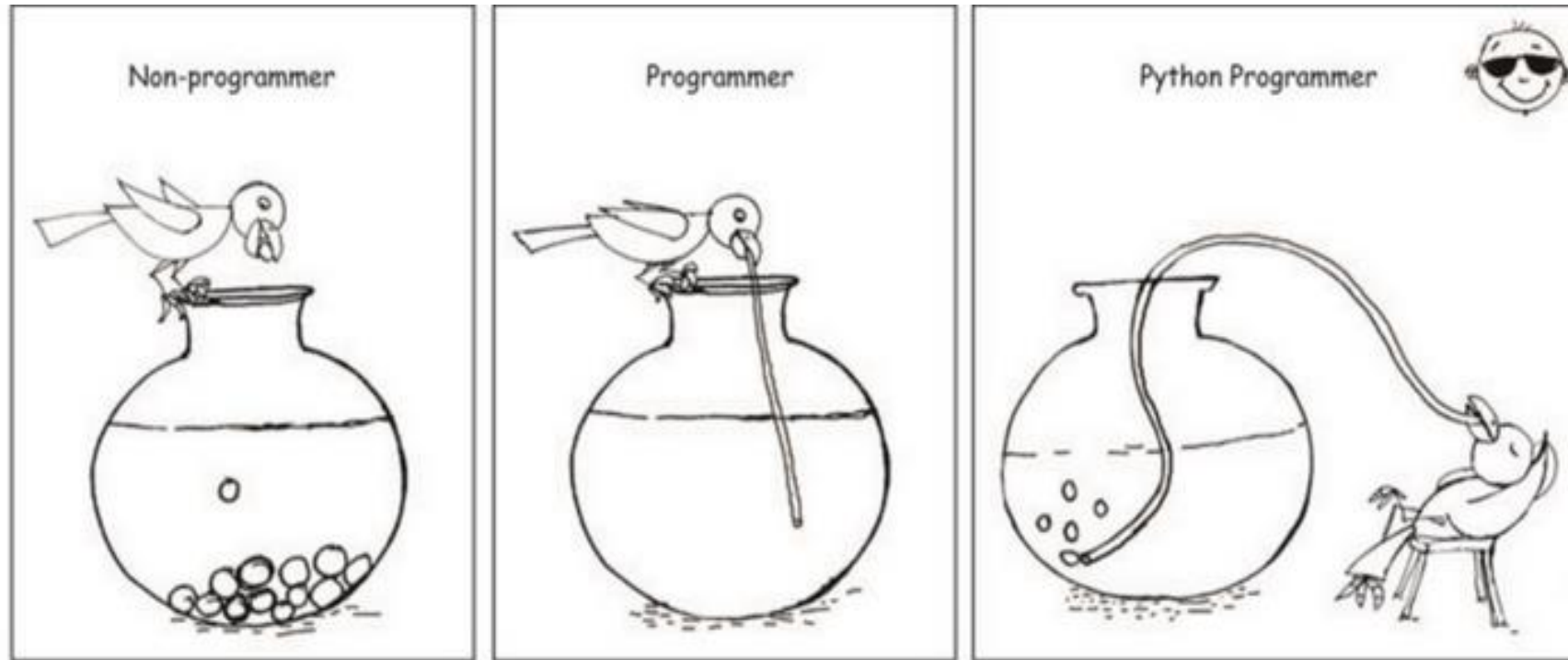
# Functions: Third Party Packages

- Python has an active supporting community of contributors and users who also make their software available for other Python developers to use under its open source license terms. These are called Third Party packages.
- The [SciPy](#) ecosystem is a collection of python packages for scientific computing in Python.
- Third-party packages **must** be installed before they can be used in Python environments.
- Once installed, the content of the package can be accessed using the import statement
- Numpy, pandas and matplotlib libraries come pre-installed via Anaconda package manager.



**Always make sure it  
is safe to install a  
third party package!**

# Questions?



Who wants to become a Python Programmer?

# Thank you