**Python**

**Functions:**

* **Logical operators: And**; **or**; **not**; (returns True if the argument is false and false when it is true)
* **type**() : will tell you the type of function in the brackets
* **Range**( start, stop, [step]): produces sequence of integers
* **Separator**: sep = ‘\_\_’ : put at the end of a string; for separators between words e.g. “-“ will be put between every word
* **Int()** ; int function takes a certain value and converts it to integer
* **Float**: converts integers and strings (when possible) into floating point numbers.
* **Str():** same as float but for string
* **Input()**: variable = input(prompt) e.g. name = input(‘What is your name?’) what is your name <— then type your name and variable assigned to name.
* **If**: e.g. : x = int(input(‘Enter the value of x: ‘))

if x > 0

print (‘X is positive’)

* **if**-**else**: it is the same as if but with with a secondary condition, that is an else statement.
* **If-elif-else:** it is a set if else-if, that is, allows multiple ifs.
* **In**: use when you want to include a list of items.
* **Random.randint()**: returns a random integer value between 2 higher and lower limits (including both limits) provided as 2 parameters: randint(beg,end). **it is necessary to use import random to use this function.**

**Boolean expressions**: (return true or false statements)

* == : Equal to: e.g. 4 == 4 —> **true**; 4 == 73—> **false**
* != : different from
* > greater than
* < smaller than
* >= greater than or equal to
* <= less than or equal to

**Mathematical operators:**

* \* : multiplication
* // : floor division (divides two numbers and rounds down to an integer: >>> 56/3 = 18.6666666
* \*\* : exponential : elevates a number to a power
* % modulus: divides two integers and returns the remainder of the division e.g. 16%3 = 1. to create or use multiples in python, use the **modulus %,** making it equal to 0 e.g. for a multiple of 7, if num%7 == 0:

**Loops (general):**

* Loops controlled by a condition use a true/false condition to control the number of repetitions, while a loop controlled by a counter is repeated a given number of times.
* **pass** **statement** in loops: to avoid syntax errors when running an incomplete program. Simply does nothing

**The while loop:**

* **While** statement is used to write a loop controlled by a condition. In a while loop, the repetition of the code is executed until the while condition is true.
* **As long as the condition is true, the block of statements under the while condition is re-executed.** When the condition becomes false, the program execution continues outside the loop.
* **Example:**

Countdown = 5

While countown >= 1:

**Print(countdown)**

**Countdown = countdown – 1**

Print(“go”)

Output : 54321 Go

* The **bold** are the while statements.
* The countdown variable is defined and initialized before the while loop (i.e. the control of the while condition starts with the countdown variable equal to 5).
* The countdown variable is then decremented inside the loop **at each iteration (every time the condition is true). As long as this condition is true, the sequence of statements is indeed reexecuted and the last statement decrements the countdown variable by one unit.**

**The infinite while loop:**

**(The while-if loop)**

* A while loop can be executed infinite times, until a specific condition occurs. The while True syntax can be used instead of the simple while when the loop doesn’t have an easy end to define. **Thus, we don’t have to initialize a variable before entering the loop, but one needs to go add an if containing the exit condition for the loop, and break is the statement that is executed when the if condition occurs. The break statement is used to force the exit from the loop at a given time.**
* P.**88 for example**

**The for loop:**

* **For** loop is used to write a loop controlled by a counter. In a for loop, the repetition of the code is executed a predefined number of times.
* In for loops, a counter variable is used to count the number of loop iterations. The for clause together with the name of the *counter variable ,* **the in keyword**, followed by the colon to close the statement. A sequence of values stands within the square brackets and each value is separated from the other by a comma (a list).
* A statement or a set of statements that is executed at each loop run.
* Hence a for statement is executed as follows: the first value of the list is assigned to the counter variable and the statements within the block are executed. At this point, the next value of the list is assigned to the counter variable and the statements within the block are re-executed again. This process goes on until the last value of the list is assigned to the counter variable.

**Example**: same as before but now with for loop:

For countdown in [5,4,3,2,1]

Print(countdown)

Print(‘Go’)

Output: 54321 Go.

**The range function:**

Range(start, stop, step)

The range function is often used in conjunction with for loops instead of a list to iterate a set of integer values. For instance, the range function range(1,5,2) can be used in a for loop to print the integer values 1,3 as follows:

For I in range (1,5,2):

Print(i)

For a negative step: i.e. the countdown function

For i in range (5,0,-1):

Print(i)

Print(‘stop!’)

The range function can also be adopted inside while loops.

I = 1

While I in range(1,5)

Print(i)

I = I + 1

This program prints the integers 1,2,3,4.

THE RANGE FUNCTION WORKS ONLY WITH INTEGER VALUES, NOT FLOATS.

**Nested loops:**

Example:

For a in range(1,4):

Print(’\*\*\*\*\*’,a)

For b in range (3,0,-1):

Print(‘’,b)

Print(‘stop’)

Output:

\*\*\*\* 1

1

1

1

\*\*\*\* 2

2

2

2

\*\*\*\* 3

3

3

3

Stop

* **Break** statement: **break** lets you terminate the execution of the nearest enclosing loop, and to move the program control outside the loop. i.e. the loop stops when you have done what you want to do.

e.g. for friend in [a,b,c,d,e,f]

if friend == ‘d’

print(“I’ve found him”)

break

print(“Now I’m reading:”, friend)

print(“END”)

Output:

Now I’m reading: a

Now im reading: b

I’ve found him!

End

* **Continue** statement: rejects all the remaining statements in thecurrent iteration and moves the control back to the top of the loop

**Practice 5:**

**Coin tosses:**

import random

heads = 0

tails = 0

n\_tosses = int(input('How many tosses you want to simulate? '))

for toss in range(n\_tosses):

random\_nr = random.randint(0,1)

if random\_nr == 0:

heads = heads + 1

else:

tails = tails + 1

print('Number of heads tosses =', heads, '\nNumber of tails tosses =', tails)

**divisors:**

num = abs(int(input('Enter an integer number: ')))

if num == 0:

print('Invalid number')

else:

divisors = 0

for n in range(1, num+1):

if num % n == 0:

divisors = divisors + 1

print(n)

print('We found', divisors, 'divisors')

**Sum of all numbers from 1 to a given number entered by the user (Exercise 11.13 or something like that)**

total = 0

x = int(input("Please enter an integer value, i.e. a number "))

print("The number you entered is", x)

for i in range(1,x+1):

total = total + i

print("The sum of all numbers from 1 to", x, "is", total)

* **In online compiler, at times one will have to Print(“”) i.e. print an empty string as the first number in a loop list will not appear. Questions in the exam should take care of this.**

**Lab 8:**

**We want to show on the screen all integers from 0 to a number entered by the user (this last excluded). The loop is set to perform a maximum of 100 iterations.**

num = int(input('Insert an integer number: '))

for i in range(100):

if i < num:

        print(i)

    else:

        break

**The following program written in Python is designed to calculate the total number of digits of an integer number entered by the user. For example, the number of digits of 428 is 3, whereas the number of digits of 40782 is 5.**

num = input("Enter an integer number: ")

num\_int = int(num)

digits = 0

while num\_list > 0:

    num\_int = num\_int//10

    digits = digits + 1

print("The number", num, "is made of", digits, "digits")

**Chapter 7 - custom functions**

* Header and body: def function\_name(parameter1, parameter2,…):

Instruction

Instruction

Instruction

* One can either set no parameters, thus calling a function such as quote(), and expecting it to return simple prints(“bla bla bla’)
* Or one can insert parameters in the function such as
* def multiply(num1,num2):
  + Print(“the result of the multiplication is:’, num1\*num2)
* One can either have **optional** or **mandatory** parameters
* **Mandatory** parameters must be specified first. These are parameters that the user has to put into the function.
* **Optional** parameters are parameters that the user can choose to put into the function. If he chooses not to, a **default value** is put in for him.
* **E.g. def calculate(par1, par2, par3=5,par4=0):**

Result = (par1\*par2)/(par3 + par4)

Print(result)

* The arguments are passed by position. If we wanted to pass an argument to par4 but not par3, we would use keyword arguments, that is, placing your optional parameter wherever you want using the general form **parameter\_name = value.**
* **Using keyword arguments, we can call the function specifying the arguments in the order we prefer, without respecting the positions of the definition.**

**Productive functions:**

* The functions we have seen until now are called **void functions** and cannot be stored into a variable. By using the **return** statement, one can use the result of the function and assign it to a variable.
* E.g.:

Def calculate(num1,num2,num3):

Result = num1 + num2 + num3

**return** result

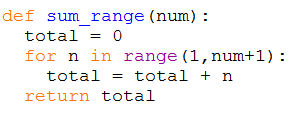
>>> x = calculate(4,5,2)

>>> x

10.0

. Global variables, placed outside a function, can be used within the whole program. **Local variables** on the other hand, are defined within the function and therefore cannot be called to use in IDLE shell.

**Functions with loops and conditional statements:**



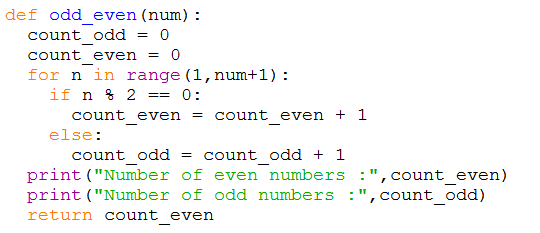
We must pass to the function an integer as an argument, which is later used in a for loop to calculate the sum of all the numbers between 1 and the number passed as an argument.

E.g. >>> sum\_numbers(8)

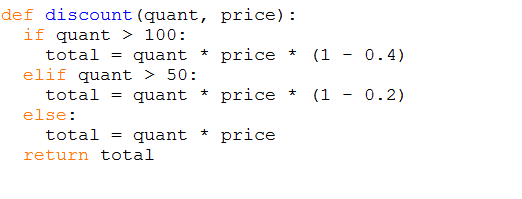
36

The value returned by the function (36) is the sum of all the numbers between 1 and 8.

Other example:



Other example:



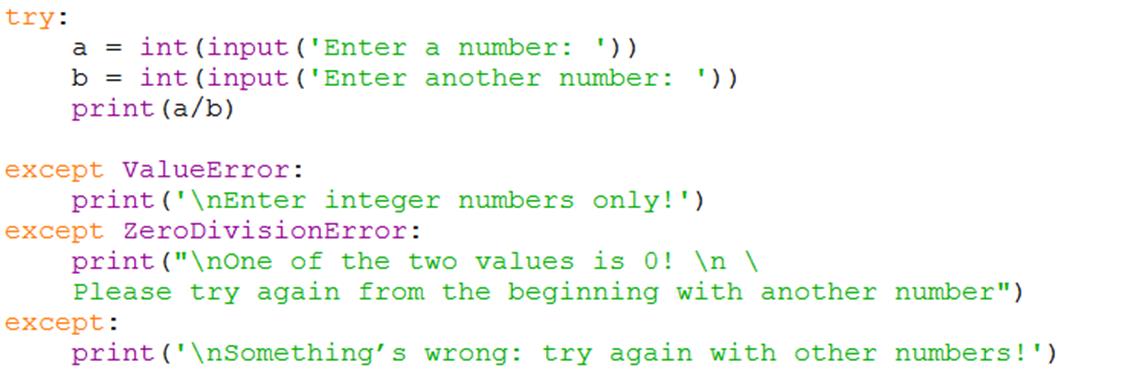
**. The docsting helps you understand what the function does.**

**Exception handling:** **TRY and EXCEPT**

* An exception is an event triggered by an error. If we expect an error of some kind, we can write some code to handle it.
  + Syntax errors: error In how code is written
  + Runtime errors: error in code even if syntax is correct e.g. division by 0 or name not defined.
  + Semantic errors: when program is executed without producing error messages, but the results are not the correct ones – inconsistent or not expected – hard to find.

**Example:**

**create a program that performs a simple division between two integer numbers entered by the user. The program must prevent from any kind of errors occurring while entering numbers:**



**Practice session 6:**

Create a new Python file called sum.py and create a function called harmonic\_sum, which must calculate the value of the following series:

A picture containing text

Description automatically generated

For example, considering N = 6 and *α* = 2 you will obtain:

Diagram, schematic

Description automatically generated

In particular, the function must:

* have as mandatory parameters *N* and *α*, that are the maximum number to be considered for the series and the exponent to be used
* return the result of the sum of all the fractions to the calling instruction

Test the program calling several times the function harmonic\_sum with different values.

**ANS:**

**def harmonic\_sum(N, a):**

**total = 0**

**for n in range(1, N+1):**

**total = total + 1/n\*\*a**

**return total**

**Chapter 8:**

+ operator:

* Concatenates 2 strings
* Text = “after” + “noon” 🡪 afternoon
* Text1 = “after”
* Text2 = “noon”
* Text3 = text1 + text2
* Print(text3) 🡪 afternoon
* One can’t concatenate two different types of data. To do this, use a converter.

\* operator:

* X = ‘python’
* Y = x \* 4
* Print(y)
* PythonPythonPythonPython

In operator:

* X = “rome is in the Italian capital.”
* “capital “ in x 🡪 true

Not in:

* ‘capital’ not in x
* False
* “Paris” not in x
* True

Is/is not operators:

* X = “Rome is in the Italian Capital
* Y = “Paris is in the French capital
* X is y 🡪 false
* X is noy y 🡪 true

Lists:

* Example of a list:
  + Numbers = [50,100,300,200,400]
  + Italian\_cities[‘rome’,’florence’,’modena’]
* **list function: converts certain object types, such as a text string or an iterable object into a list.**
* **E.g.**
* **Text = list(“milan”)**
* **Print(text)**
* **[‘m’,’I’,’l’,’a’,’n’]**
* **Can also do this with range functions:**
* **Numbers = list(range(1,7))**
* **Print(numbers)🡪[1,2,3,4,5,6,]**

List operations:

* The + operator concatenates two lists respecting the order of insertion.
* Unlike strings, lists can be created with different data types.
* The \* operator: cities = [‘Florence’,Florence’]
  + X = cities \* 3
  + Print(x) 🡪 [‘florence’,’florence’,’florence’,’……
* In and not operators:
  + Destinations = [‘rome’,’milan’,’venice’]
  + ‘Naples’ in destinations
  + True
  + ‘Turin’ not in destinations
  + True

**Indexing**

* Way to access single element of sequence. 01234567… from left to right and -1,-2,-3,-4… from right to left.
* Possible to access single elements of a list: **sequence[index] where index is the number of the element you want to extract.**
* we can also modify the list.
  + E.g.
  + Where capitals = [‘rome’,’london’,’berlin’]
  + Capitals[2] = ‘madrid’
  + Print(capitals)🡪 now instead of berlin you will get madrid.
* While loops to show on screen all elements of a list:

Capitals = [‘rome’,’london’,’pasis’]

Index = 0

While index < 3:

Print(capitals[index])

Index = index + 1

* Print(capitals[-4]) 🡪 ‘London’
* While loops negative order:

Capitals = [‘rome’,’london’,’pasis’]

Index = -1

While index >= -3

Print(capitals[index])

Index = index – 1

Output: paris, london, rome

* **To show individual characters:**
* **Text = ‘Python basics’**
* **Text[10]**
* **‘I’**
* **Strings are immutable: can’t change individual characters of the string stored in text variable.**

**Slicing:**

* To select multiple elements of a sequence at the same time Python allows us to use a technique called slicing, very similar to indexing. It allows us to select a portion of the sequence (called slice) using the indexes of the elements of the sequence. To select a slice of the sequence use:

**Sequence[start:end:step]**

* With this it is possible to select a range within the sequence. Start is included and end is excluded, like the range fct.

**A picture containing table

Description automatically generated**

To select and store in a variable the first six letters of the string 'PYTHON BASICS', both positive and negative indices can be used:

text = 'PYTHON BASICS'

x = text[0:6]

y = text[-13:-7]

|  |  |  |
| --- | --- | --- |
| **Instruction** | **Meaning of slicing** | **Result** |
| text[2:8] | Selects all elements with index from 2 (included) to 7 (the element with index 8 is excluded) | 'THON B' |
| text[:9] | Selects all elements from the first one to the one with index 8 (the element with index 9 is excluded) | 'PYTHON BA' |
| text[5:] | Selects all elements with index from 5 (included) to the last one | 'N BASICS' |
| text[:] | Selects all the elements of the sequence, from the first one to the last one | 'PYTHON BASICS' |

sequence[start:end:step] is used identically to the range function to select the step of stuff u wanna extract.

To have a negative step parameter it is necessary that also the start and end parameters are negative: in this case the selection of the elements takes place from right to left, and also the step parameter, whose meaning doesn’t change, must adapt.

Alternatively, it is possible to use a negative value of the step parameter with positive start and end parameters, but in this case they must be inverted, starting from end.

**Slicing works with lists:**

* E.g.: Consider for example the following list, to which we applied a slicing with step2:
* Capitals = ['Rome', 'London', 'Berlin', 'Tokyo', 'Moscow', 'Lisbon', 'Paris']
* Capitals[0:5:2]
* Result: ['Rome', 'Berlin', 'Moscow'].
* **Replacing elements simultaneously:**
* We want to replace the first two elements of the Capitals list with 'Oslo' and 'Dublin'. With indexing, it is possible to replace the first and the second element in two steps:
* With indexing we can do:

Capitals[0] = 'Oslo'

Capitals[1] = 'Dublin'

>>>Capitals

['Oslo', 'Dublin', 'Berlin', 'Tokyo',

'Moscow', 'Lisbon', 'Paris']

* With slicing:

Capitals[0:2] = 'Oslo', 'Dublin'

Capitals[0:2] = ['Oslo', 'Dublin']

>>>capitals

['Oslo', 'Dublin', 'Berlin', 'Tokyo',

'Moscow', 'Lisbon', 'Paris']

**Functions and methods of strings:**

* General syntax: function\_name(text) where text contains a string.
* Syntax of methods: **object.method\_name(arguments)**

Main built-in funcitons for strings:

|  |  |
| --- | --- |
| **Function** | **Description** |
| len() | Returns the length of a string, that is the number of characters in the string |
| max() | Returns the largest value in a sequence. When applied to a string, the function returns the highest alphabetical character, considering the letter «a» as the lowest value and the «z» letter as the highest value from an alphabetical point of view |
| min() | Returns the smallest value in a sequence. As for the max() function, when it is applied to a string, the function returns the minimum alphabetical character |

Len function: calculates the length of a string:

e.g.:

>>> quote = 'Stay hungry. Stay foolish.'

>>> len(quote)

26

Example with loops:

for c in range(0, len(quote)):

    print(quote[c])

    c = c + 1

the result will show all the characters of the stirng, in the original order, one below the other.

|  |  |
| --- | --- |
| **Method** | **Description** |
| .upper() | Returns a copy of the string with all alphabetic letters converted to uppercase. Any character that is already uppercase is not modified |
| .lower() | Returns a copy of the string with all alphabetic letters converted to lowercase. Any character that is already lowercase, is not modified |
| .capitalize() | Returns a copy of the string where the first character is upper case and all the other characters are lowercase |
| .strip() | Returns a copy of the string with all leading and trailing whitespace characters removed. The method also accepts the car argument, in which a character to be removed instead of the space can be specified |
| .find(sub) | Searches in the string for the substring specified in the sub argument and returns the index of the first occurrence found. Two other arguments, start and end, are optional and allow to restrict the search to a substring of the main string.  If the substring is not found, the method returns -1 |
| .replace(old, new) | Returns a copy of the string with all the occurrences of the substring old are replaced by the substring new |
| .startswith(prefix) | Returns True if the string ends with the string specified in the prefix argument, otherwise it returns False. Two other arguments, start and end, are optional and allow to restrict the search to a substring of the main string |
| .endswith(suffix) | Returns True if the string ends with the string specified in the suffix argument, otherwise it returns False. Two other arguments, start and end, are optional and allow to restrict the search to a substring of the main string |
| .count(sub) | Returns the number of occurrences of the substring sub in the string |
| .split(iterable) | Splits a string in words and returns a list of strings, considering the space character as the separator between the words. It is possible to specify two optional arguments, separator and maxsplit, that allow to specify any text string as a separator and to indicate the maximum number of splits of the string to be made. If the separator argument is omitted, space is the default value. If the maxsplit argument is omitted, the default is -1, which means that the splits can be infinite.  When separator not space example:  music2 = 'Blues, Rock, Pop, Soul, R&B, Jazz, Metal, Folk'  music2.split(", ")  ['Blues', 'Rock', 'Pop', 'Soul', 'R&B', 'Jazz', 'Metal', 'Folk'] |
| .join(iterable) | Returns a string obtained concatenating all the elements of an iterable object (§ 8.4) containing only strings. A string containing the delimiter to be used must be specified: join is a method of this string, whereas the iterable object must be indicated between the parentheses of the method  e.g.  myseparator = '/'  x = myseparator.join(music3)  The x variable contains the following string:  'Blues/Rock/Pop/Soul/R&B/Jazz/Metal/Folk' |

List functions:

|  |  |
| --- | --- |
| **Function** | **Description** |
| len() | Returns the length of a list, that is the number of elements of the list |
| list() | Converts an iterable object into a list |
| max() | Returns the largest element of a list. When the elements are strings, the function returns the string beginning with the highest alphabetical character or characters |
| min() | Returns the smallest element of the list. When the elements are strings, the function returns the string beginning with the minimum alphabetical character or characters |
| sorted() | Returns a new sorted list with elements sorted in ascending order. It doesn’t modify the original list, but is a throwaway function. Usa e butta |
| sum() | Returns the sum of all the elements of a list |

**Main list methods**

|  |  |
| --- | --- |
| **Method** | **Description** |
| .append(element) | Appends new elements to the list.  e.g.  >>> music5.append('Reggae')  >>> music5  ['Rock', 'Dance', 'Reggae'] |
| .insert(index, element) | Inserts the desired element in the position specified by the index parameter |
| .remove(element) | Searches for the specified element in the list and removes the first occurrence |
| .pop([index]) | Removes and returns the element with the specified index. If we omit index, it removes the last element of the list |
| .extend(list2) | Appends the specified list (list2) to the list |
| .index (element) | Searches for the specified element within the list and returns its index. If the element occurs more times, returns the index of the first occurrence |
| .sort() | Sorts the elements of the list in ascending order |
| .reverse() | Reverses the elements of the list |
| .clear() | Removes all the elements of a list |
| .count(element) | Counts how many times a given element is found in the list |
| .copy() | Returns a copy of the list |

**w**

**Practice 7:**

Text, letter

Description automatically generatedDictionaries: instead of an index, we use keys

**Key : value**

A tuple is a list and you can put whatever you want in it i.e. any type. It is immutable. Tuples needs less ram memory than lists.

Separate example:

S = “Hello World”

L = list(s)

For elem in L:

Print(s)

**Dictionaries:**

* In Python, a dictionary is an object that contains a collection of data or items
* Element are made of two parts: a key and a value. The keys are unique and can be any kind of immutable objects. Values can be any kind of object, mutable or immutable.
* To create a dictionary it is necessary to write in curly brackets { } the elements divided by commas (,). Every element is made of a **key** followed by a colon **(:)** and a **value**
* Alternatively you can use the dict function, which creates a new dictionary with no elements
* Dictionaries are mutable objects

**Keys and values: how to create dictionaries**

• The order of items in a dictionary is unpredictable, for this reason we cannot use indexing as we can do in sequences

• To retrieve a value stored in the dictionary we use the corresponding key : **dict\_name[key]**

• In order to add key-value pairs: **dict\_name[key] = value**

myDict = {10: “K”, “A”:0, 2.5:(1,2,3), ”Greetings”:”hello”}

print(myDict)

print(“myDict[‘A’], (“myDict[“A”])

myDict[5] = 500 **: to assing a new value**

myDict[“A”] = 48 🡪 changing existing content. i.e. assigning a new value to A.

print(myDict)

**>>> {10: “K”, “A”:48, 2.5:(1,2,3), ”Greetings”:”hello”,5:500}**

Operations with dictionaries:

• **in** allows to check the presence of a key  
• **del** removes a key-value pair from a dictionary

• **len** returns the number of **key-value pairs** (i.e. the couples of key and values) e.g.len(myDict) -- > **elements are made by couples.**

Text

Description automatically generated

**Traversing dictionaries**

• The keys of a dictionary can be traversed using a for loop:

>>> address\_book = { 'pippo': '555-123456', 'pluto': '555-654321' } >>> for key in address\_book:

print(key)

* Output: pippo pluto: it shows us the **keys**

Elements:

🡪 to show us **elements**:

* For elem in myDict:

Print(myDict[elem])

🡪 special method to print elements (using methods in chart above):

* For elem in myDict.**values**():

Print(elem)

• We can also traverse key-value pairs using the for loop of the list dict.**items**()

>>> **for myKey, myValue in myDict.items():**

**Print(myKey,myValue)**

**Practice 7 worksheet**

1. Def top\_cities(myList, num\_cities = 5):

Import random

Top = {}

For key in range(1,number+1):

Rnd\_num = **random.randrange**(len(myList))

Town = myList[rnd\_num]

Top[key] = town

Return top

Y = Top\_cities(destinations,7)

Print(y)

**Format function:**

Format(99.87, ‘.1f’) 🡪 99.9

Isinstane(): takes a variable and