CONTROL STATEMENTS AND LOOPS

Why `if-elif-else` statements?

Let's answer this question with a simple example. Suppose you receive a lot of emails and for that reason, you decide to separate them into three classes Important, Promotions and Spam. Now you know that if mails come from email addresses (let's say) A, B and C, they must belong to the Important category. Similarly, you also know the email addresses which contribute to Promotions and Spam categories. A simple pseudocode for tackling the above problem can be written as:

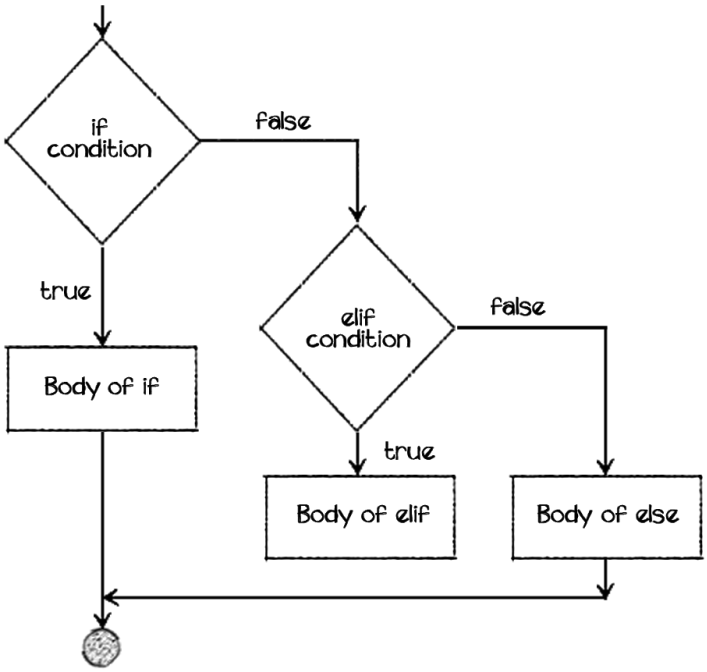
Pseudocode

```
if email is A or B or C:
    email is Important

elif email is D or E or ....:
    email is Promotions

else:
    email is Spam
```

The if-elif-else condition is used where there are different possible actions for different conditions.

An `if` statement can be followed by optional `elif` and `else` statements which execute when the boolean expression for the previous condition is `False`. Below shown is a flow of how this conditional works.

Workflow



Example

Now, let us look at how we implement the same in Python. Below is a code snippet:

```
x=5

if (x>10):
    print("Your value is greater than 10. ")
elif (x>7):
    print("Your value is between 7 and 10.")
elif(x>1):
    print("Your value is between 1 and 7.")
```

Nested If

You can also use nested if-elif-else statements (means `if-elif-else` within `if-elif-else` statements). An example is shown below where we first check if the number is first checked if it is greater than `0` and if yes then it is checked for two more conditions: (a) equal to zero and (b) greater than zero.

```
# variable initialized
x=5

# check if number is greater than or equal to zero
if x >= 0:
    # check if it is equal to zero or greater than zero
     if  x == 0:
        print('Number is zero')
     else:
        print('Number is positive')
else:
      print('Number is negative')
```

Can you guess the output of the above code?

Output

```
'Number is positive'
```

You guessed it right! It is the second print statement. You can also use nested `if` statements to solve your problem.

Why `while` loops?

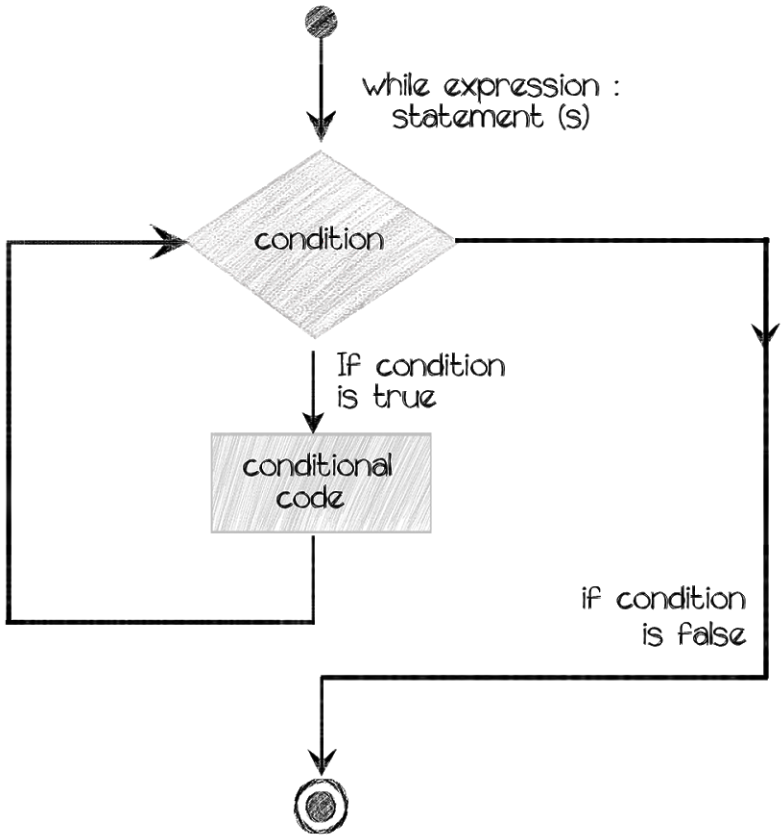
The `while` loop is used to repeat a section of code an unknown number of times until a specific condition is met. For example, say you want to know how many times a given number can be divided by `2` before it is less than or equal to `1`. Below is the pseudocode for the `while` loop:

Pseudocode

```
get our number
set our initial count to 0
while our number is greater than 1
    divide the number by 2
    increase our count by 1
end
```

It repeats a statement or group of statements while a given condition is `TRUE`. The flow of a `while` loop with syntax is shown below:

Workflow



Translating the pseudocode given above that calculates the number of times a given number can be divided by `2` before it is less than or equal to `1`

Example

The below example is an example of the implementation of a `while` loop in Python.

```
number = 5
count = 0
while number > 1:
    number = number / 2
    count  += 1

print(count)
```

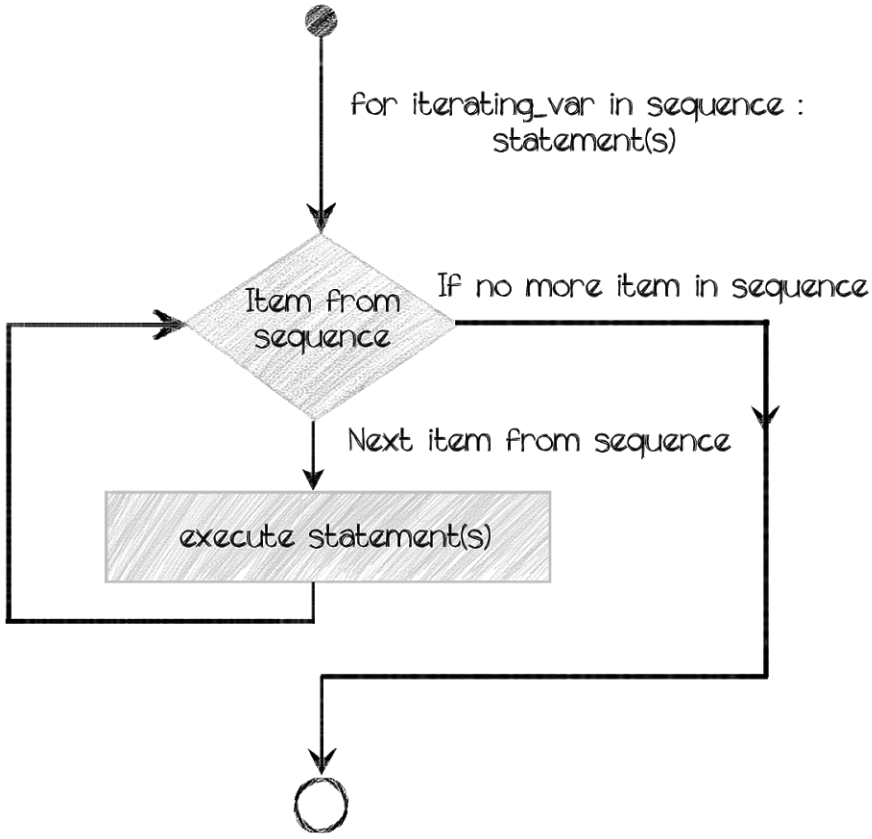
In the code block above:

- First, initialize a variable `count` and set its value to 0
- Now, we start the `while` loop where we check and perform the operation (dividing it by 2 and incrementing `count` by 1) until the variable is less than or equal to 1

Why `for` loops?

A `for` loop is used to repeat a specific block of code a known number of times. For example, if you want to check the grade of every student in the class, we loop from 1 to that number. In general, when the number of times is not known beforehand, we use a `while` loop.

Like all loops, `for` loops execute blocks of code over and over again. The advantage to a `for` loop is we know exactly how many times the loop will execute before the loop starts.



Example

```
# sum numbers from 1 to 10
total = 0
for i in range(1, 11):
    total = total + i
```

Here,

- Initialize a variable `total` and set it to `0`
- What the `range()` does is it returns a sequence of integers within a range. Its syntax is `range(start, stop[, step])` where
    - start: integer starting from which the sequence of integers is to be returned
    - stop: integer before which the sequence of integers is to be returned. The range of integers end at `stop - 1`.
    - step (Optional): integer value which determines the increment between each integer in the sequence
- Iterate in the range from `1` to `10` using a `for` loop and increment our variable by adding the number

Workflow

The following image and the example below gives you an outlook at the workflow and syntax of a `for` loop.

Why loop control statements?

Sometimes you may need to change the way a normal loop is functioning. For that, Python has three keywords: `break`, `continue` and `pass`. Their descriptions are described below:

Control statement	Description
`break`	Terminates the loop and control shifts to the first statement outside it
`continue`	Causes the loop to skip the remainder of its body and shifts to the next item in the sequence
`pass`	Used when the statement is required but we don't want any output

You will get a more definitive idea about its syntax from the example below:

Example

```
alist = [1, 3, 5, 7, 9, 11, 13, 15]

blist = []

for i in alist:

    if i <= 10:

        j = i - 1
```

```
        blist.append(j)

    else:

        break

print(blist)
```

Here,

- We have a list named `alist` containing odd integers from 1 to 15 and another empty list `blist`
- Iterate over the list to check if the number is less than or equal to 10 and if yes, then subtract 1 and append to the second list; otherwise break off from the loop.

FILES INPUT AND OUTPUT

`print()` function

You already know that we use `print()` to display our results.

The actual syntax is:

```
print(*objects, sep=' ', end='\n', file=sys.stdout, flush=False)
```

Here,

- `objects` is/are the values to be displayed
- `sep` is the separator used between the values (defaults into a space character)
- After all the values are printed, `end` is printed (defaults into a new line)
- `file` is the object where the values are printed (default value is `sys.stdout` i.e. screen)

`input()` function

Now let's look at the `input()` statement.

The syntax for `input()` is: `input([prompt])`, where `prompt` is the string we want to display into the screen and is optional.

Need for `input()`

So far, the programs were static. All the values/variables were pretty much pre-defined or hardcoded To allow a user to take custom inputs, we have the `input()` function.

Regardless of the type of input we enter, the input is always taken in the form of a string. So, if you want to change its type, use type conversion.

Example of `input()`

Let's take an example where we first take custom input with `input()` and pass it as `'Robbie'` and store in the `name` variable.

```
>>> name=input()

Robbie

>>> print("Hey!" My name is " + name)

Hey! My name is Robbie
```

File Operations

In data manipulation and analysis, you will often come across reading and writing files. Python can handle files of different types including `txt`, `JSON`, `HTML`, `CSV` etc.

Let's look at the pipeline to handle file operations:

Step 1: Open the file

Lets say that you have a text file named `months.txt`. How to open this file? We use the `open()` function to open files and its syntax is given below:

```
open(file, mode='r', buffering=-1, encoding=None, errors=None, newline=None, closefd=True, opener=None)
```

For now, let's focus only on the `path` and `mode` arguments.

You can open the file passing the `path` argument to the `open()` function. Here, `file` is the file path of the file `months.txt`.

Now you may want to use the file plainly for reading but in some cases, you may want to modify it also. For this purpose, use the argument `mode` to chose how you want to treat the file object. Below are some of the mode options:

- `'r'` : Opens a file for reading only. The file pointer is placed at the beginning of the file. This is the default mode.
- `'rb'`: Opens a file for reading only in binary format. The file pointer is placed at the beginning of the file. This is the default mode.
- `'rb+'`: Opens a file for both reading and writing in binary format. The file pointer will be at the beginning of the file.
- `'r+'` : Opens a file for both reading and writing. The file pointer will be at the beginning of the file.
- `'w'` : Opens a file for writing only. Overwrites the file if the file exists. If the file does not exist, creates a new file for writing.
- `'wb'`: Opens a file for writing only in binary format. Overwrites the file if the file exists. If the file does not exist, creates a new file for writing.

- `'w+'`: Opens a file for both writing and reading. Overwrites the existing file if the file exists. If the file does not exist, it creates a new file for reading and writing.
- `'wb+'`: Opens a file for both writing and reading in binary format. Overwrites the existing file if the file exists. If the file does not exist, it creates a new file for reading and writing.
- `'a'`: Opens a file for appending. The file pointer is at the end of the file if the file exists. That is, the file is in the append mode. If the file does not exist, it creates a new file for writing.
- `'a+'` : Opens a file for both appending and reading. The file pointer is at the end of the file if the file exists. The file opens in the append mode. If the file does not exist, it creates a new file for reading and writing.
- `'ab'`: Opens a file for appending in binary format. The file pointer is at the end of the file if the file exists. That is, the file is in the append mode. If the file does not exist, it creates a new file for writing.
- `'ab+'`: Opens a file for both appending and reading in binary format. The file pointer is at the end of the file if the file exists. The file opens in the append mode. If the file does not exist, it creates a new file for reading and writing.

Step 2: Read the file

In this step, you read the contents of the file. It can be done in several ways:

- `<file>.read()`: It returns the entire content of the file as a single string.
- `<file>.readline()`: It returns the next line of the file, returning the text up to and including the next newline character. Simply put, this operation will read a file line-by-line.
- `<file>.readlines()`: It returns a list of the lines in the file, where each item of the list represents a single line.

Step 3: Write to a file

To write to a file use `<file>.write()`. The `write()` method does not add a newline character (`\n`) to the end of the string.

Step 4: Close the file

Although Python automatically closes a file when the object reference changes, it is considered a good practice to close the file. It keeps our data safe by restricting access to any other running programs.

It is done with the help of the `<file>.close()` method.

example:

In this task you will open a file, read the content inside it, write on the file and save it.

- Open the file in the read-only mode `('r')` with the `open()` function
- Read every line with the `.readlines()` method and use a `for` loop to iterate over it and print every line
- Close the file with `.close()` method
- Open the file in append at the end mode i.e. `('a+')`
- Next, use the `.write()` method to append the message `"Successfully written to the text file."`
- Now, close the file with `.close()` method

```
# filepath
filepath = text_file
```

```
# Code starts here

# open file in read mode
file = open(filepath, 'r')

# iterate over the file object
for line in file.readlines():
   print(line)

# close the file
file.close()

# open file in write mode
file = open(filepath, 'a+')

# write to the file
file.write('Successfully written to the file.')

# close the file
file.close()

# Code ends here
# Caution : do not delete this code
file = open(filepath,'r')
lines=[]
for line in file.readlines():
   lines.append(line)
file.close()
```

EXCEPTION HANDLING

What are Errors?

Introduction

Before we start discussing exceptions it is important to understand what errors are. In simple words, errors are mistakes in the code that Python doesn't like and will result in the abrupt termination of the program. Now errors are of two types:

- `Syntax Error`: As the name itself suggests they happen due to incorrect syntax of our code. Remember that syntax errors occur at compile time.

An example indicating such an error is shown in the image below where we forget to put `:` after initiating the `while` loop:

Output

```
while True print('Hello world')

File "<ipython-input-4-2b688bc740d7>", line 1
    while True print('Hello world')
                    ^
SyntaxError: invalid syntax
```

They also are known as parsing errors and are the most common kind of complaint you get while you are still learning Python. Also, they are easy to fix as Python will show you the line number where the error is, with an error message which will be self-explanatory.

Exceptions

An exception is a Python object that represents an error. If the normal flow of the program gets disrupted in spite of being syntactically correct, Python raises an exception; which if not handled properly, the program will terminate.

Remember that exceptions always occur at run-time. Python comes with various built-in exceptions as well as the possibility to create self-defined exceptions which we will be discussing in the upcoming topics.

```
An example of an exception is shown below where it shows us `IndexError` when we try to access an index which is not within the dimensions of the list `array`:
```

```
  array= [1,2,3,4]
   print(array[4])
```

Output

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
 <ipython-input-3-2543d2a0e9f4> in <module>()
      1
      2 array= [1,2,3,4]
----> 3 print(array[4])

 IndexError: list index out of range
```

Now let us understand how to raise an exception. In general, an exception instance can be raised with the `raise` statement. When an exception is raised, no further statements in the current block of code are executed unless the exception is handled (described in the next topic).

Let us look at an example of raising exceptions:

```
# custom input


num = int(input())




# raise exception if input is negative


if num < 0:


    raise Exception('{} is negative, please enter a positive number'.format(num))
```

```
# print input number if it is not negative

print('Your number is accepted!')
```

Its outputs when we take a negative number `-10` and a positive number `5` are:

With -10 as input

Output

```
---------------------------------------------------------------------------

Exception                                 Traceback (most recent call last)

<ipython-input-5-9029052661ca> in <module>()

      4 # raise exception if input is negative

      5 if num < 0:

----> 6     raise Exception('{} is negative, please enter a positive number'.format(num))

      7

      8 # print input number if it is not negative



Exception: -10 is negative, please enter a positive number
```

With 5 as input

Output

```
'Your number is accepted!'
```

In the above snippets:

- First, we are taking a custom input
- Check if the number is negative and raise an exception using `raise` statement asking the user to take a positive number
- If we enter a negative number
- -10
- $-10$for example; it will raise an exception and the program stops after `raise` statement

Exception Handling with Python

Why do you need exception handling?

Now that you are hopefully acquainted with the exceptions and the `raise` statement its high time to understand how to handle different types of exceptions. But first, why do you need to handle exceptions?

Exception handling provides a mechanism to decouple handling of errors or other exceptional circumstances from the typical control flow of your code. This allows more freedom to handle errors.

How does Python handle exceptions?

Python handles exceptions using the `try` and `except` blocks. Any code that we think might throw an error is placed inside the `try` block. If an exception is raised, control flow leaves this block immediately and goes to the `except` block which handles the corresponding exception.

Pseudocode for exception handling

```
try:
    Operational/Suspicious Code
except SomeException:
    Code to handle the exception
```

Example

Look at the example below to understand the syntax and flow of the `try-except` statement.

```
name = 'Hello World!'
try:
    char = name[15]
    print(char)
except IndexError:
    print('IndexError has been found!')
```

Output

```
'IndexError has been found!'
```

Here,

- A variable `name='Hello World!'` is initialized
- At first, the suspicious code inside the `try` block is executed. In the example above `name[15]` is the suspicious code
- If no exception occurs, then the code under `except` clause is skipped.
- Since index 15 doesn't exist in `name`, an exception will be raised and if the exception type matches exception name after `except` keyword, then the code in that `except` clause is executed. In our example exception is of type `IndexError`.

Multiple `except` statements

It is legal to have multiple except statements, each of which names different types of exceptions. If no exceptions are named in the except statement, it will catch all exceptions. For example:

```
# multiple except statements
try:
    print(45/den)
except ValueError:
    print('caught ValueError')
except ZeroDivisionError:
    print('caught ZeroDivisionError')
```

The above snippet has multiple `except` statements which catches `ValueError` and `ZeroDivisionError`.

else and finally Clauses

You can also use the `else` and `finally` clauses during exception handling. Let us see why and how you can use them.

`else` clause

The code inside `else` clause runs only when the exception doesn't occur i.e. if the `else` block is executed, then the `except` block is not, and vice versa. This block is optional.

Example using `else` block

```
# try-except-else block
try:
    print(45/12)
except ValueError:
    print('caught ValueError')
except ZeroDivisionError:
    print('caught ZeroDivisionError')
else:
    print('Successfully divided')
```

Output

```
3.75
'Successfully divided'
```

`finally` clause

The code inside `finally` clause always executes after the other blocks, even if there was an uncaught exception or a return statement in one of the other blocks. This block is optional.

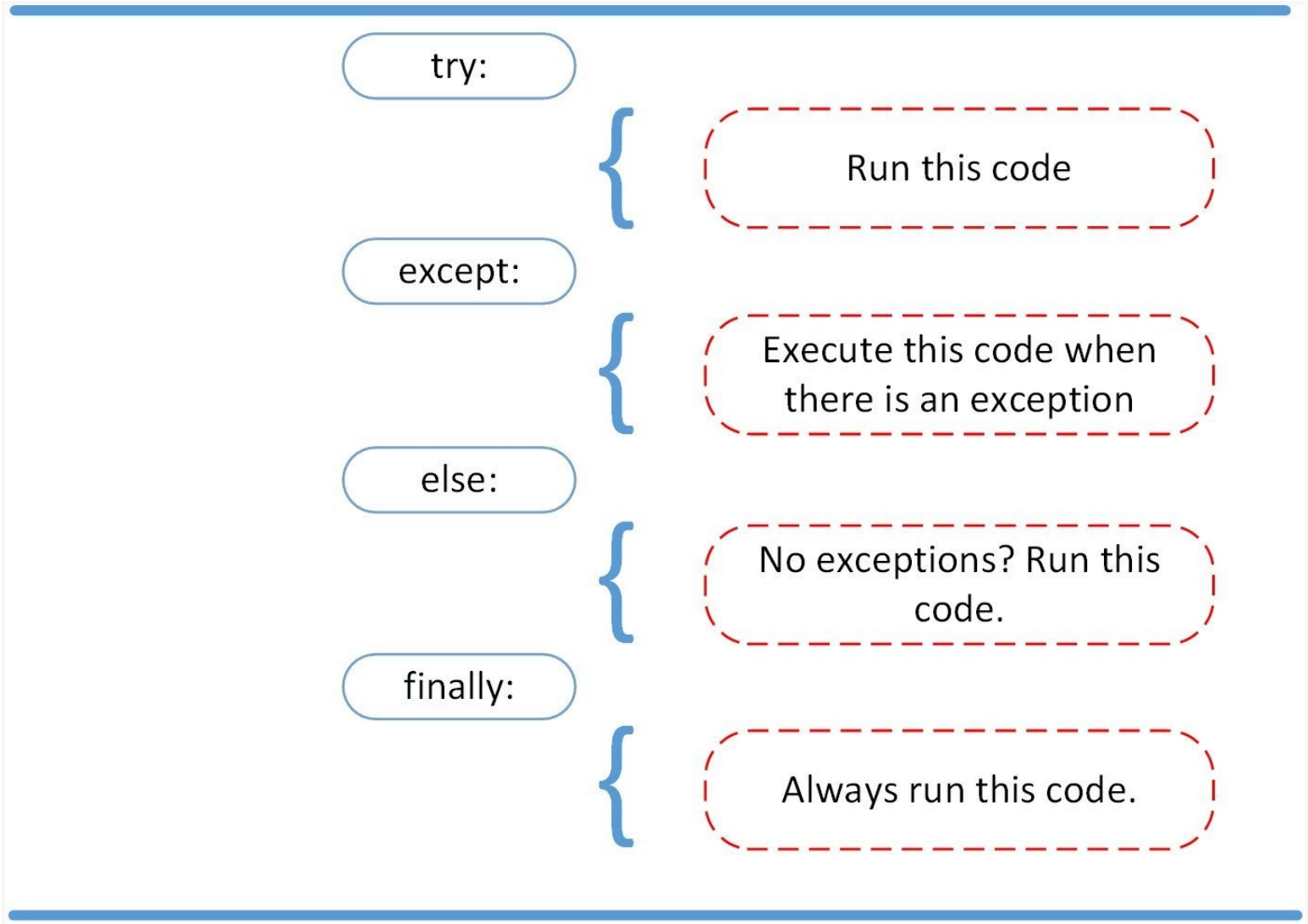
Example using `finally` block

```
try:
    print(45/0)
except ValueError:
    print('caught ValueError')
except ZeroDivisionError:
    print('caught ZeroDivisionError')
else:
    print('Successfully divided')
finally:
    print('Every block executed!')
```

Output

```
'caught ZeroDivisionError'
'Every block executed!'
```

Pipeline of try-except-else-finally blocks



Some Built-in Python Exceptions Below are some of the built-in python exceptions:

- Exception: The base class for all kinds of exceptions. All kind of exceptions are derived from this class
- ArithmeticError: Base class for the exception raised for any arithmetic errors.
- EOFError: Raised when `input()` function read End-of-File without reading any data.
- ZeroDivisionError: Raise when the second argument of a division or modulo operation is zero
- AssertionError: Raised when an `assert` statement fails
- FloatingPointError: Raised when a floating-point operation fails
- KeyError: Raised when a mapping (dictionary) key is not found in the set of existing keys
- KeyboardInterrupt: Raised when the user hits the interrupt key (normally Control-C or Delete). During execution, a check for interrupts is made regularly.

FUNCTIONS

What is a Function?

What is a function and why are they important?

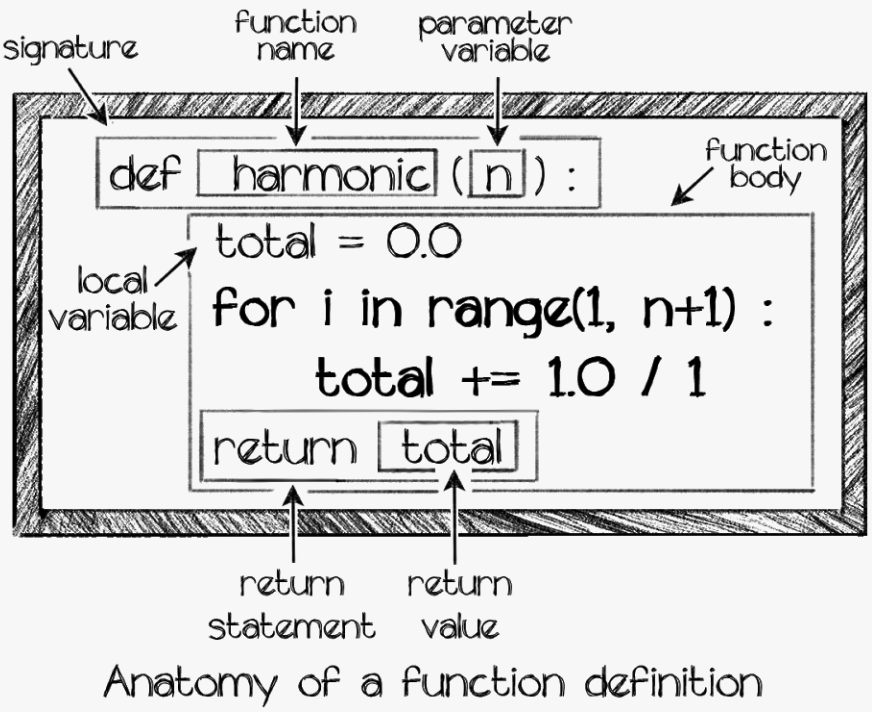
A function is a block of code that is meant to perform a single, relatable task. During problem-solving, it becomes necessary to break a problem into smaller tasks; functions do just that. It makes our code more readable and modular. It also avoids repetition and makes code reusable.

Python provides a lot of in-built functions like `len()`, `print()` etc. Like any other programming language, Python also provides support to create customized functions also called user-defined functions.

Let's discuss how to create a function and its syntax in the next topic.

Create your own functions

Let's understand how to create your own function with the help of an example.



Anatomy of a function definition

In the above function we have created the function named `harmonic` which calculates the harmonic mean of numbers from `1` to `n`. Let's understand its anatomy. Here,

- Keyword `def` marks the start of the function header.
- A function name to uniquely identify it. In our example, name is `harmonic`.
- Parameters (arguments) through which we pass values to a function. They are optional. `n` is our argument in our function.
- A colon
- (:)
- (:)to mark the end of function header.
- Optional documentation string (docstring) to describe what the function does.
- One or more valid python statements that make up the function body. The part labeled as the function body makes up this part.
- An optional `return` statement to return a value from the function. The return value is named `total` in the function `harmonic`.
- To call a function simply type the function name with the appropriate parameters.

For more information on functions go through the official Python3 documentation at https://docs.python.org/3/tutorial/controlflow.html#more-on-defining-functions.

Scoping

How long a variable exists, depends on where it is defined. We call the part of a program where a variable is accessible its scope, and the duration for which the variable exists its lifetime. For example, parameters and variables defined inside a function is/are not visible from outside. Hence, they have a local scope. Variables declared outside a function have global scopes and they can be used throughout the program.

To use a local variable as a global one, use the global keyword before the variable name. In this way, Python understands that you are using the reference of the global variable.

Lets understand this with an example below:

```
def f():
    # Local scope
    s = "Me too."
    print(s)

# Global scope
s = "It is great."

print(f())
print(s)
```

Output

```
"Me too."
"It is great."
```

In the above example, we have the function `f()` which returns a variable `s`. The local variable `s` is defined as a string `" Me too."` The variable is again defined in the global scope as `"It is great."`

When we print out the function `f()` the output will be that of the local variable `s`. While printing out the global variable `s` will give the value of the global variable `s`.

Lambda Functions

Earlier you saw that a function was created with the `def` keyword followed by its name. However, Python also supports the creation of anonymous functions i.e. functions without a name also called *lambda* functions.

Why lambda functions?

They are usually small and can have any number of arguments just like a normal function. In the image below you can see how a lambda function is created:

Example

```
sumLambda = lambda x, y : x + y

result = sumLambda(4, 5)

print(result)
```

In the above example:

- `sumLambda` is a function that takes in two numbers and returns the sum.
- Arguments lie in between the word lambda and the colon (:).
- The function body lies to the right of the `:`
- There is no `return` statement, the function body is sufficient.

  Example:

  Compute square of a number using lambda functions

In this task you will be going to define a function that takes in a value and returns its square and use it in conjunction with a list comprehension to calculate the squares of the first `10` natural numbers

- Make a lambda function `square` that takes in a number and returns its square and store it in a variable
- Store the first `10` natural numbers in a list `nums`
- Initialize an empty list `square_nums` where you will be storing the square of the first `10` natural numbers
- Using `for` loop apply the lambda function on each element of the list `nums`, then append it to `square_nums`
- Print `square_nums`

  soln:

  ```
  # lambda function to calculate square

  square = lambda x : x*x

  # natural numbers list

  num = []

  for i in range(1,11):

     num.append(i)

  # empty list
  ```

```
square_nums = []

for i in num:

    sq = square(i)

    square_nums.append(sq)

print(square_nums)

# loop through every element in list

# display new list
```

Working with Higher-Order Functions

Higher order functions are those functions which can accept other functions as arguments.

Python provides support for them too. Some of the examples of higher order functions are `map()`, `reduce()`, `filter()` etc.

Let us understand what is a higher order function and how a higher order function works with the help of the following snippet:

```
# higher order function

def combine_values(func, values):

    current = values[0]

    for i in range(1, len(values)):

        current = func(current, values[i])

    return current


# addition

def add(x, y):

    return x + y


# multiplication
```

```
def mul(x, y):
    return x*y


# adds all the values in the list
print(combine_values(add, [1,2,3,4] ))


# multiplies all the elements in the list
print(combine_values(mul, [1,2,3,4] ))
```

Output

```
10
24
```

Here,

- `combine_values()` is a function that takes in a function and an iterable as arguments; hence it is a higher-order function.
- `add()` and `mul()` are two functions that return the addition and multiplication of two numbers.
- When we pass `add()` function as an argument to `combine_values()`, it calculates the summation of the numbers inside the `values` argument which is a list `[1,2,3,4]`.
- Similarly passing `mul()` as an argument results in the multiplication of the numbers inside the list `[1,2,3,4]`.

map, reduce and filter

Python has many popular higher-order functions like `map()`, `reduce()`, `filter()`. Let's look at one example from each of them:

- `map(function, sequence)` calls `function(item)` for each of the sequence's items and returns a list of the returned values.

```
# function to square numbers
def square(x):
```

```
    return x**2

# map function

mapped = list(map(square, [1,2,3,4]))

print(mapped)
```

Output

```
[1, 4, 9, 16]
```

- `filter(function, sequence)` returns a sequence consisting of those items from the sequence for which `function(item)` is true.

```
# function to check if number is positive

def positive(x):

    return x > 0


# filter function

filtered = list(filter(positive, [-3,-2,-1,1,2,3,]))

print(filtered)
```

Output

```
[1, 2, 3]
```

- `reduce(function, sequence)` takes an iterable of input data and consumes it to come up with a single value.

```
# import reduce

from functools import reduce
```

```
# function to divide two numbers

def add(x,y):

    return x + y


# reduce function

reduced = reduce(add, ['a', 'b', 'c'])

print(reduced)
```

Output

```
'abc'
```

You might wonder that the map and filter function are both the same. But they work a little bit differently.

Comprehensions

By now, you are already familiar with existing data structures in Python like lists, tuples, dictionaries, etc. All of them need a significant amount of time and code length to create until now.

But they can be created in a much shorter format which is more time-efficient as well as readable. They are called comprehensions.

Depending on the type of brackets you use, they may be called:

- [ ] as list comprehension
- ( ) as generator expressions
- { } for dictionary and set comprehensions

The syntax for comprehensions remains the same irrespective of the type of bracket we chose and is given below

```
[expression(iterator) for iterator in iterable if condition]
```

You can also use nested for loops with multiple conditions in order to create comprehension.

Let's understand with a simple example.

```
def even_addition(values):
    evens = [i for i in values if not i%2]
    return sum(evens)

print(even_addition([1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]))
```

Output

```
42
```

In the above example,

- `even_addition()` is a function that calculates the sum of even elements in the iterable `values`.
- A list comprehension is defined in the second line upon which we use the `sum()` function.

Example:

In this task you will make a list comprehension to return a list of numbers which are not present in another list of numbers.

- Define a list of numbers `alist` from `1` to `50` using a list comprehension.
- Define another list `blist` and store its values as `[2,3,5,7,11,13,17,19,23,29,31,37,41,43,47]`.
- Make a list `final` using list comprehension which would contain elemnts which are in `alist` but not in `blist`.
- Print `final`

```
# initialize both lists

alist = []

for i in range(1,51):

   alist.append(i)

blist = [2,3,5,7,11,13,17,19,23,29,31,37,41,43,47]

final = [i for i in alist if i not in blist]

print(final)

# final list
```