Python –Data Science

#DATASCIENCEJOURNEYWITHANR#

Python Intro

What can Python do?

Welcome to the world of Python :)

Python is really simple to learn yet powerful at the same time. Its syntax is similar to the English language.
Python was created by Guido Van Rossum in 1991 and it soon became very popular.
Today you can do almost everything in Python including

Web and software development

System Automation

Building bots

Interacting with Raspberry Pi

Why Python

Why Python?

It has a simple syntax similar to the English language.

It works on all the platforms like Windows, Unix, Mac, Raspberry pi etc

It is free and open source. You do not have to worry about buying costly licenses

Python as a Calculator

We can use Python as a quick and handy calculator. We can perform all the basic four operations - addition, subtraction, multiplication, and division in Python.

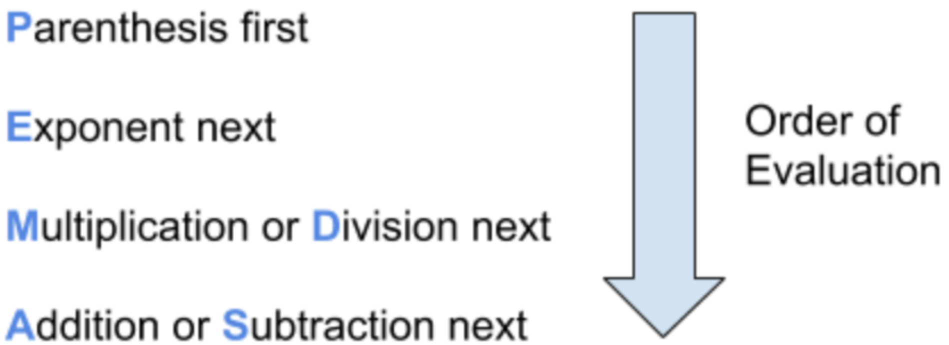
Order of Operations in Python

Let's understand a very important concept - order of operations. Let's calculate the average of five numbers 7, 6, 0, 4, and 3

Apply the above formula to calculate the average. Divide sum of numbers 7 + 6 + 0 + 4 + 3 by count of numbers 5

PEMDA

- Execute the above expression in the Jupyter notebook and check the average. It prints the average as 17. Is average correct?
- No, it is wrong. 7 + 6 + 0 + 4 + 3 is 20 and the count of numbers is 5. So average should be 4
- So what is wrong and how do find the correct average?
- Hope you remember PEMDAS from your High School math classes. **PEMDAS defines the order of operations.**
- Below is the order of operations
- Notice Multiplication or Division and Addition or Subtraction are on the same level. We can either evaluate multiplication or division whichever comes first.
- Similarity we can either evaluate addition or subtraction whichever comes first.
- Now since we know the order of operations, we can figure out why the above average calculation was wrong.
- Our code first evaluated 3 // 5 and got the result as 0 and then it added 7, 6, 0, 4 and 0 which resulted in 17.
- Below is the explanation of why we got the incorrect average



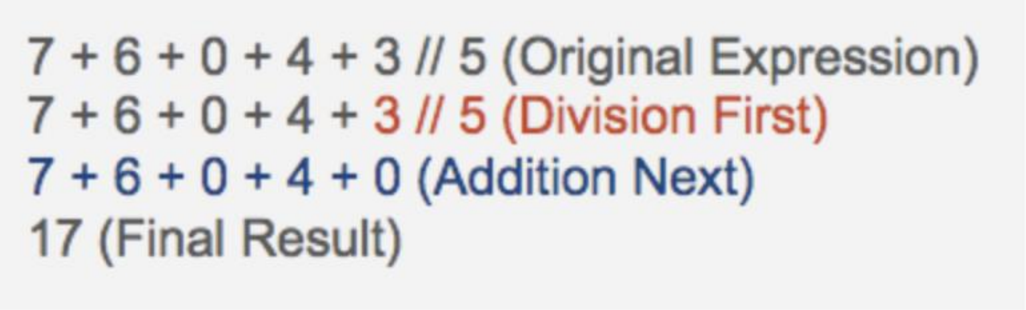
Parenthesis first

Exponent next

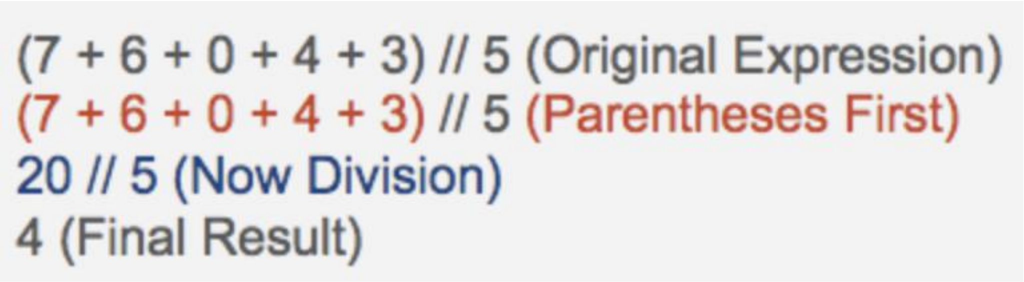
Multiplication or Division next

Addition or Subtraction next

Order of Evaluation



7 + 6 + 0 + 4 + 3 // 5 (Original Expression)
7 + 6 + 0 + 4 + 3 // 5 (Division First)
7 + 6 + 0 + 4 + 0 (Addition Next)
17 (Final Result)



(7 + 6 + 0 + 4 + 3) // 5 (Original Expression)
(7 + 6 + 0 + 4 + 3) // 5 (Parentheses First)
20 // 5 (Now Division)
4 (Final Result)

Advanced Operations in Python

So far we have seen basic operations in Python such as addition, subtraction, multiplication, and division.

Let's see some of the advanced operations in Python

Exponentiation: **. This operator raises the number to its left to the power of the number to its right. For example, 8 ** 2 gives 64.

Modulo: %. This operator returns the remainder of the division of the number to the left by the number on its right. For example, 16 % 7 equals 2 and 5 % 2 equals 1

Variables in Python

In the previous exercise, we have seen Python is a great calculator. In real-life when we write code, we use variables to define values so that we can reuse them again and again in our code.

A variable is a name that refers to a value. Let's see some examples.

```
my_name = "John"
age = 28
pi = 3.14
```

Here John is a string which is assigned to my_name variable. Similarly 28 is a number which is assigned to variable age. Variable pi is assigned the value of 3.14

Print & Get Values

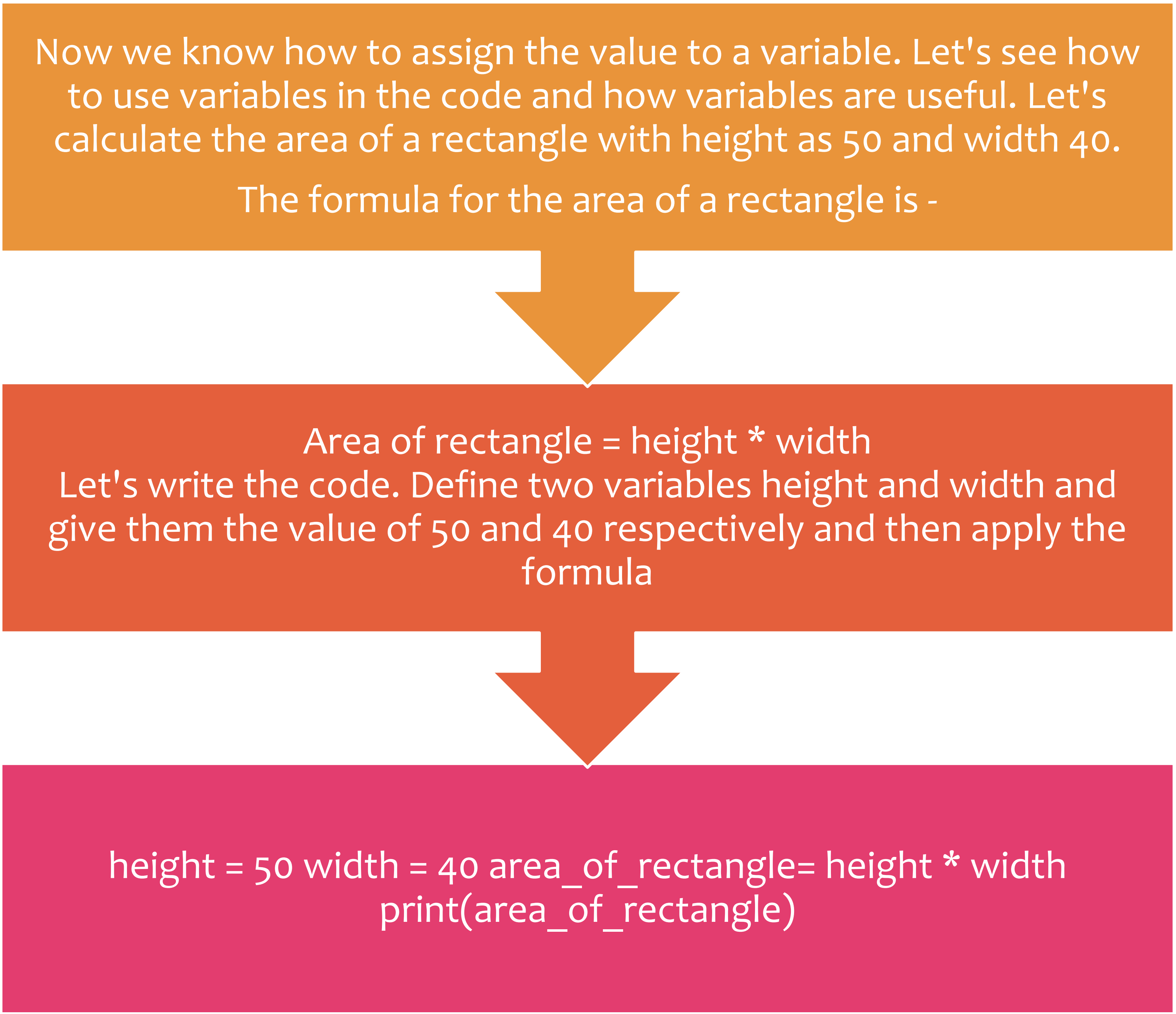
Pro Tip - = is an assignment operator in Python. You can assign a value to a variable using = operator

Display value of a variable

To print the value of a variable, we use print statement.
To print the value of my_name and pi, simply type below statement into a Jupyter notebook on the right-hand side and execute the cells.

```
print(my_name)
print(pi)
```

Calculations With Variables



Now we know how to assign the value to a variable. Let's see how to use variables in the code and how variables are useful. Let's calculate the area of a rectangle with height as 50 and width 40.

The formula for the area of a rectangle is -

Area of rectangle = height * width
Let's write the code. Define two variables height and width and give them the value of 50 and 40 respectively and then apply the formula

```
height = 50 width = 40 area_of_rectangle= height * width
print(area_of_rectangle)
```

Types of Variables in Python

Hands-on on Python Variables

Now we know the operations and variables in Python. Let's write a program to calculate the BMI.

BMI = weight / (height * height) Here weight is in kg (kilogram) and height is in mt (meters)

Types of Variables in Python

In the previous exercise, we worked with two types of variables

integer, or int: a number without decimal part. bmi with the value 25 is an example of an integer. height with the value of 40 (in rectangle example) is also an integer.

float, or floating point: a number that has both integer and decimal parts. weight with the value of 72.25 is an example of a float.

Other common data types in Python are

str, or string: Represents sequences of characters such as a message or text.
Variable my_name = "John" is an example of a string

Pro Tip - We can use either single or double quotes to represent strings.

Example - "John" (notice double quotes) and 'John' (notice single quotes) both are same strings

bool, or boolean: Represents logical values such as True and False.

Pro Tip - Notice capitalization in True and False. Capitalization is important for Boolean variables in Python

Operations with Other Types

Different types behave differently in Python. For example, + operator

On integers it just sums up the integers -> print(5 + 8) -> It will print 13

While on strings it paste together(joins) the strings. To print the full name of person having first name as John and last name as Barley -> print("John" + " " + "Barley") -> Notice " " here, it adds space while joining the strings. It will print John Barley

Now since we know the + operator we can print a message like this:

print("BMI is " + 25)

Run above code in the notebook. Did it run?

This will not work as you cannot simply sum strings and integers.

How do we fix it then?

To fix this error, you'll need to explicitly convert the bmi, which is an integer, to a string. You'll need str() to convert a value into a string. str(bmi) will convert the integer bmi to a string.

Pro Tip - Remember, we can only sum like types

Functions

A function is basically a sequence of statements that collectively perform some task or a group of tasks.

We call a function by its name.

We have already encountered certain functions like,type(23) The name of the function is type. It returns int.

The expression in parentheses is called the argument of the function.

The argument is a value or variable that we are passing into the function as input to the function.

Therefore, a function takes an argument and returns a result.

Built-in Functions

Python provides you with certain built-in functions. For eg,

len - It returns how many items are in its argument. If the argument to len is a string, it returns the number of characters in the string.

len("myname") It returns the length as 6.

min - It returns the smallest value in the list or a string

min("python") It returns the character 'h'.

max - It returns the largest value in the list or a string. (We will explain the list data type later in the slides)

max(['p','y','t','h','o','n']) It returns the element 'y'.

These functions are not only limited to strings. They can operate on any set of values, as we will see in later topics.

You should treat the names of built-in functions as reserved words (i.e., avoid using "max" as a variable name).

Type Conversion Functions

We can also use certain built-in functions to convert one data type to another.

int - It takes any value and converts it to an integer, if it can, or returns the error.

int(23.45) It returns 23. Try converting the string ”datasciencejourneywithanr" to int and see the result.

float - It converts integers and strings to floating-point numbers.

float(2) It returns 2.0

str - It converts its argument to a string

str(123) It returns '123'

Lets Get Hands-on On The Go

Calculate the area of a square whose length of one edge is 8.9 and store it in the variable square_area

Store the type of square_area in square_area_type

Convert the area into int and store it in square_area_int

Again, convert the int into str and store it in square_area_str

Math Functions

Python provides us with a module called math which has certain functions and variables that help us in doing mathematical calculations.

Functions in math module return float values. Before we can use the module, we have to import it:

import math
Try print(math) and see the result.
(You need to import math before printing)

To access one of the functions, we need to specify the name of the module and the name of the function, separated by a dot (also known as a period).

This format is called dot notation.

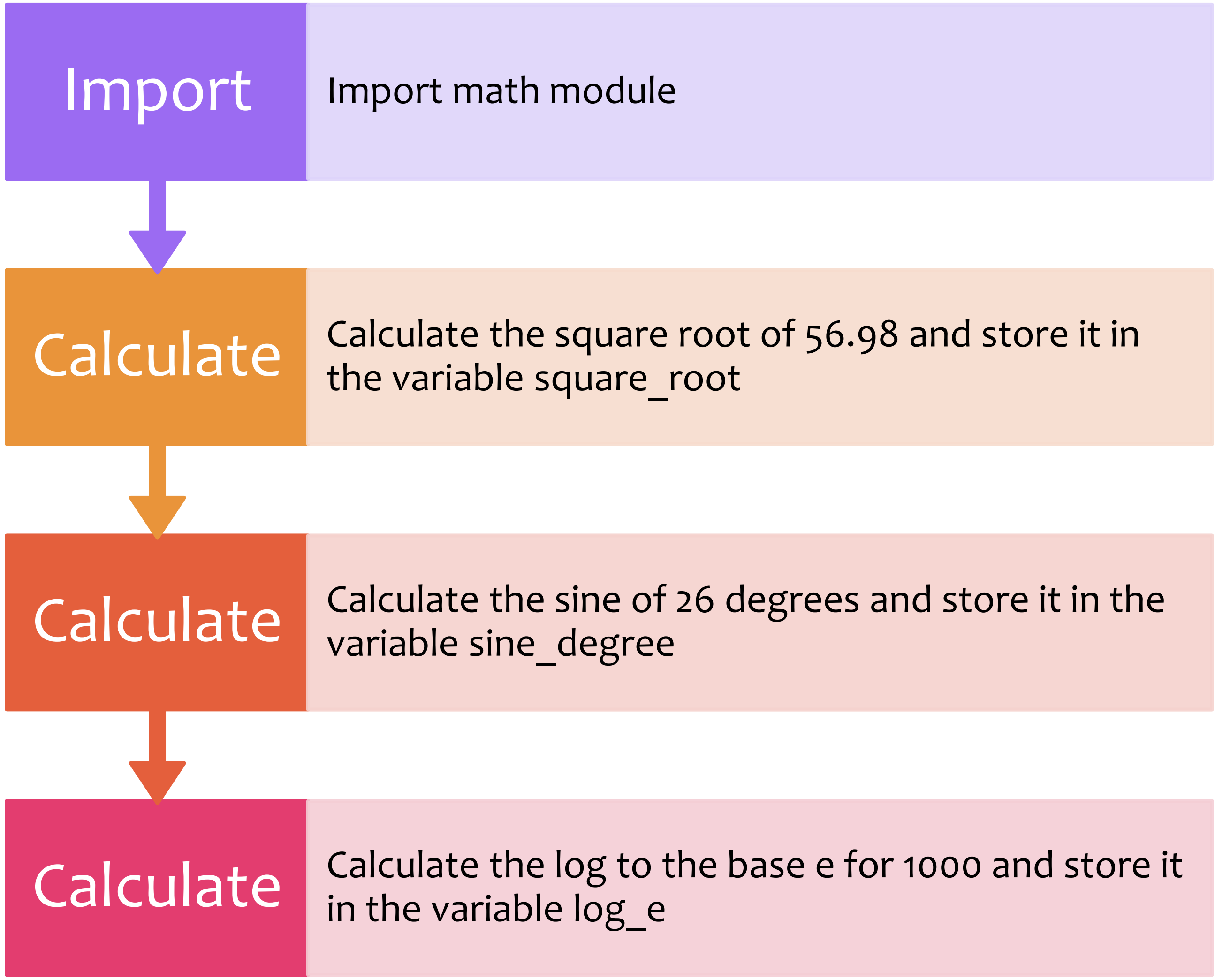
math.log10(1000) -> It will calculate log to the base 10 and return 3.0 in this case. The math module also provides a function called log that computes logarithms base e.

math.cos(0) It will find the cosine of radians and return 1.0 in this case. The name of the variable is a hint that cos and the other trigonometric functions (sin, tan, etc.) take arguments in radians.

The expression math.pi gets the variable pi from the math module. The value of this variable is an approximation of π, accurate to about 15 digits. To convert from degrees to radians, divide by 360 and multiply by 2π.math.pi gets

math.sqrt(4) It will calculate the squre root of 4 and return 2.0 here.

Lets Gets Hands-on the Go



Import — Import math module

Calculate — Calculate the square root of 56.98 and store it in the variable square_root

Calculate — Calculate the sine of 26 degrees and store it in the variable sine_degree

Calculate — Calculate the log to the base e for 1000 and store it in the variable log_e

Random Number Generation

The random module provides functions that generate pseudorandom numbers.

import random
random.random()
It returns a random float value between 0.0 and 1.0 (including 0.0 but not 1.0).

Each time you call random, you get the next number in a long series.
To see a sample, run this loop and observe the results:

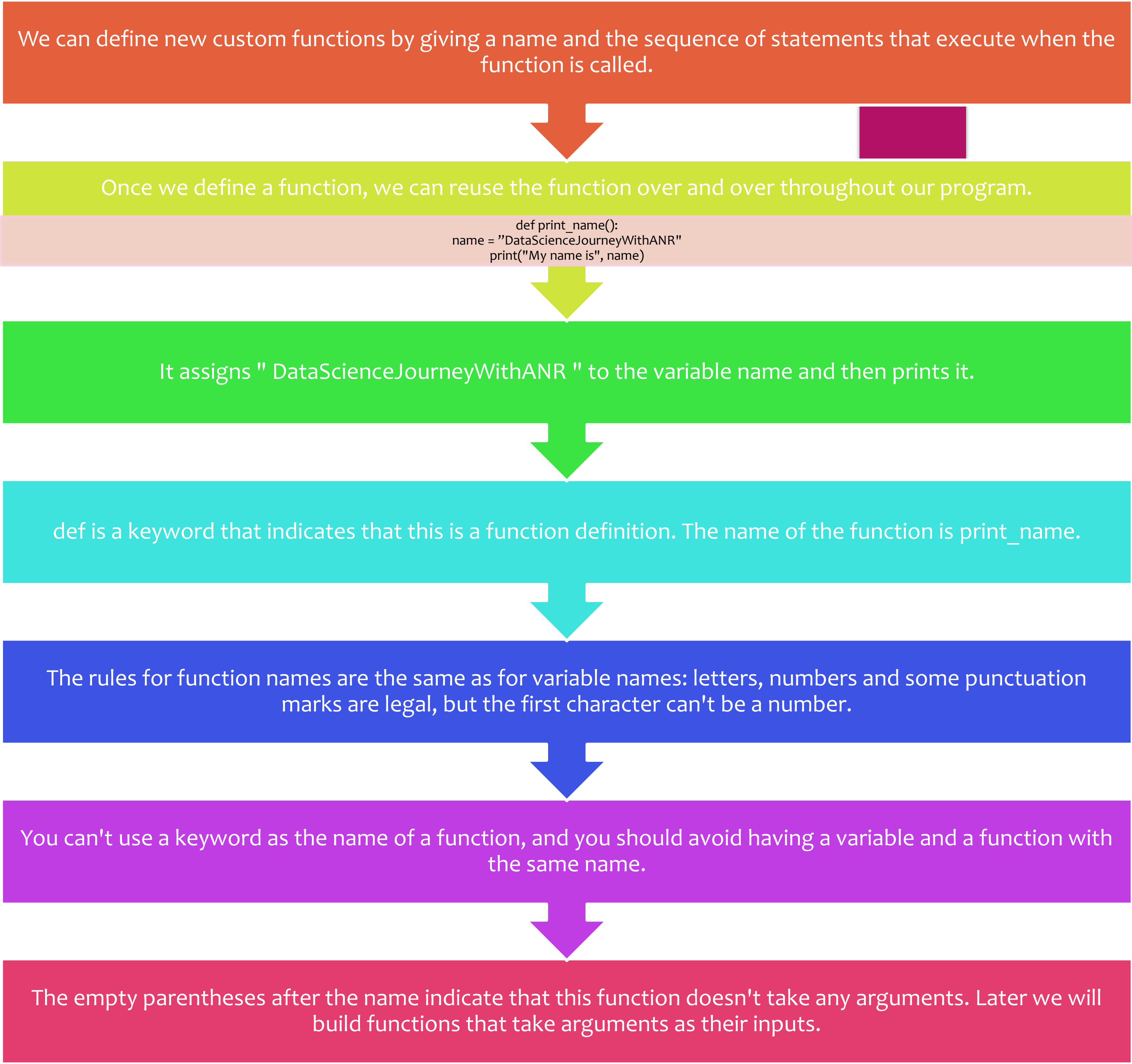
```
for a in range(10):
number = random.random()
print(number)
```

This loop runs from 10 times and prints a random in every iteration.

Lets Gets Hands-on the Go

- Import random module
- Generate a random number and store it in the variable random_number
- Convert the random_number into int using the type conversion and store it in random_int

Making your own functions



We can define new custom functions by giving a name and the sequence of statements that execute when the function is called.

Once we define a function, we can reuse the function over and over throughout our program.

```
def print_name():
name = ”DataScienceJourneyWithANR"
print("My name is", name)
```

It assigns " DataScienceJourneyWithANR " to the variable name and then prints it.

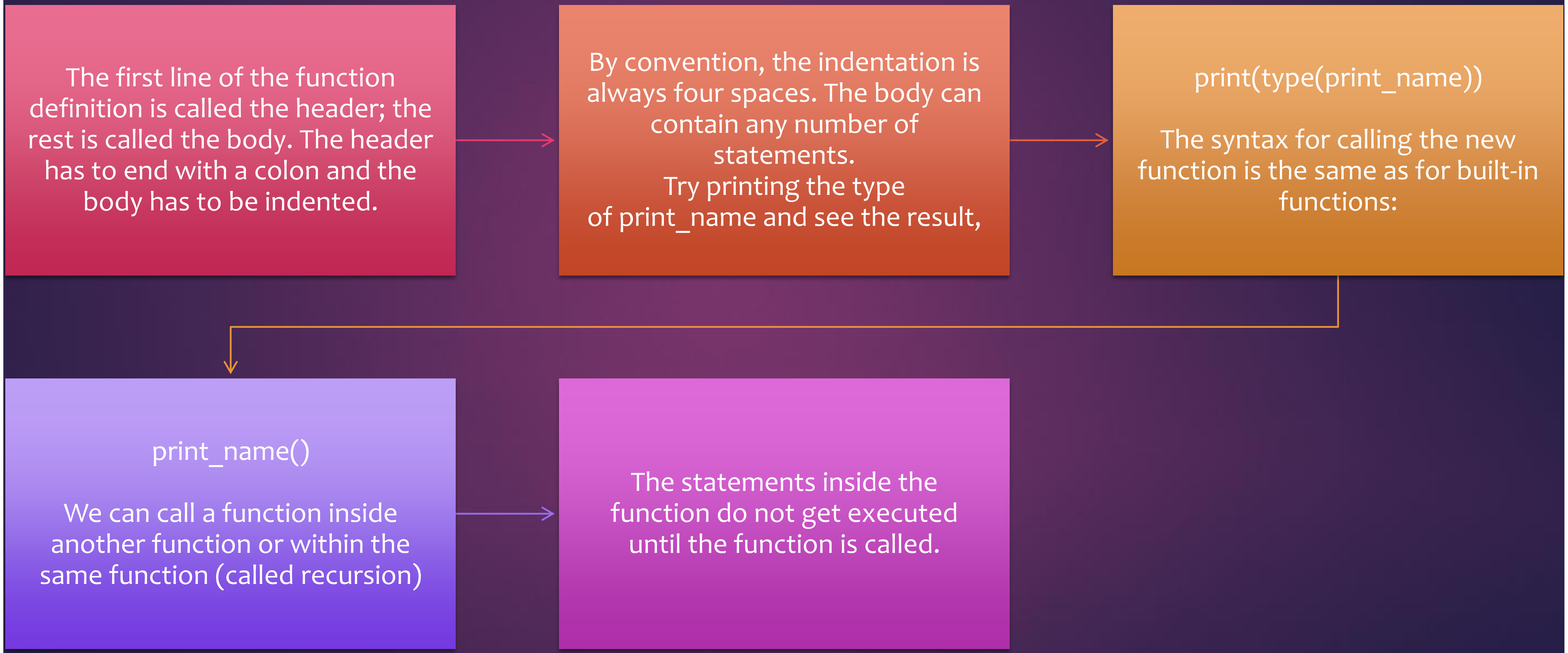
def is a keyword that indicates that this is a function definition. The name of the function is print_name.

The rules for function names are the same as for variable names: letters, numbers and some punctuation marks are legal, but the first character can't be a number.

You can't use a keyword as the name of a function, and you should avoid having a variable and a function with the same name.

The empty parentheses after the name indicate that this function doesn't take any arguments. Later we will build functions that take arguments as their inputs.

Making Your Own Functions Cont



The first line of the function definition is called the header; the rest is called the body. The header has to end with a colon and the body has to be indented.

By convention, the indentation is always four spaces. The body can contain any number of statements.
Try printing the type of print_name and see the result,

print(type(print_name))

The syntax for calling the new function is the same as for built-in functions:

print_name()

We can call a function inside another function or within the same function (called recursion)

The statements inside the function do not get executed until the function is called.

Flow of Execution

Execution always begins at the first statement of the program. Statements are executed one at a time, in order from top to bottom.

Function definitions do not alter the flow of execution of the program but remember that statements inside the function are not executed until the function is called.

A function call is like a detour in the flow of execution. Instead of going to the next statement, the flow jumps to the body of the function, executes all the statements there, and then comes back to pick up where it left off.

You need to create a function before you can execute it. In other words, the function definition has to be executed before the first time it is called.

Try calling the function before executing it and observe the result,

```
get_name()
def get_name():
print(”DataScienceJourneyWithANR")
```

Lets Gets Hands-on the Go

Correct the function call in the example above and execute it correctly.

```
get_name()
def get_name():
print(”DataScienceJourneyWithANR")
```

```
def get_name():
print("DataScienceJourneyWithANR ")
get_name()
```

Parameters and Arguments

Some of the built-in functions we have seen require arguments. For example,

When you call math.cos you pass a number as an argument.

math.pow takes two arguments, the base and the exponent to calculate the exponential of some number.

Inside the function, the arguments are assigned to variables called parameters. Here is an example of a user-defined function that takes an argument:

```
def print_name(value):
x = value
print(value)
```

This function assigns the argument to a parameter named value. When the function is called, it assigns the value to x and then prints the value of the parameter (whatever it is).

The same rules of composition that apply to built-in functions also apply to user-defined functions, so we can use any kind of expression as an argument for print_name:

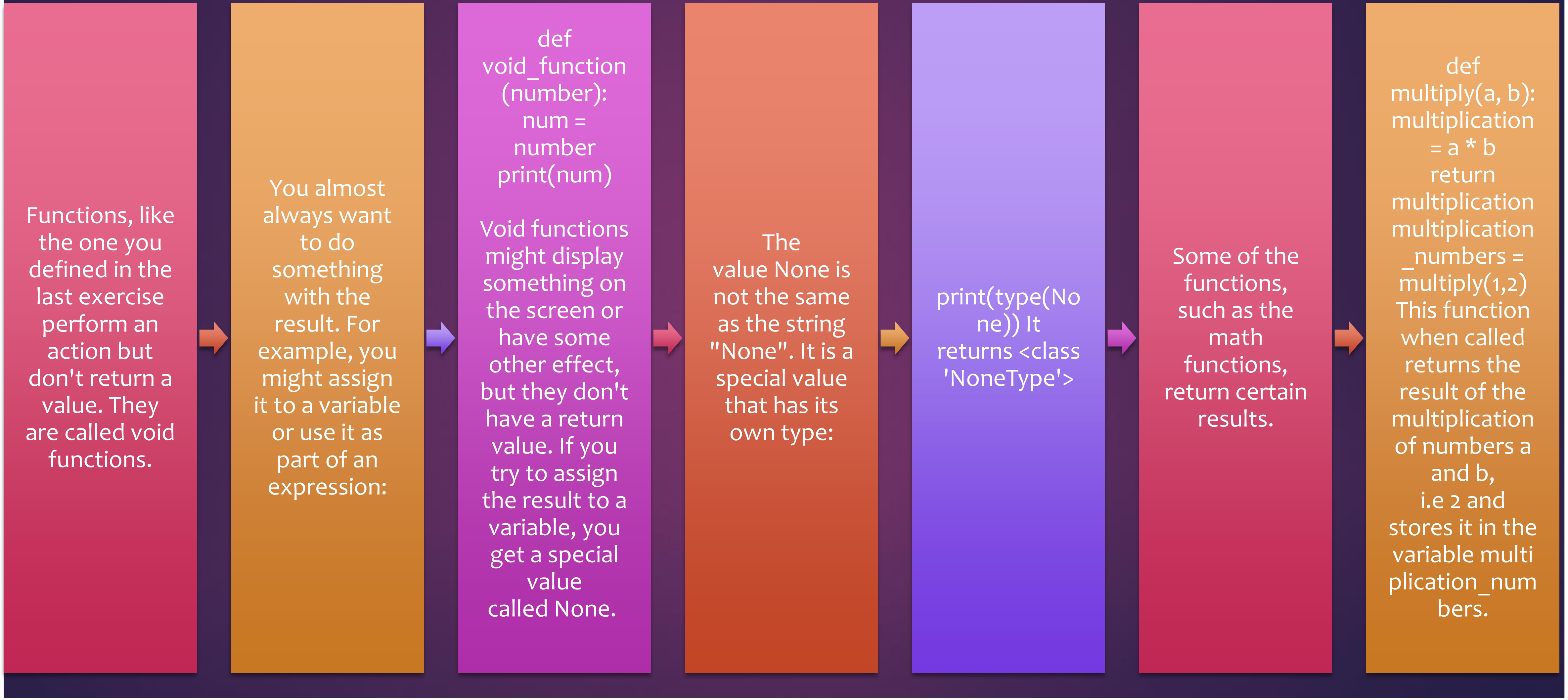
```
print_name(str(math.cos(0)))
It prints 1.0
```

The argument is evaluated before the function is called, so in the above example, str(math.cos(0)) is evaluated only once.

Lets Gets Hands-on the Go

Write	Write a function with name my_function which takes one argument with name number. Then, inside the function, write these statements,
Store	1.) Store the number in a variable xyz 2.) Convert it into str and store it in xyz_str 3.) Concatenate ”PythonWorkShop" to xyz_str 4.) print xyz_str
Call	Call your function by passing an argument 10

Types of Functions



Functions, like the one you defined in the last exercise perform an action but don't return a value. They are called void functions.

You almost always want to do something with the result. For example, you might assign it to a variable or use it as part of an expression:

```
def void_function(number):
num = number
print(num)
```

Void functions might display something on the screen or have some other effect, but they don't have a return value. If you try to assign the result to a variable, you get a special value called None.

The value None is not the same as the string "None". It is a special value that has its own type:

print(type(None)) It returns <class 'NoneType'>

Some of the functions, such as the math functions, return certain results.

```
def multiply(a, b):
multiplication = a * b
return multiplication
multiplication_numbers = multiply(1,2)
```

This function when called returns the result of the multiplication of numbers a and b, i.e 2 and stores it in the variable multiplication_numbers.

Lets Gets Hands-on the Go

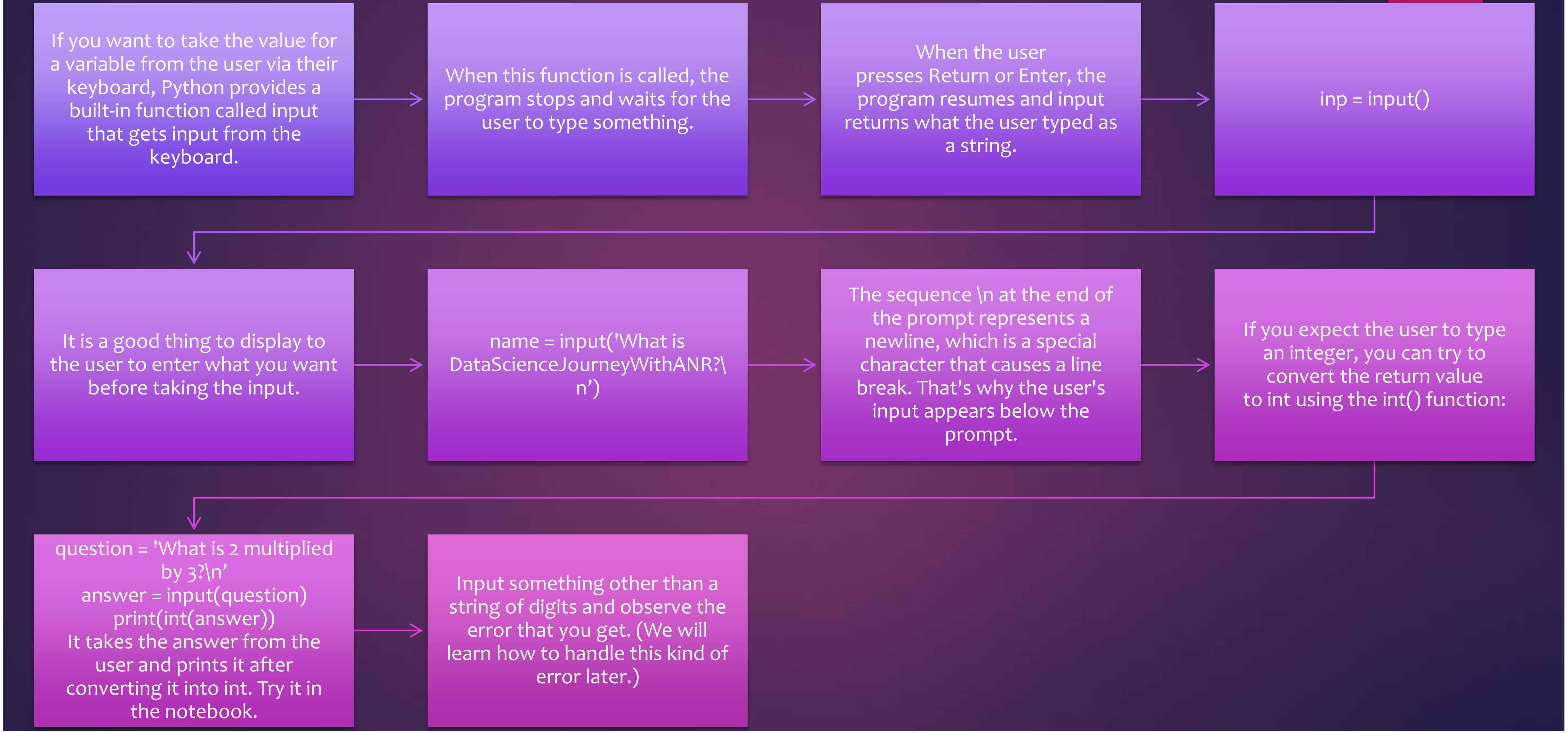
Define a function with name new_function that takes an argument num and returns its multiplication with π.

Define a void function with the name void_function which takes two arguments num1 and num2 which makes the call to new_function with an argument as num1 raised to the power of num2.

Withing the void_function print the value returned by the call.

In a new cell, call void_function with arguments as 4.6 and 7.3.

Input from User



If you want to take the value for a variable from the user via their keyboard, Python provides a built-in function called input that gets input from the keyboard.

When this function is called, the program stops and waits for the user to type something.

When the user presses Return or Enter, the program resumes and input returns what the user typed as a string.

inp = input()

It is a good thing to display to the user to enter what you want before taking the input.

name = input('What is DataScienceJourneyWithANR?\n’)

The sequence \n at the end of the prompt represents a newline, which is a special character that causes a line break. That's why the user's input appears below the prompt.

If you expect the user to type an integer, you can try to convert the return value to int using the int() function:

question = 'What is 2 multiplied by 3?\n’
answer = input(question)
print(int(answer))
It takes the answer from the user and prints it after converting it into int. Try it in the notebook.

Input something other than a string of digits and observe the error that you get. (We will learn how to handle this kind of error later.)

Lets Get Hands-On The Go

Define a function with name input_name that displays a prompt, "What am I studying?"

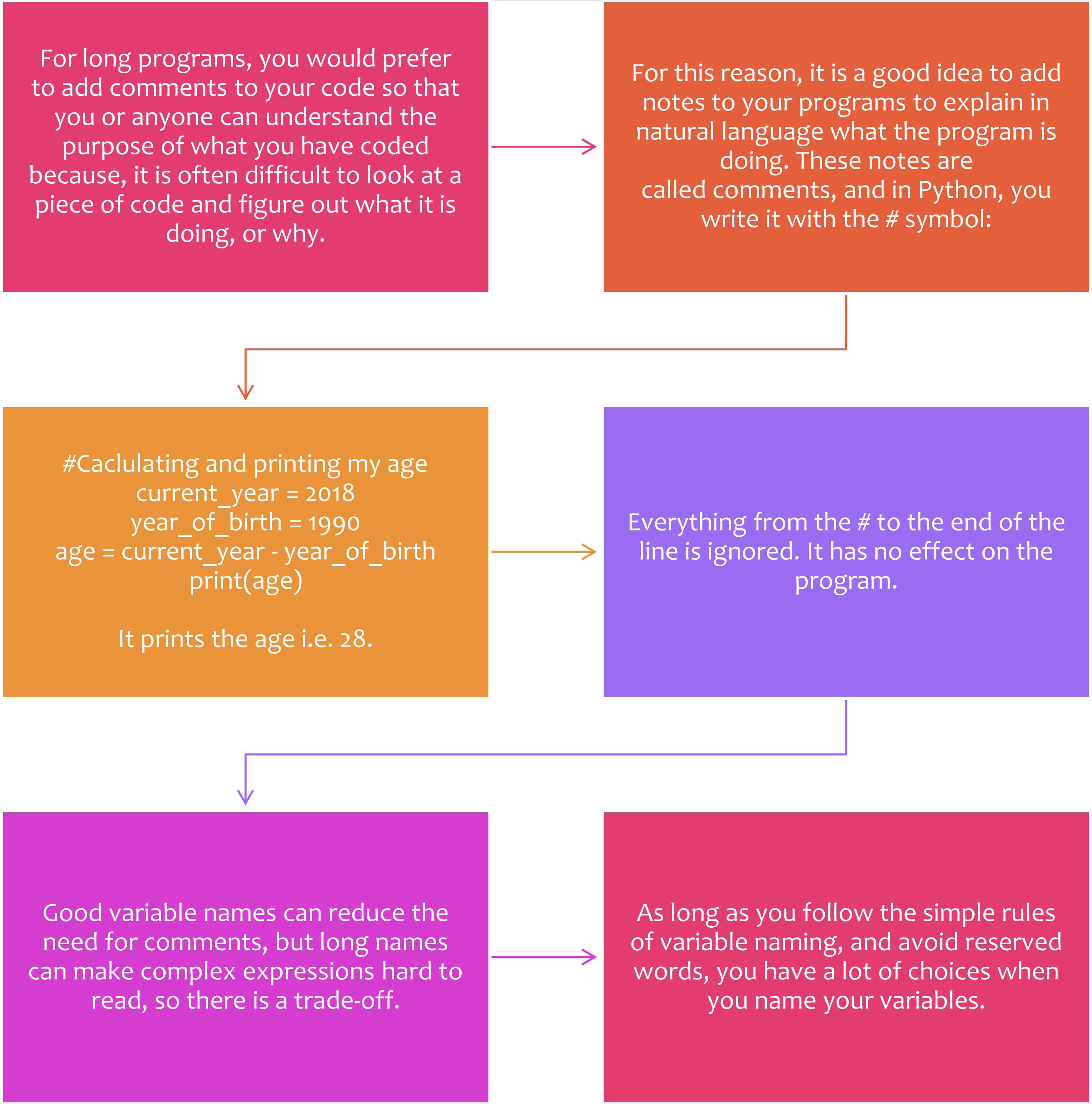
Then, within the function take an input from the user and return it.

In a new cell, call the function, input Python after the prompt.

Store the returned value in the variable subject.

Print subject.

Comments



For long programs, you would prefer to add comments to your code so that you or anyone can understand the purpose of what you have coded because, it is often difficult to look at a piece of code and figure out what it is doing, or why.

For this reason, it is a good idea to add notes to your programs to explain in natural language what the program is doing. These notes are called comments, and in Python, you write it with the # symbol:

```
#Caclulating and printing my age
current_year = 2018
year_of_birth = 1990
age = current_year - year_of_birth
print(age)
```

It prints the age i.e. 28.

Everything from the # to the end of the line is ignored. It has no effect on the program.

Good variable names can reduce the need for comments, but long names can make complex expressions hard to read, so there is a trade-off.

As long as you follow the simple rules of variable naming, and avoid reserved words, you have a lot of choices when you name your variables.

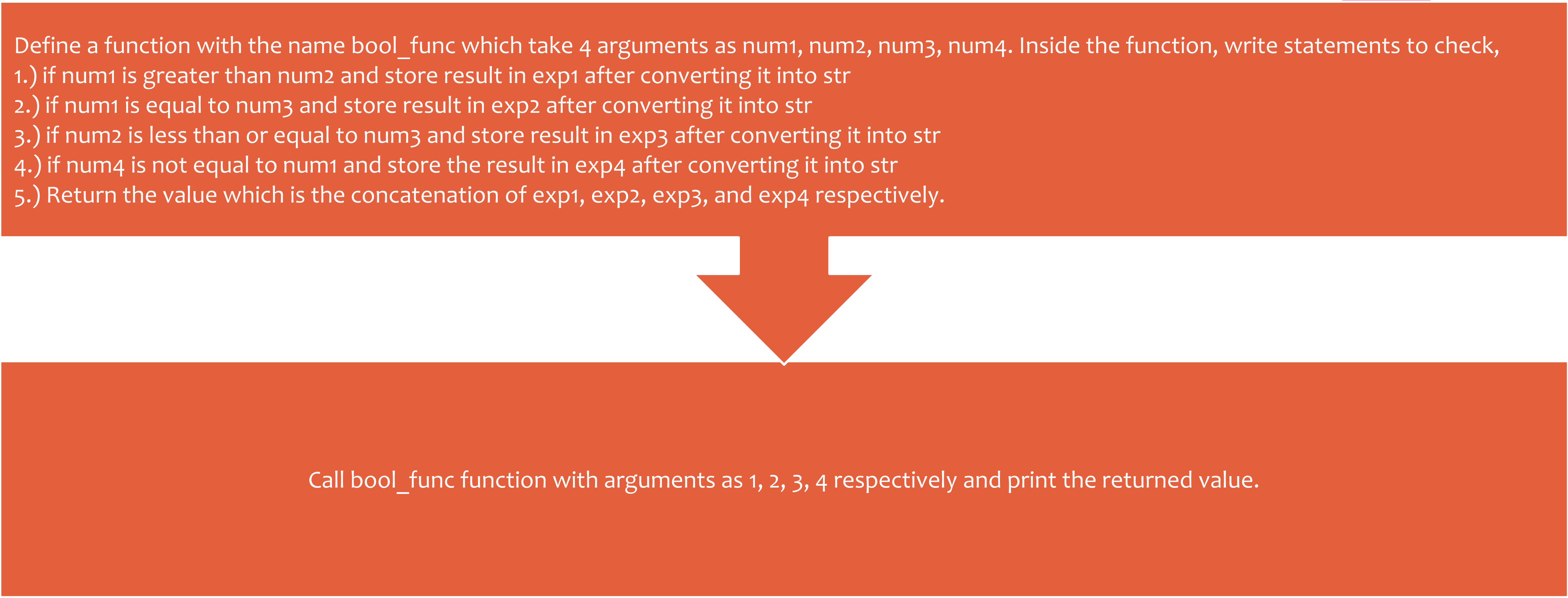
Boolean Expressions

A Boolean expression is either true or false. The following examples use the operator ==, which compares two operands and produces True if they are equal and False otherwise:
1 == 1 It returns True.
2 == 1 It returns False.
True and False are special values that belong to the class bool,
print(type(True)) -> It returns <class 'bool'>

The == operator is one of the comparison operators; the others are:

```
x != y          # x is not equal to y
x > y           # x is greater than y
x < y           # x is less than y
x >= y          # x is greater than or equal to y
x <= y          # x is less than or equal to y
x is y          # x is the same as y
x is not y      # x is not the same as y
```

Avoid the mistake of using a single equal sign (=) instead of a double equal sign (==).
= is an assignment operator and == is a comparison operator.
There is no such thing as =< or =>



Define a function with the name bool_func which take 4 arguments as num1, num2, num3, num4. Inside the function, write statements to check,

1.) if num1 is greater than num2 and store result in exp1 after converting it into str
2.) if num1 is equal to num3 and store result in exp2 after converting it into str
3.) if num2 is less than or equal to num3 and store result in exp3 after converting it into str
4.) if num4 is not equal to num1 and store the result in exp4 after converting it into str
5.) Return the value which is the concatenation of exp1, exp2, exp3, and exp4 respectively.

Call bool_func function with arguments as 1, 2, 3, 4 respectively and print the returned value.

Lets Gets Hands-on the Go

Logical Operators

There are 3 logical operators in Python: and, or, and not.

The meaning of these operators is similar to their meaning in English.
3 < 4 and 4 > 1- It returns True

x % 2 == 0 or x % 4 == 0 - It returns True if either of the conditions is true, i.e. if the number x is divisible by 2 or 4.

The not operator negates a boolean expression, so, - not (a > b)
It returns True if a > b is false i.e. if a is less than or equal to b.
Generally, **the operands of the logical operators should be boolean expressions,** but Python is not very strict. Any nonzero number is interpreted as True.
23 and True

Logical Operators Cont

While processing a logical expression such as x >= 2 and (x/y) > 2, Python evaluates the expression from left to right.

Because of the definition of and, if x is less than 2, the expression x >= 2 is False and so the whole expression is False regardless of whether (x/y) > 2 evaluates to True or False.

So, when there is nothing to be profitable by evaluating the rest of a logical expression, it stops its evaluation and does not do the computations in the rest of the logical expression.

When the evaluation of a logical expression stops because the overall value is already known, it is called **short-circuiting** the evaluation.

Short-circuiting helps in creating a guard. For eg

```
if y != 0 and x/y:
print("Correct")
```

Here, y != 0 acts as a guard to ensure that we only execute (x/y) if y is non-zero because division by 0 would give an error.

Conditional Execution

Conditional statements give us the ability to check conditions and change the behaviour of the program accordingly. Most basic is if statement,

```
if x > y:
print("Yes x is greater than y")
elif x == y:
print("Oops! x is equal to y")
else:
print("No x is not greater than y")
```

The boolean expression after the if statement is called the condition.

We end the if statement with a colon character (:) and the line(s) after the if statement are indented. Same is valid for elif and else statements.

Conditional Execution

If the logical condition is true, then the indented statement gets executed. If the logical condition is false, the indented statement is skipped and the flow goes forward to elif or else statement whatever is present.

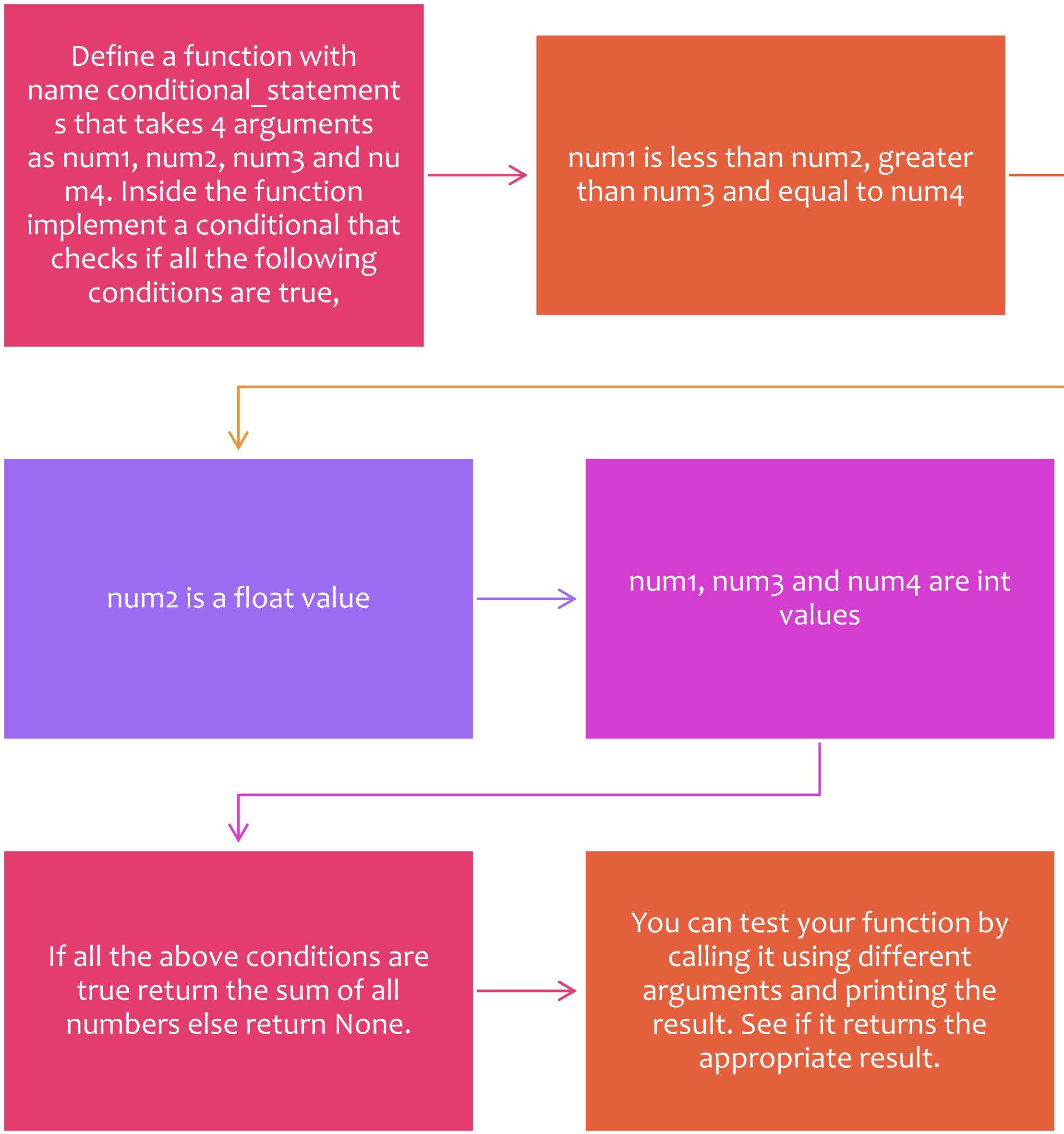
It may be possible nothing is present after the if statements depending upon the requirement.

elif is an abbreviation of "else if." There is no limit on the number of elif statements. If there is an else clause, it has to be at the end, but there doesn't have to be one necessarily.

One conditional can also be nested within another.

```
if x == y:
print('x and y are equal’)
else:
if x < y:
print('x is less than y’)
else:
print('x is greater than y')
```

Lets Get Hands-on The Go



Define a function with name conditional_statements that takes 4 arguments as num1, num2, num3 and num4. Inside the function implement a conditional that checks if all the following conditions are true,

num1 is less than num2, greater than num3 and equal to num4

num3 is the smallest of all the other arguments

num2 is a float value

num1, num3 and num4 are int values

If all the above conditions are true return the sum of all numbers else return None.

You can test your function by calling it using different arguments and printing the result. See if it returns the appropriate result.

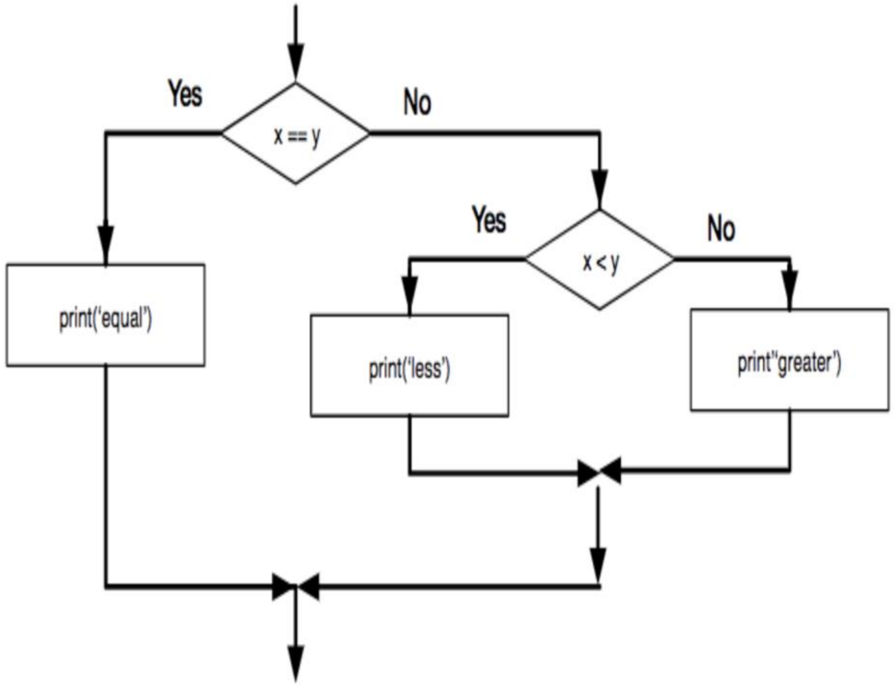
Visualize the Conditional

We can better visualize the the conditional by using the flowchart for this code,

```
if x == y:
print('x and y are equal’)
else:
if x < y:
print('x is less than y’)
else:
print('x is greater than y’)
```

The outer conditional has two branches.

The second branch contains another if statement, which has two branches of its own.
Those two branches are both simple statements, although they could have been conditional statements as well.



Catching Exceptions

Earlier when we used the input and int functions to read and parse an integer number entered by the user, there was a possibility of error if the user entered something which can't be parsed into an int. For eg, if the user entered some str instead of an int.

```
prompt = input("Enter your age\n")
>>Enter your age
>>my age is twenty three
age = int(prompt)
```

As you try to convert the str into an int you receive an error, ValueError: invalid literal for int() with base 10: 'my age is twenty three’
When any such error arises, the program stops immediately and doesn't execute any statements further.

Python provides us with a conditional structure to deal with these types of expected or unexpected errors called "try/except".

Basically, if you know that certain statements in your program may encounter such errors, you may want to add some statements to be executed if an error occurs

These extra statements (the except block) are ignored if there is no error. Let's apply this conditional to the example discussed above

Catching Exceptions CONT…..

```
prompt = input("Enter your age\n")
try:
print (int(prompt))
except:
print('You should have entered a number’)
```

First, the statements inside try block are executed. If everything seems good, it skips the except block and proceeds
If an exception (error) occurs in the try block, Python jumps out of the try block and executes the sequence of statements in the except block.

Handling an error with a try statement is called catching an exception.

In the above example, the except statement prints the error message.

In general, catching an exception gives us a chance to fix the problem, or try again, or at least end the program correctly.

Lets Get Hands-On The Go

Define a function with name except_func which takes one argument with name num and returns its multiplication with itself. If the argument passed during the function call is not valid for multiplication, return a str with content invalid number.

Updating Variables

When it comes to updating the existing variable, we need to re-assign it after updating its value,

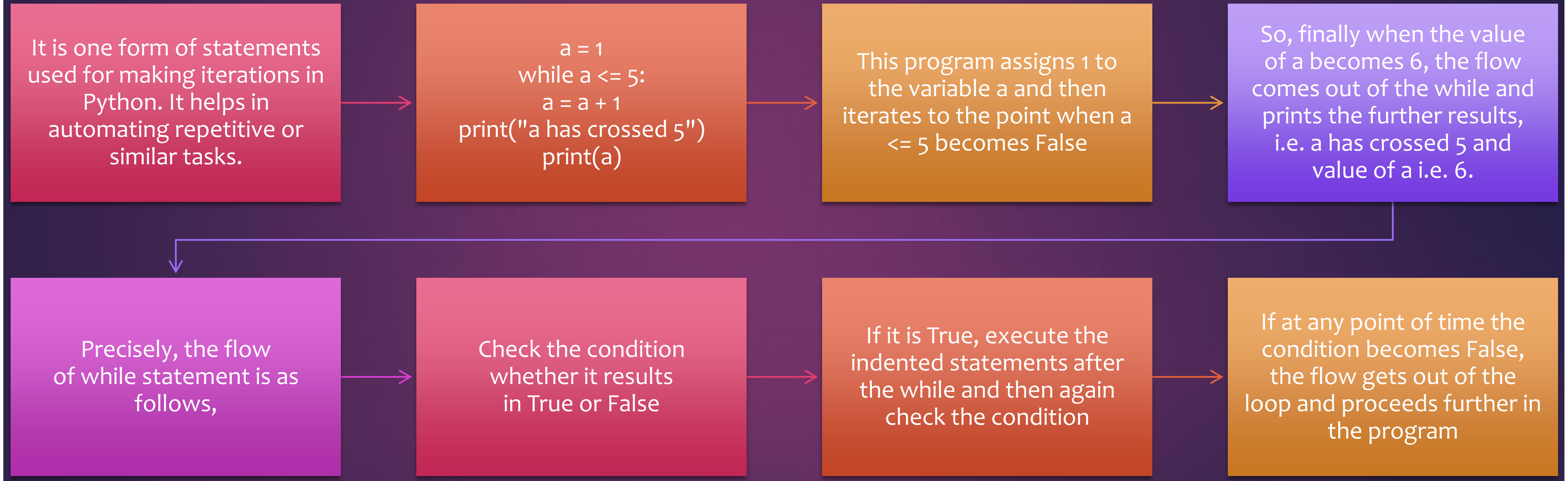
a = a + 1 It gets the current value of a, adds 1, and then updates a with the new value.

If we try to update a variable that is not yet assigned any value i.e. it doesn't exist, you get an error, because Python evaluates the right side before it assigns a value to a.

Therefore, before updating, we need to initialize the variable,

```
a = 0
a = a + 1
a = a - 1
a = a * 2
a = a // 3
```

While Loop



It is one form of statements used for making iterations in Python. It helps in automating repetitive or similar tasks.

```
a = 1
while a <= 5:
a = a + 1
print("a has crossed 5")
print(a)
```

This program assigns 1 to the variable a and then iterates to the point when a <= 5 becomes False

So, finally when the value of a becomes 6, the flow comes out of the while and prints the further results, i.e. a has crossed 5 and value of a i.e. 6.

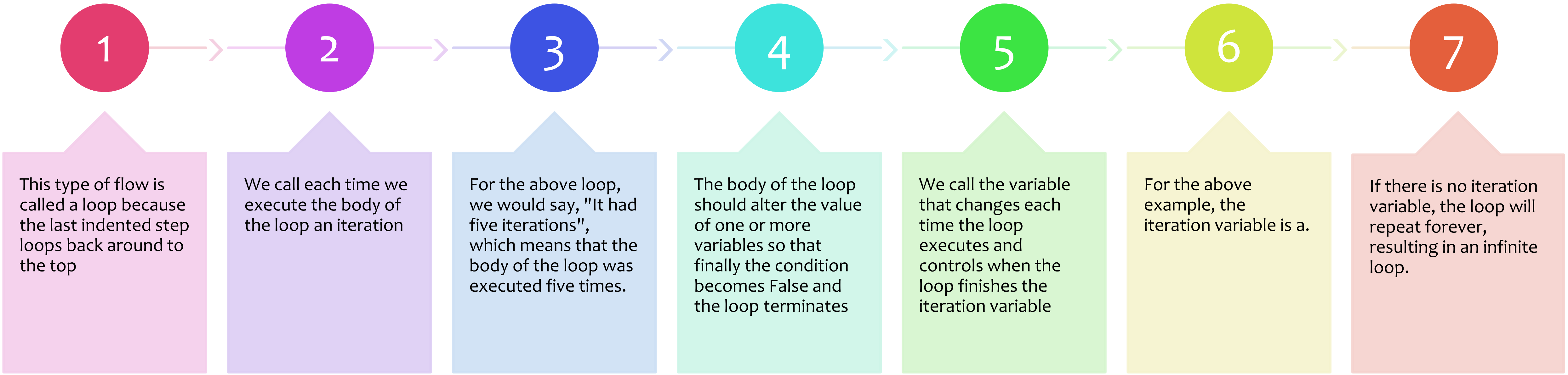
Precisely, the flow of while statement is as follows,

Check the condition whether it results in True or False

If it is True, execute the indented statements after the while and then again check the condition

If at any point of time the condition becomes False, the flow gets out of the loop and proceeds further in the program

While Loop Cont….



1. This type of flow is called a loop because the last indented step loops back around to the top

2. We call each time we execute the body of the loop an iteration

3. For the above loop, we would say, "It had five iterations", which means that the body of the loop was executed five times.

4. The body of the loop should alter the value of one or more variables so that finally the condition becomes False and the loop terminates

5. We call the variable that changes each time the loop executes and controls when the loop finishes the iteration variable

6. For the above example, the iteration variable is a.

7. If there is no iteration variable, the loop will repeat forever, resulting in an infinite loop.

Lets Get Hands-On The Go

Define a function with the name sum_func that takes one argument.

Return -1 if the argument is not int
If the argument is an int and if it is non-negative, return the sum of all integers from 0 to that argument.

In case, the argument passed is negative int, return -1.

Break and Continue

We may have certain cases where we would like to skip the iteration or stop all further iterations and move ahead in the program.

break and continue statements help us ful-fill that purpose as well

In the last exercise, we returned the sum of all integers up to the number passed to the function as an argument.

Suppose, if we would like to calculate the sum of all numbers input by the user

If we take the input one by one, we can give the option to the user to end the process when he/she is done with inputting all the numbers.

```
sum = 0 while True:
n = input("Enter 1 to enter a number, Enter 2
to stop the process\n")
if n is "1":
number = input("Enter the number\n")
try:
sum = sum + int(number)
except:
print("Please enter a valid number\n")
continue
elif n is "2":
break
else:
print("Please enter a valid choice\n")
continue
```

Break and Continue Cont..

Here, we initialized the sum as 0 and then there is a while statement whose condition will always be True (infinite loop).

We give the option to the user at every iteration, whether to enter a new number or to stop entering the numbers.

If the user enters 1, we ask to enter the number

We take the precaution using try except statements to check whether the input given by the user is a valid number or not.

If it is a valid number we add it to the sum, else we ask to enter a valid number.

The continue stops that iteration there and goes to the next iteration.

If the user enters 2 we use break statement to stop the complete loop and the flow goes out.

If the user enters anything other than given choices we ask to enter a valid choice.

Remember whatever input we take from the user, it is in str format,

break and continue process on the innermost loops in case of nested loops.

For loop

We can iterate through a set of things such as a list of words, the lines in a file, or a list of numbers.

When we have a list of things to loop through, we can construct a definite loop using a for statement.

We call the while statement an indefinite loop because it simply loops until some condition becomes False,

whereas the for loop is looping through a known set of items so it runs through as many iterations as there are items in the set.

The syntax of a for loop is similar to the while loop in that there is a for statement and a loop body:

```
numbers = [1,2,3,4,5]
for number in numbers:
print(number)
```

For Loop Cont

Suppose there is a list of numbers from 1 to 5 stored in the variable numbers. (Do not worry, we will cover list data type later).

Then, we iterate over all the elements of the list and print each of the element.

We can do any operation over the elements depending upon the requirement.

Generic Loop process:

Initializing one or more variables before the loop starts

Performing some computation on each item in the loop body, possibly changing the variables in the body of the loop

Looking at the resulting variables when the loop completes

Strings

A string is a sequence of characters. For eg, s = "Python”

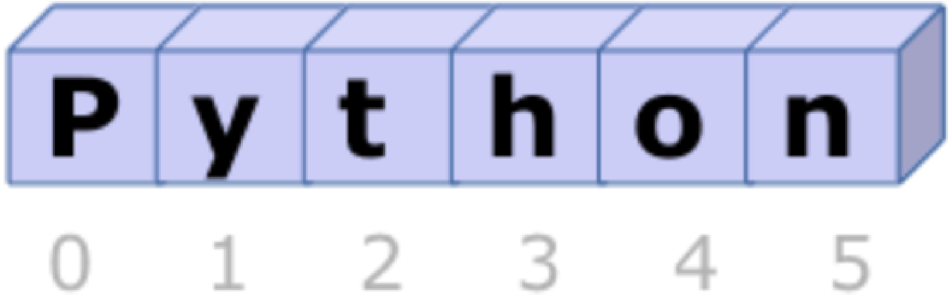
To access a single character, we can use a bracket operator and an index, c1 = s[0] & c2 = s[1]

c1 stores 'P' and c2 stores 'y'. Index starts from 0 for a str (or a list).

Indices must be integers. Try accessing a character using a non int index and observe the error.

Length starts counting from 1 not 0. You can get the length of the string using len function.

len(s) -> It gives the answer as 6.



Lets Get Hands-On The Go

Define a function with the name str_func that takes one argument and returns the last character of the string passed to the function as a parameter.

Define another function with the name sum_str_func that takes one argument and returns the sum of all digits in that number. The argument can be str or an int. You need to take care of that part.

Hint & Solution

Hint:

Make use of def and len function to get the last element of the string.

Make use of a loop like for or while to traverse and get the sum of all digits in the second function. In the case of int, cast it into str.

Solution:

```
def str_func(string):
length = len(string)
return string[length-1]

def sum_str_func(num):
sum = 0
index = 0
length = len(num)
while index < length:
digit = int(num[index])
sum = sum + digit
index = index + 1
return sum
```

String Slices

A segment of a string is called a slice. s = ”DataScienceJourneyWithANR" print(s[1:5])

It prints the segment starting from the character with index 1 to index 4 i.e., ’ataS'. Therefore, it includes the 1st i.e. 1 and excludes the last i.e. 5.

If we don't mention the 1st index before the colon, it considers the segment starting from the beginning of the string

If we don't mention the 2nd index after the colon, it considers the segment to the end of the string

If the 1st index is greater or equal to the 2nd index, it returns an empty string enclosed in the quotation,

print(s[4:4]) - It prints ''. This is also a string with length 0.

We can also use negative indices to get some character from the string. Negative indices count backward from the last. To access the last character we can write, print(s[-1]) For second last character, print(s[-2])

Lets Get Hands-On The Go

Define a function with the name slicing_func that takes an argument (assume string)

It should print the result after slicing the string without mentioning both the 1st and 2nd index.

Hint:

Define a function using def and make use of [:] for the argument.

Solution:

```
def slicing_func(string):
print(string[:])
```

Immutability of Strings

Immutable means once a string is created, it can't be edited or modified.

Then, guess what does this print?

```
a = ”DataScienceJourney"
a = a + ”WithANR"
print(a)
```

It prints DataScienceJourney as per your guess. But, the original string isn't modified here.

a is just a pointer to the str object in the memory which gets assigned a new value when the new string is created.

Remember, here a new string is created by concatenating copy of a and ”WithANR" and then assigned to a again. It has no effect on the original string i.e. " DataScienceJourney ”

So, we can't change a character inside the string by using the indexing,

```
a = "DataScienceJourneyWithANR”
a[1] = 'm'
```

in Operator

in operator helps in looping and counting within some string or list.

```
a = "DataScienceJourneyWithANR is all about results driven
approach learning journey of Machine Learning & Data Science”
count = 0
for letter in a:
if letter == 'l’:
count = count + 1
```

letter is just a variable which iterates over the string from 1st character to the last using the for loop and the in operator. It checks if the letter is 'l' it increases the counter.

Enter the value of count when loop is over

The in operator can also work upon 2 strings and return True or False depending upon whether the 1st string is present as a substring in the 2nd string.

In Operator Cont

Apart from checking for a substring, we can also compare strings. The comparison operators work on strings,

```
string = "machine"
print (string == " machine")
```

Other comparison operators like <, >, <= and >= are useful for putting words in alphabetical order: Try comparing various strings in the notebook and observe the results.

All the uppercase letters come before all the lowercase letters in Python.

String methods

A str object just like other objects in Python has 2 components-
1.) data (string itself), and
2.) methods associated with it i.e. built-in functions associated with any instance of the object.

The dir function lists the methods associated with the object. Try it in the notebook.

To get information regarding any method, you can use help function - help(a.translate)

It provides a simple documentation on the translate method of str object. You can also write as, help(str.translate)

We call a method just like the function call but here the variable or the instance of object is followed by a . and then the method name. For eg, print(a.upper())

print(a)

Parsing Strings

Suppose there is a series of lines consisting of email ids. In case we want to find out the university email ids in a group of various email ids consisting of both Gmail and university ids. For eg,

data1 = hadoopthoughtworks@gmail.com

Basically, we will check if there is a substring starting with @gmail , position = data1.find('@gmail')

The index of @ i.e, 18 gets assigned to the position. It means it has the substring that we are searching for. If the substring is not present it returns -1.

If we have some other information on the line along with email id, we can slice it off using the technique discussed earlier

since we have the position of @ we can find the position of any blank space before and after @ if present and extract the email id out.

We can also mention the position from where we want the find function to start searching. For eg,

position = data1.find('@gmail', 4, 6) - it will search from the index 4 up to index 6.

If we do not mention anything, find returns the position of the substring that occurs for the first time.

Lets Get Hands-On The Go

Define a function with the name email_func that takes a str argument and extracts out and returns the first email id. If there is no email id present return None.

Return only if there is valid email id. An email id is valid if at least one @ is present in it and no blank spaces. Else, you can return None.

Hint:

Define the function using def. Find out the 1st index of @ in the parameter passed (assume string can only be passed) and split the string based on its index and find the blank spaces if present in the 1st part and the 2nd part. Then split based on the blank spaces if present and then concatenate the two halves of the email id.

Solution

```
def email_func(content):
position = content.find('@’)
index = position
if position!= -1:
while index >=0 and content[index]!=' ‘:
index-=1
first_part = content[index+1:position]
ending_position = content.find(' ',position)
if ending_position == -1:
second_part = content[position:]
else:
second_part = content[position:ending_position]
return first_part + second_part
else:
return None
```

Format Operator

The format operator % works on strings for variable substitution purpose.

When applied to integers, % is the modulus operator. But when the first operand is a string, % is the format operator.

```
a = 73
b = "%d" % a
print (b)
```

It prints 73. Basically, it substituted %d with the value of the variable a. the format sequence %d means that the second operand should be formatted as an integer ("d" stands for "decimal").

For more than one format sequence in the string, the second argument has to be a tuple (We will cover it later). Each format sequence is matched with an element of the tuple, in order.

The following example uses %d to format an integer, %g to format a floating-point number, and %s to format a string:

```
b = "%d is an integer, %g is floating-point and %s is string" % (23, 23.34, "Twenty-Three")
print(b)
```

The number of elements in the tuple must match the number of format sequences in the string. The types of elements also must match the format sequences.

Lists

A list in Python is a sequence of values. Unlike strings which are a group of characters, a list can have any data type. It can also be a group of different data types at once which can ba list type as well.

```
a = [12, 3.4, 34, ‘datascience', [2.3, 1.4]]
print(a)
```

A list within another list is nested.

Although a list can contain another list, the nested list still counts as a single element.

A list that contains no elements is called an empty list. We can create one with empty brackets, [].

Lists are mutable unlike strings.

Therefore, we can write something like this, a[1] = 4 print(a)

Lists v/s Strings

For traversing the list, it's just like the strings,

```
a = [1,2,3,4,5]
for number in a:
print(number)
```

This prints the elements within the list. But, if we want to write or update the elements, we need to deal with the indices.

A common way to do that is to combine the functions range and len:

Since lists are mutable, it increases every number by 1 within the list.

```
for number in range(len(a)):
a[number] = a[number] + 1
```

Here, len returns the number of elements in the list and range returns a list of indices from 0 to n-1, where n is the length of the list.

The + operation that concatenates two strings also works with lists,

```
a = [1,2]
b = [3,4]
c = a + b
print(c)
```

It prints the concatenated list as [1, 2, 3, 4].

Lists v/s Strings Cont

For a list, we can use '*' to repeat it a given number of times, a = [1,2,3] c = a * 3

We can also slice the list just like strings, a = [1,2,3,4,5,6,7,8,9] s = (a[2:5])

A slice operator on the left side of an assignment can update multiple elements: a = [1,2,3,4,5,6,7,8,9] a[2:5] = [7,4,9]

List Methods

- Just like strings, you can check for methods for lists as well using dir and help. Some important methods are like append, extend and sort.
- append adds a new element to the list,
- extend takes a list as an argument and appends all of the elements,
- sort arranges the elements of the list from small to large.
- Understand the code and make use of notebook to figure out the results,
- a = [23,12,32,1,2,34,56] a.append(12) print(a[len(a)-2])
- a.extend(a)
- a.sort() print(a[2] + a[7])
- Some other functions are,
- sum() only works when the list elements are numbers to calculate the sum of all elements
- other functions (max(), len(), etc.) work with lists of strings and other types that can be comparable.

Deleting Elements of List

As we know that lists are mutable, we can delete elements within it. There are a number of ways to delete elements from a list. Like, if we know the index of the element to be deleted, we can use pop function,

list = [12,23,43,[2,4,5], 2, 4,5]
list.pop(3)
It deletes the element at index 3.

pop modifies the list and returns the element that was removed. If we don't provide an index, it deletes and returns the last element.

If we do not want the deleted value, we can use del operator which deletes the element but doesn't return anything.

And, we know the element that needs to be removed, we can use remove, list.remove('lab')

The return value from remove is None.

To remove more than one element, we can use del with a slice index:

list = ['DataScience', 'lab', 'provides', 'machinelearning', 'lab']

del list[:]

Lists and Strings

A list is a sequence of values while a string is a sequence of characters. But, a list of characters is not a string.

We can convert a string to a list of characters using list function

The list function splits a string into individual letters. Suppose, if we want to break a long string into words we can use split function to create a list of words,

```
s = "I am learning Python for DataScience”
sp = s.split()
print (sp)
```

We can also split the string on the basis of a certain character or a substring called as a delimiter. It is an optional argument for the split function.

```
s = "I am learning Python at DataScienceJourneyWithANR”
sp = s.split("am")
print(sp)
```

What if there are more than one "am" in the original string? Try that in the notebook.

We can join the split strings using join function,

```
delimiter = "am"
print(delimiter.join(sp))
```

It adds the pivot between the words in the list and prints the original string

Lets Gets Hands-On The Go

Define a function with the name str_list_func that takes an argument (assume str) and interchanges the 1st and last letter of each word in that argument and then returns the resulting string.

Hint:

Split the argument with blank spaces as the pivots and then work upon each element of the list (which is a word) to do the required process.

Solution:

```
def str_list_func(s):
   l = s.split(' ')
   i = 0
   for element in l:
      if len(element) > 1:
         first = element[0]
         last = element[len(element)-1]
         middle = element[1:len(element)-1]
         l[i] = last + middle + first
      i+=1
   return ' '.join(l)
```

Dictionaries

A dictionary is nothing much different from the list. In the list, the indices are integers but in a dictionary, the indices can be of any type. It's like a key mapping to a value.

```
d = dict()
print(d)
```

It prints {} ie.empty dictionary. dict is the function that creates a dictionary.

For adding and accessing the elements we make use of square brackets. Let us make a dictionary containing the number of coins each of 1 rupee, 2 rupees, 5 rupees, and 10 rupees.

```
d['1 rupee coins'] = 10
d['2 rupees coins'] = 5
d['5 rupees coins'] = 6
d['10 rupees coins'] = 12
```

Here the keys are strings and values are integers. But, they can be of any type of your choice. If we print the dictionary again, we can see the key-value pairs printed as follows.

```
print(d)
```

We can also directly create a dictionary by assigning the values in one go,

```
d = { 'one' : 1 , 'two' : 2, 'three' : 3, 'four': 4 }
```

Now if you print d, it may or may not come in the order one, two and three.

In general, the order of items in a dictionary is unpredictable. But that doesn't concern us in any way because the elements of a dictionary are never indexed with integer indices.

Instead, we use the keys to get some value.

Working with Dictionaries

Dictionaries have a method called get that takes a key and a default value as arguments. If the key is found in the dictionary, get returns the corresponding value, else returns the default value.

```
d = {"apples" : 2, "bananas" : 3, "carrots" : 12}
print(d.get("oranges", 0))
```

It prints 0, because "oranges" isn't a key available and 0 is the default value.

If you use a for loop to traverse in the dictionary, it traverses over the keys and using which you can iterate over the values as well.

for fruit in d:

print(fruit) - It prints the keys present in d.

Lets Get Hands-On The Go

Define a function with the name dict_func that takes one argument (assume string) and returns a dictionary with keys as words in the string and values as the number of times those words occur in the string.

You can assume that the string will always have at least one word.

Hint:

Create a list of words using the split function and then iterate over the list to create a dictionary.

Solution:

```
def dict_func(string):
    l = string.split(" ")
    d = dict()
    for word in l:
        if(d.get(word)):
            d[word] += 1
        else:
            d[word] = 1
    return d
```

Tuples

Tuple is also a sequence of values much like a list. The values stored in a tuple can be of any type, and they are indexed by integers.

Tuples are immutable but also comparable and hash able so we can sort lists of them and use tuples as key values in Python dictionaries.

t = (12,323, 'd', [1,23], False)

Here t is a tuple with values of different data types as int, str, list and bool. Each value is separated by a comma.

Even if a tuple has a single element, we need to mention the comma, t = (1,) type(t)

We can also create a tuple using the function tuple. With no argument, it creates an empty tuple:

t = tuple() print(t)

What if we pass an argument which is a sequence like a list, string or a tuple itself?

t1 = tuple("1,2,3,4,5")

t2 = tuple([1,2,3,4,5])

t3 = tuple((1,2,3,4,5))

Working With Tuples

We can access elements of the tuple using square brackets but we can't reassign the value because tuples are immutable. We can also slice the tuples to get multiple elements using the indices as the range.

```
t = (1,3,5,2,54,34,2,34,5)
l = len(t)
print(t[l-2])
```

```
t2 = t[2:5]
```

Comparison with Tuples

Comparisons work with sequences. So, it works well with the tuples.

Python compares the tuples element-wise starting with the 1st element. If they turn out to be equal, it proceeds to the next one. If it finds any difference in elements, it gives the result without considering the further elements.

```
print((0, 1, 2) < (0, 3, 4))
```

Can you tell what will (0, 1, 2) < (0, 3) return?

It doesn't depend on the number of elements in each tuple. It is only concerned with the two elements that it is comparing at one time.

sort function also works the same way by comparing elements inside each tuple when given a list of tuples.

```
l = [(0,23,34), (2,34,23), (1,34,23)]
l.sort()
l.sort(reverse=True)
```

Tuple Assignment

You can have tuple on the left-hand side of the assignment as well.

l = [1,2,3] (a,b,c) = l

Here we assigned a list with values 1, 2 and 3. And, then we assign the list to a tuple containing 3 elements. As a result, a gets assigned with l[0], b with l[1] and c with l[2].

We can also write it without any parenthesis (brackets) and it is equally valid like this,

l = [1,2,3] a, b, c = l

It helps us in swapping elements in a pretty way, a , b = b, a

Both sides of this statement are tuples, but the left side is a tuple of variables. The right side is a tuple of expressions. Each value on the right side is assigned to its respective variable on the left side.

All the expressions on the right side are evaluated before any of the assignments.

The number of variables on the left and the number of values on the right must be the same.

If you are wondering why we used the only list on the right-hand side here. So, for your information, the right side can be any kind of sequence (string, list, or tuple).

Tuples and Dictionaries

There is a function with the name items associated with dictionaries that returns a list of tuples, where each tuple is a key-value pair:

```
d = { "one" : 1, "two" : 2, "three" : 3} k = d.items() print(k)
```

```
l = list(k) print(l)
```

It prints a list of tuples [('one', 1), ('two', 2), ('three', 3)].

Since it is a dictionary, the items are in no particular order.

However, since the list of tuples is a list, and tuples are comparable, we can now sort the list of tuples

Converting a dictionary to a list of tuples is a way for us to output the contents of a dictionary sorted by key.

We can also do multiple assignments with dictionaries, For eg,

```
for key, value in list(k): print(key, value)
```

Python - Understanding Files

The data on our systems is stored in the files. Python helps us handle files as well. For reading or writing to a file, we must open the file.

```
f = open("file_name.txt")
```

If the file opens successfully, the operating system returns us a file handle. The file handle is not the actual data contained in the file, but instead, it is a "handle" that we can use to read the data. We are provided with a handle if the requested file exists and we have the proper permissions to read the file.

If there is no file with the name we mentioned, open will fail with a traceback and we will not get a handle to access the contents of the file.

The file should be stored in the same folder that you are in when you start Python. In that case, i.e. if there is any chance of file not being present, we can try and except to deal with the exception.

A text file is just a sequence or lines or str in Python. Each line is separated by a "\n".

In Python, we represent the newline character as a backslash-n in string constants. When we try printing the "\n" in the string, it breaks the string into different lines.

```
print("DataScience\nx\nlab")
```

So, when we are reading lines, we need to imagine that there is a special invisible character '\n' called the newline at the end of each line that marks the end of the line.

Reading Files

We can use a for loop to read a file's data using it's handle, f = open("/data/python_sample_file").

You also need to close the handle once it is of no use by f.close().

So there is a better way to read the file using with so that you do not have to worry about closing the handle. In this case, the handle is only valid only within the block of with,

We need to place the path of the file inside the brackets and the file handle is stored in f. Try it in the notebook and see the results. It should print each line in the file and then print all the all lines as concatenated one

We can also read the whole file into one string using the read method on the file handle.

content = f.read()

Searching in Files

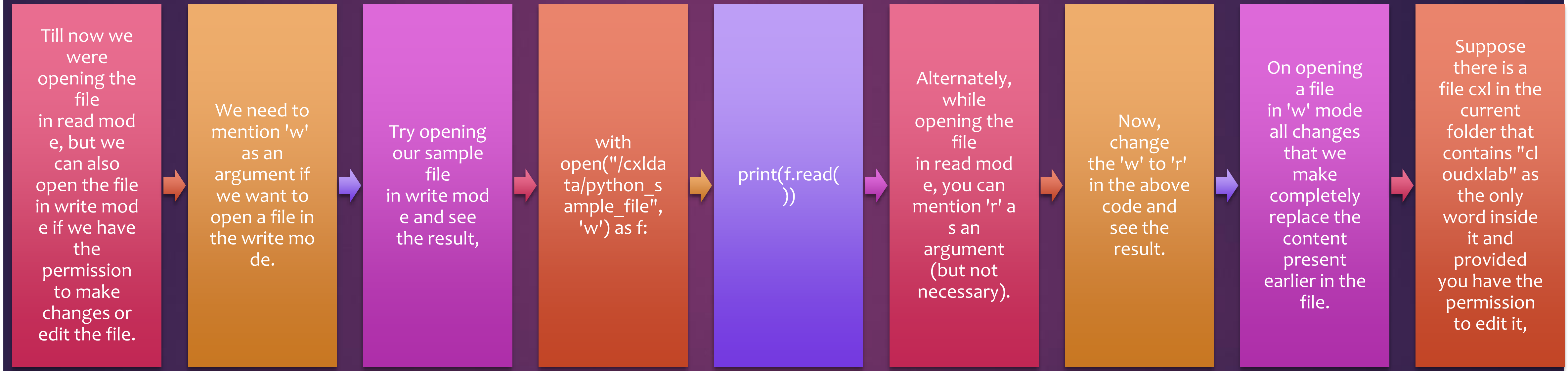
We can search in the file line by line or by creating a single string for the whole content of the file.

```
check = 0
with open("/data/python_sample_file") as f:
    for line in f:
        if(line.startswith('w')):
            check = check + 1
```

startswith function checks if the string is starting with the argument mentioned in the brackets.

Basically, here were searching for the lines in the file which are starting with 'w'.

Writing Files



Till now we were opening the file in read mode, but we can also open the file in write mode if we have the permission to make changes or edit the file.

We need to mention 'w' as an argument if we want to open a file in the write mode.

Try opening our sample file in write mode and see the result,

with open("/cxldata/python_sample_file", 'w') as f:

print(f.read())

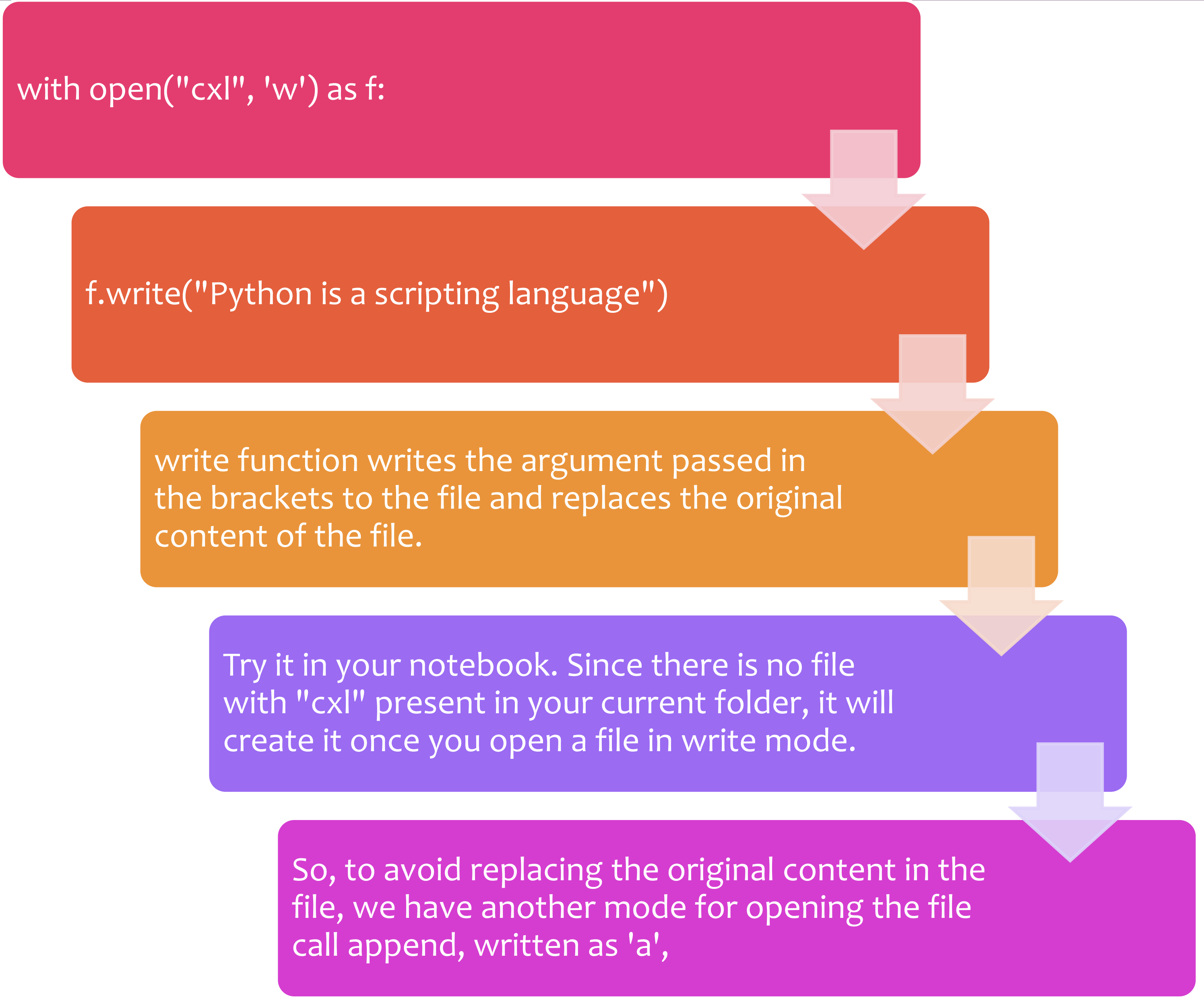
Alternately, while opening the file in read mode, you can mention 'r' as an argument (but not necessary).

Now, change the 'w' to 'r' in the above code and see the result.

On opening a file in 'w' mode all changes that we make completely replace the content present earlier in the file.

Suppose there is a file cxl in the current folder that contains "cloudxlab" as the only word inside it and provided you have the permission to edit it,

Writing Files Cont..



```
with open("cxl", 'w') as f:
```

```
f.write("Python is a scripting language")
```

write function writes the argument passed in the brackets to the file and replaces the original content of the file.

Try it in your notebook. Since there is no file with "cxl" present in your current folder, it will create it once you open a file in write mode.

So, to avoid replacing the original content in the file, we have another mode for opening the file call append, written as 'a',