Analytics using Python

Learning outcomes

1. You will learn Python , a useful language

2. Use programming for problem solving

Great Lakes Institute of Management

A guide to learn python for analytics

great lakes faculty

**Chapter 1. Basics of Python**

**Table of contents**

Contents

[**1.** **Introduction** 4](#_Toc4857954)

[**1.1** **What is Python?** 4](#_Toc4857955)

[**1.2** **What is a programming language?** 4](#_Toc4857956)

[**1.3** **Common basic instructions in a language** 5](#_Toc4857957)

[**1.4** **Debugging** 5](#_Toc4857958)

[**1.5** **Type of errors** 5](#_Toc4857959)

[**1.6** **Frustrating debugging** 6](#_Toc4857960)

[**1.7** **Popular debugger tools** 6](#_Toc4857961)

[**1.8** **Coding standards** 6](#_Toc4857962)

[**2.** **Constants, Variables, Expressions** 7](#_Toc4857963)

[**2.1** **Python as a calculator** 7](#_Toc4857965)

[**2.2** **Operators and operands** 7](#_Toc4857966)

[**2.3** **Types of values** 7](#_Toc4857967)

[**2.4** **Order of operations or rule of precedence (PEDMAS)** 9](#_Toc4857968)

[**2.5** **Constants and variables** 10](#_Toc4857969)

[**2.5.1.** **Variable** 10](#_Toc4857977)

[**2.5.2.** **Python identifiers** 10](#_Toc4857978)

[**2.5.3.** **Naming convention for Python identifiers** 10](#_Toc4857979)

[**2.5.4.** **Reserved words or Key words** 11](#_Toc4857980)

[**2.5.5.** **Constant** 12](#_Toc4857981)

[**2.5.6.** **Literal** 12](#_Toc4857982)

[**2.5.7.** **Type of literals** 12](#_Toc4857983)

[**2.5.8.** **Numerical types supported by Python** 13](#_Toc4857984)

[**2.6** **Statements** 13](#_Toc4857985)

[**2.7** **Expressions** 13](#_Toc4857986)

[**2.7.1.** **Operations on Strings** 14](#_Toc4857989)

[**2.7.2.** **Composition** 14](#_Toc4857990)

[**2.8** **Types of operators** 14](#_Toc4857991)

[2.8.1. **Arithmetic Operators** 15](#_Toc4858003)

[**2.8.2.** **Relational Operators** 15](#_Toc4858004)

[**2.8.3.** **Assignment operators** 16](#_Toc4858005)

[**2.8.4.** **Comparison operators** 16](#_Toc4858006)

[**2.8.5.** **Logical operators** 17](#_Toc4858007)

[**2.8.6.** **Membership operators** 17](#_Toc4858008)

[**2.8.7.** **Identity operators** 18](#_Toc4858009)

[**3.** **Comments** 18](#_Toc4858010)

[**4.** **Built-in Function call** 19](#_Toc4858011)

[**4.1** **Number Type Conversion** 19](#_Toc4858017)

[**4.2** **STRINGS** 19](#_Toc4858018)

[**5.** **Conditionals and iterations** 25](#_Toc4858019)

[**5.1** **Conditionals** 25](#_Toc4858021)

[**5.1.1** **Chained conditional** 25](#_Toc4858028)

[**5.2** **Nested conditional** 26](#_Toc4858029)

[**5.3** **Iterations or Loop** 26](#_Toc4858030)

[**6.** **Standard Modules** 29](#_Toc4858031)

[**7.** **Date Time Operations** 31](#_Toc4858032)

[**7.1** **Create a date object** 31](#_Toc4858035)

[**8.** **User defined functions** 34](#_Toc4858036)

[**8.1** **Function** 34](#_Toc4858045)

[**8.2** **Statements** 34](#_Toc4858046)

[**8.3** **ANONYMOUS FUNCTION, LAMBDA** 36](#_Toc4858047)

[**8.4** **Scope of variables** 36](#_Toc4858048)

[**8.4.1.** **Local variable** 36](#_Toc4858062)

[**8.4.2.** **Global variable** 36](#_Toc4858063)

[**8.5** **Recursion** 38](#_Toc4858064)

[**8.6** **Boolean function** 39](#_Toc4858065)

[**9.** **Random numbers** 39](#_Toc4858066)

[**10.** **Error handling in Python** 41](#_Toc4858067)

[**11.** **Regular expression** 43](#_Toc4858068)

[**12.** **File handling** 47](#_Toc4858069)

[**A)** **Text files** 47](#_Toc4858083)

# **Introduction**

Python was developed a decade ago by Guido Van Rossum.

Python was inspired by ABC, a teaching language in 1980s. ABC is an interactive programming language and environment for personal computing, originally intended as a good replacement for BASIC. It was first designed by first doing a task analysis of the programming task. It was developed by Leo Geurts, Lambert Meertens and Steven Pemberton of Centrum Wiskunde & Informatica (CWI), Netherlands in 1980s.

This language was influenced by SETL & ALGOL 68 and influenced the language, Python.

<https://homepages.cwi.nl/~steven/abc/>

<https://en.wikipedia.org/wiki/ABC_(programming_language>)

## **What is Python?**

Python is a high-level programming language such as C, C++, Java etc. It is easier to code, read & comprehend in a high-level language than a low-level language. High-level languages are portable, that is they can run on any operating system with few or no modifications unlike low-level language.

Python is considered an interpreted language. Python programs are executed by an interpreter either in command-line mode or script mode by translating it one line at a time.

## **What is a programming language?**

A **program** is a sequence of instructions that specifies how to perform computing task such as inverting a matrix or search for an email address of a customer from customer profile description field.

Programming languages are *formal languages* designed for computing as opposed to *natural languages* that people speak and evolved naturally.

## **Common basic instructions in a language**

1) Input: Get data from a file or other device including key board

2) Output: Display the results of a program on the screen or send the same to a device such as a file

3) Processing: Perform basic operations including mathematical, or textual

4) Conditional execution: Before executing a sequence of programming statements check for appropriate conditions.

5) Repetition: Perform certain task repeatedly such as reading all records from a file and perform computations.

## **Debugging**

*Alexander Pope said* to err is human, to forgive divine*.* *George Bernard Shaw said,* A life spent making mistakes is better than a life spent doing nothing. Hence, it is natural for programmers, who are humans to commit errors, but we cannot expect the computers to be lenient to forgive.

Programming errors are called *bugs*. The process of tracking and rectify them is called *debugging*.

## **Type of errors**

1. Syntax error: These occur when the rules of the structure of the program is violated

2. Run time errors: These occur when the program is executed and something exceptional event has occurred such as search for a non-existent file

3. Semantic errors: These occur when the program does not perform as intended similar to the case *Operation Successful but patient is dead*. Semantic is the meaning of a program.

## **Frustrating debugging**

*Debugging is one of the most intellectually challenging and important part of programming. In order to debug, first you must have an idea what went wrong before modify the program to rectify the mistakes.*

*It is like a detective work. You need to recognize the clues that you confront. You have to infer the processes and events that led to the results you see.*

## **Popular debugger tools**

1. ***PixieDebugger for Jupyter Notebooks***

[*https://medium.com/ibm-watson-data-lab/the-visual-python-debugger-for-jupyter-notebooks-youve-always-wanted-761713babc62*](https://medium.com/ibm-watson-data-lab/the-visual-python-debugger-for-jupyter-notebooks-youve-always-wanted-761713babc62)

1. ***Spyder***[*https://docs.spyder-ide.org/debugging.html*](https://docs.spyder-ide.org/debugging.html)

## **Coding standards**

*A Program is read much more often than it is written. - Guido Van Rossum.*

*Guidelines provided by the coding standards for Python such as PEP 8 are intended to improve the readability of code and make it consistent across modules, projects.*

*It is important to ensure consistency within a project, one module, across modules. You need to follow the guidelines set by your organization.*

*For more details, refer to* [*https://www.python.org/dev/peps/pep-0008/*](https://www.python.org/dev/peps/pep-0008/)

# **Constants, Variables, Expressions**



## **Python as a calculator**

*Python interpreter acts as a calculator and when you type an expression, it is evaluated, and the output value is displayed.*

## **Operators and operands**

***Operators*** *are special symbols that represent computations like addition and multiplication. The values the operator uses are called operands.*

*The operators + (for addition) , - (for subtraction), \* (for multiplication) and / (for division), \*\* (to raise to the power), // (for floor division) and % (for getting remainder of the division) are used as in most of the other languages such as C. For grouping, use parenthesis().*

## **Types of values**

*Value is an element like letter or number that a program manipulates.*

*Python has four distinct numeric types such as integer, long integers, floating point numbers and complex numbers. Booleans (such as 0 or 1) are subset of integers.*

*Integers are implemented using long in C resulting in at least 32 bits of precision.*

*Long integers have unlimited precision.*

*Floating point numbers are implemented using double in C.*

*An integer (int) is a whole number (positive, negative or zero) such as 6, 7, 72, 88*

*A decimal integer contains one or more digits (0, 1, 2, 3, 4, 5, 6, 7, 8, 9).*

*Underscores (\_) may be used to improve readability. Example: 50\_000 for 50000*

*Leading zeros is not allowed in a decimal integer*

*A real number represents a value that represents a quantity that can have fractions in decimal forms.*

*For example. 4.55, 1.34 are real numbers. Float numbers are represented as floats in computer science.*

In [2]: 1\_000 + 5\_00\_000  
# Add two numbers 1,000 and 500,000. Use (\_) instead of comma(,) in the number  
Out[2]: 501000

In [3]: 50 -5  
# Subtract 5 from 50  
Out[3]: 45

In [4]: 50 / 5  
# Divide 50 by 5  
Out[4]: 10.0

*Here we notice that the output is given as a float instead of an integer. Is there any way of checking if it is an integer or float?*

*Hint: Use a Python built-in function such as type() command or* ***isinstance(x, int)\*\****

In [5]: type(50/5)  
Out[5]: float

In [6]:isinstance(50/5, float)  
Out[6]: True

*How do you get an integer when diving two numbers?*

In [7]: type(53//5)  
# Use floor division which gives the value not greater   
# than the result of the operation  
Out[7]: int

In [8]: 53 // 5  
Out[8]: 10

*You may want to round off to nearest two decimal places.*

In [9]: round(53/5,2)  
Out[9]: 10.6

*How do you get the remainder of a division?*

In [10]: 53 % 5  
Out[10]: 3

In [11]: (50 - 40) \*\* 2  
 # Subtract 40 from 50 and then find its square  
Out[11]: 100

In [12]: x = round(55 / 3, 2)

print(x, type(x))  
 Out[12]: 18.33 <class 'float'>

## **Order of operations or rule of precedence (PEDMAS)**

*PEDMAS is an acronym for* ***P****arenthesis (),* ***E****xponentiation \*\*,* ***D****ivision,* ***M****ultiplication,* ***A****ddition and* ***S****ubtraction. So the highest precedence is given to the parenthesis and the least precedence is given to subtraction."*

*Let us do the following calculation:*

1. Subtract 50 from 500

2. Multiply the result with 2

3. Add 10 to the result

4. Divide the result by 5

5. Raise the result to the power 3

***Using parenthesis, let us do the calculation***

In [13]: ((((5000 - 50) \* 2 ) + 10 ) / 5 ) \*\* 3  
Out[13]: 7785938168.0

***Without using parenthesis, let us do the calculation***

In [14]: 5000 – 50 \* 2 + 10 / 5 \*\* 3  
Out[14]: 4900.08

***Do you realize the importance of knowing the order of precedence and use of parenthesis?***

## **Constants and variables**



### **Variable**

*A variable is a named location identified by a unique name called identifier and used to store data in the memory of the computer.*

*In python, we do not need to declare variable and its type in advance. The variable declaration happens automatically when we assign a value to a variable using the assignment operator, #. For example, a = 5 declares a variable a and assigns a value 5. You can change the value of a variable any time.*

### **Python identifiers**

*A python identifier is a name used to identify an object such as variable, class, function, module or other object.*

*An identifier starts with a letter A to Z or a to z or an underscore (\_) followed by zero or more letters, one or two underscores and digits (0 to 9).*

*Python does not allow punctuation marks such as $, %, @ within identifiers. Python is case sensitive.*

*Identifiers are unlimited in length but readability counts; hence limit to 79 characters to conform to PEP-8 standards.*

### **Naming convention for Python identifiers**

Except for class names, all identifiers start with a lowercase letter.

A variable with one underscore as the first character indicates that the identifier is private.

A variable with two underscores as the first two characters indicates that the identifier is strongly private. This triggers name mangling when used in a class context as enforced by the Python interpreter to avoid name conflicts.

Language defined special name are written with two trailing underscores*.*

### **Reserved words or Key words**

|  |  |  |
| --- | --- | --- |
| and | exec | not |
| assert | finally | or |
| break | for | pass |
| class | from | print |
| continue | global | raise |
| def | if | return |
| del | import | try |
| elif | in | while |
| else | is | with |
| except | lambda | yield |

The following are Python key words which are reserved and cannot be used as constant or variable or any other identifier name. Note that all these contain lowercase letters only.

### **Constant**

A constant is a type of variable whose value cannot be changed. By convention, constants are written in all capital letters and underscore to separate words.

For example, PI = 3.14159

pi represented as PI above, is a name given to the ratio of the circumference of a circle to the diameter.

E = 2.71828

e represented as E above is the **Euler's number** after Leonhard Euler (pronounced "Oiler").

The number e is a famous irrational number, and is one of the most important numbers in mathematics.

The first few digits are: 2.7182818284590452 and there are many more decimal places.

e is the base of the Natural Logarithms (invented by John Napier).

e is found in many interesting areas. Refer <https://www.mathsisfun.com/numbers/e-eulers-number.html> for more details.

### **Literal**

Literal is a raw data assigned to a variable.

### **Type of literals**

* Numeric literal such as a = 100
* String literal such as string = 'This is a string'
* Boolean literal such as x = True
* Special literal such as None

### **Numerical types supported by Python**

1) **int** (signed integers) b Known as integers or ints; positive or negative whole numbers with no decimal point.

2) **long** (long integers ) b Aka longs, they are integers of unlimited size, written as integers and followed by an uppercase or lowercase L.

3) **float** (floating point real values) b Aka floats, they represent real numbers and are written with a decimal point dividing the integer and fractional parts.

4) **complex** (complex numbers) b They are of the form a + bJ, where a and b are floats and J (or j) represents the square root of -1 (which is an imaginary number). The real part of the number is a, and the imaginary part is b.

## **Statements**

A statement is an instruction that the Python interpreter can execute. A script contains a sequence of statements.

Examples: assignment and print

The assignment statement only assigns values to a variable and does not produce any output like print.

In [15]: x = 1000  
 # Assignment operator produces no output

print(x)  
# produces a result  
Out [15]:1000

## **Expressions**

An expression is a combination of values, variables and operators.

In [16]: msg = "Hello World"

msg

Out[16]: 'Hello World'

*When the Python interpreter displays a value of the expression, msg, it uses the same format you would use to enter a value (within single quotes).*

In [17]: print(msg)  
Out [17]: Hello World  
*When you use a print statement, Python displays the contents of the string without the quotation marks.*



### **Operations on Strings**

Operator + concatenates the two operands by linking them end to end. For example, beverage = 'Coffee'; food = 'bagel; beverage + ' with ' + food displays 'Coffee with bagel'

In [18]: beverage = 'Coffee ' ; food = 'bagel '  
 beverage + ' with ' + food  
 Out[18]: 'Coffee with bagel '

Operator \* repeats the first operand the number of times specified in the second operand.

In [19]: beverage \* 2  
Out[19]: 'Coffee Coffee '

### **Composition**

A very useful feature of the programming language is the ability of the program to take elements of a program such as variables, expressions and statements and combine (or compose) them.

In [20]: number\_of\_employees = 500 + 400 + 100

print('\nNumber of employees is %d'%number\_of\_employees)  
 ## \n indicates skip one line; %d print decimal int  
 print('\nNumber of employees is', number\_of\_employees)

Out[20]: Number of employees is 1000

Number of employees is 1000

## **Types of operators**

We have the following types of operators:

**1) Arithmetic Operators**

**2) Relational Operators**

**3) Assignment Operators**

**4) Logical Operators**

**5) Membership Operators**

**6) Identity Operators**



### **Arithmetic Operators**

In[21]: ### Arithmetic operators  
In [22]: print(345 + 1223)  
 # '+' to perform addition  
 print(346 – 234 )  
 # '-' To perform subtraction  
 print(344 / 56 )  
 # '/' To perform division  
 print(34 \* 2001)  
 # To perform multiplication  
 print(56 \*\* 2 )  
 # '\*\*' to raise to power  
 print(57 % 2)  
 # '%' to return remainder  
 print(57 // 2 )  
 # '//' to return floor division  
Out [22]: 1568  
 112  
 6.142857142857143  
 68034  
 3136  
 1  
 28

### **Relational Operators**

In [23]: ### Relational operators  
 print("\nRelational operators")  
 print('10 < 20', 10 < 20)  
 # < less than  
 print('10 > 20' , 10 > 20)  
 # > greater than  
 print('10 <= 20', 10 <= 20 )  
 # <= less than or equal to  
 print('10 >= 20', 10 >= 20)  
 # >= Greater than or equal to  
 print('10 == 20', 10 == 20)  
 # == Equal to  
 print('10 != 20', 10 != 20)  
 # != Not equal to  
Out [23]: Relational operators  
 10 < 20 True  
 10 > 20 False  
 10 <= 20 True  
 10 >= 20 False  
 10 == 20 False  
 10 != 20 True

### **Assignment operators**

Assignment operator assigns the value of its right operand to its left operand.

In [24]: a = 1637  
 # = assignment  
 print("Result = operator:", a )  
 a += 5  
 # += Add and assign  
 print("Result += operator:", a)  
 a -= 5  
 # -= Subtract and assign  
 print("Result -= operator:", a)  
 a \*= 5  
 # \*= Multiply and assign  
 print("Result\*-= operator:", a)  
 a /= 5  
 # /= Divide and assign  
 print("Result/= operator:", a)  
 a %= 5  
 # % = Modulus and assign  
 print("Result%= operator:", a)  
 a \*\*= 5  
 # \*\*= Raise to the power and assign  
 print("Result\*\*= operator:", a)  
 a //= 5  
 # Perform floor division  
 print("Result //= operator:", a)

Out [24]: Result = operator: 1637  
 Result += operator: 1642  
 Result -= operator: 1637  
 Result\*-= operator: 8185  
 Result/= operator: 1637.0  
 Result%= operator: 2.0  
 Result\*\*= operator: 32.0  
 Result //= operator: 6.0

### **Comparison operators**

x == y # x is not equal to y

x > y # x is greater than y

x >= y # x is greater than or equal to y

x <= y # x is less than y or equal to y

x < y # x is less than y

x != y # x is not equal to y

### **Logical operators**

**Boolean expression**

A Boolean expression is an expression that is either true or false. By using comparison operators such as == or > or >= or <= or < or !=, one can write the Boolean expression.

10 > 6 is True and 12 < 10 is False. True and False are special values built in Python.

Python supports following 3 logical operators:

1. and ==> When both conditions are true output will be true
2. or ==> When any one condition is true, output will be true
3. not ==> Compliment (reverse) the condition

In [25]: x = True  
 y = False  
 print("x and y ==>> ", x and y )  
 print("x or y ==>> ",x and y)  
 print("not y ==>> ", not y)

Out [25]: x and y ==>> False  
 x or y ==>> False  
 not y ==>> True

### **Membership operators**

in and not in are the membership operators in Python.

It used to test whether a value or variable is found in a sequence(List ,tuples ,dictionary etc.).

In a dictionary we can only test for presence of key, not the value.

In [26]: print('H in Hello World is ', 'H' in 'Hello World')  
 print('H not in Hello World is ' , 'H' not in 'Hello World')  
 print('h in Hello World is ', 'h' in 'Hello World')  
H in Hello World is True  
H not in Hello World is False  
h in Hello World is False

### **Identity operators**

**is** and **is not** are the identity operators in Python.

It used to check if two values (or variables) are located on the same part of the memory.

In [27]: S1 = "Orange"  
 S2 = "Apple"  
 S3 = "Orange"  
 print("S1 is S3 is ", S1 is S3)  
 print("S1 is not S3 is ", S1 is not S3)  
 print("S1 is S2 is ", S1 is S2)  
 print("S1 is not S2 is ",S1 is not S2)  
  
Out [27]: S1 is S3 is True  
 S1 is not S3 is False  
 S1 is S2 is False  
 S1 is not S2 is True

# **Comments**

**Please remember we write code for others. Comments improve understanding of the code and its logic.**

In [28]: # is a single line comment

**Multi-lines comments**

In [29]: """\  
 This is a \  
 two line comment\  
 """  
Out[29]: 'This is a two line comment'

# **Built-in Function call**

We have seen function, type(x) with x as the value or variable as its argument mentioned within parenthesis. This function returns the type of the variable as its return value and this can be assigned to another variable.

The function, id() takes a value or variable and returns an integer that acts as a unique identifier for the value which is stored in the memory of the computer.



## **Number Type Conversion**

There may be times when you need to coerce a number explicitly from one type to another to satisfy the requirements of an operator or function parameter.

* Type int(x) to convert x to a plain integer.
* Type float(x) to convert x to a floating-point number.
* Type complex(x) to convert x to a complex number with real part x and imaginary part zero.
* Type complex(x, y) to convert x and y to a complex number with real part x and imaginary part y. x and y are numeric expressions

In [30]: x = 5; y = 2  
 print(type(x))  
 print('type(float(x))', type(float(x)), 'float(x)', float(x))  
 print('type(complex(x))', type(complex(x)), 'complex(x)', complex(x))  
 print('type(complex(x,y))', type(complex(x, y)), 'complex(x,y)', complex(x, y))

Out[30]: <class 'int'>  
 type(float(x)) <class 'float'> float(x) 5.0  
 type(complex(x)) <class 'complex'> complex(x) (5+0j)  
 type(complex(x,y)) <class 'complex'> complex(x,y) (5+2j)

## **STRINGS**

Strings are different from other data types such as int and float because they are made up of smaller pieces, characters. This data type is known as compound data types. Depending on the requirement, we can treat it as a single element or access its parts.

The square bracket operator, [ ] selects a single character from a string. For example, expression s1[0] selects 0th character from the string s1. This is known as slicing. String can also be sliced from both the direction that is “left” and “right”

<String\_name>[Start Index**:**End Index]

<String\_name>[**:**End Index]

<String\_name>[Start Index**:**]

In [3]: s1 = "Hello"

s1[0] # Extract one character at first position

Out[3]: 'H'

In [4]: s1[1:4] # Extract characters starting at second position up to fourth position

Out[4]: 'ell'

In [5]: s1[:4] # Extract characters starting at first position up to fourth position

Out[5]: 'Hell'

In [6]: s1[3:] # Extract characters starting from fourth position till end

Out[7]: 'lo'

**String membership operators are two types:**

* **in I**t returns **True**, if a character or substring is present in the specified string. otherwise false.
* **not in I**t returns **True**, if a character or substring does not present in the specified string. otherwise false.

In [7]: gl1 = "Great Lakes"

gl1s = "Great"

gl1s in gl1

Out[7]: True

Format strings contains curly braces {} as placeholders or replacement fields which gets replaced.

1. **Default – method**

In [8]: formatted\_string\_default = "{}, {}".format('Hello','World')

print(formatted\_string\_default)

Out[8]: Hello, World

1. **Positional argument – method**

In [9]: formatted\_string\_position = "{0}, {1}".format('Hello','World')

print(formatted\_string\_position)

Out[9]: Hello, World

1. **Positional argument - method**

In [10]: formatted\_string\_key = "{h}, {w}".format(h = 'Hello', w= 'World')

print(formatted\_string\_key)

Out[10]: Hello, World

Strings in Python have a built-in operation that can be accessed with the % operator enabling you to do simple positional formatting easily.

In [11]: print("%s" %"Hello World")

Out[11]: Hello World

String operators such as Concatenation Operator(+) and Replication Operator (\*)

In [12]: s1 = 'Hello' ; s2 = ' World'  
 # Concatenation  
 print('s1 + s2', s1 + s2 )  
 s3 = '..'  
 print('s3 \* 5', s3 \* 5)  
 # Repetition -   
 print('s1[0] is', s1[0])  
  
Out [12]: s1 + s2 Hello World  
 s3 \* 5 ..........  
  
In [12]: s = "Hello World "  
 s[6:-1]  
 # # Extract from sixth element up to end of the string S1  
Out[34]: 'World'

String has built-in functions such as capitalize() to convert the initial letter to capital case.

A few useful functions:

* islower() to check the string is in lower case or not
* upper() to convert to the given string to upper case letters
* len() to get the number of characters
* find() to find the position of the character string in the main string.

In [13]: main\_string = 'Great Lakes'

print(main\_string.find('Lakes'))

print(main\_string.index('Lakes'))

Out [13]: 6

6

In [35]: \_cust\_name1 = 'Peter'  
 \_cust\_name2 = 'peter'  
In [36]: \_cust\_name1.lower() == \_cust\_name2.lower()  
Out[36] : True

In [37]: width = 10  
 fillchar = '.'  
 s1.center(width, fillchar)  
 # Returns a space-padded string with the original string centered to a total of width columns.  
Out[37]: '..Hello...'

In [38]: s1 = 'hello'  
 print('s1.capitalize returns >>> ', s1.capitalize())  
 # Capitalizes first letter of string

Out [38]: s1.capitalize returns >>> Hello

In [39]: print(' Hello'.endswith('o'))  
 # Determines if string or a substring of string   
 # (if starting index beginning and ending index end are given) ends with suffix;  
 # returns true if so and false otherwise.  
Out[39]: True

In [40]: s1 = "Hello"  
 print(s1.isalnum())  
 # Returns true if string has at least 1 character and  
 # all characters are alphanumeric and false otherwise

print(s1.isalpha())  
 # Returns true if string has at least 1 character  
 # and all characters are alphabetic and false otherwise.  
Out [40]: True  
 True

In [41]: n = '20'  
 n.isdigit()  
 # Returns true if string contains only digits (0 to 9) and false otherwise.  
Out[41]: True

In [42]: c = s1.upper()  
 c1 = "33000"  
 print(c.islower())  
 # Returns true if string has at least 1 cased character and all cased   
 # characters are in lowercase and false otherwise.  
Out[42]: False

In [43]: print(c1.isnumeric())  
 # Returns true if a unicode string contains only numeric characters  
 # and false otherwise.  
Out [43]: True

In [44]: print(c.isspace())  
 # returns true if string contains only whitespace characters  
 # and false otherwise.  
Out [44]: False

In [45]: print(c.istitle())  
 # Returns true if string is properly "titlecased" and false otherwise.  
Out [45]: False

In [46]: print(c.isupper())  
 # Returns true if string has at least one cased character   
 # and all cased characters are in uppercase and false otherwise.  
Out [46]: True

In [47]: print(len(c))  
 # Returns the length of the string  
Out [47]: 5

In [48]: print(c.lower())  
 # Converts all uppercase letters in string to lowercase.  
Out [48]: hello

In [49]: c = " " + c  
 print(c.lstrip())  
 # Removes all leading whitespace in string.  
Out [49]: HELLO

In [50]: print(' sss '.rstrip())  
 # Removes all trailing whitespace of string.  
Out [50]: sss

In [51]: print(' Hello'.startswith('He'))  
 print('Hello'.startswith('He'))

Out [51]: False  
True

In [52]: print(' sss '.strip())  
 # Performs both lstrip() and rstrip() on string.  
Out [52]: sss

In [53]: print('hello world'.title())  
 # Returns "titlecased" version of string  
 Hello World

In [54]: s = "This is a new country"  
 s.split(" ")  
 # Splits string according to delimiter str (space if not provided) and returns   
 # list of substrings; split into at most num substrings if given.  
Out[54]: ['This', 'is', 'a', 'new', 'country']

In [55]: s.splitlines(s.count('\n'))  
 # Splits string at all (or num) NEWLINEs and returns a list  
 # of each line with NEWLINEs removed.  
Out[55]: ['This is a new country']

In [56]: print('Hello World'.isalpha())  
 ### The isalpha() methods returns bTrueb if all characters in the string are alphabets  
 print('HelloWorld'.isalpha())  
 # Space is not considered as alphabet !  
Out[56]: False  
True

In [57]: d1 = '12211'

d2 = '12345.67'  
 print(d1.isdecimal())  
 # Returns true if a unicode string contains only decimal characters  
 # and false otherwise.  
 print(d2.isdecimal())  
Out[57]: True  
 False

**Search and replace**

The most important re method that uses regular expressions is sub.

**Syntax**: re.sub(pattern, replacement, string, max=0)

This method replaces all occurrences of the regular expression pattern in string with replacement, substituting all occurrences unless max provided. This method returns the new, modified string.

For example, consider the phone number '93999-12345'. Remove '-' from the number string and display all the digits of the phone number.

In [60]: import re

phone\_number = '93999-12345'

# Remove "-"

num = re.sub(r'\D',"", phone\_number)

print(num)

9399912345

# **Conditionals and iterations**



## **Conditionals**

*Conditional execution* enables us to check conditions before deciding the flow of program execution. For example, if statement.

***Note:***

* *There is no limit on the number of statements that can appear in the body of if statement, but there must be at least one.*
* *There are occasions when you have no statements for the code that is code is not written yet. Under those circumstances, we can use* pass *statement which does nothing.*
* *We use a stub or place holder for the code not written yet but this is very useful for integration with other written modules.*

In [58]: x = 5; y = 7  
 if x % 2 == 0:  
 print('x is even')  
 elif x % 2 == 1:  
 print('x is odd')  
Out [58]: x is odd



### **Chained conditional**

*When there are more than two possibilities, we need more than two branching requiring chain conditional.*

In [59]: x = 7  
 y = 6  
 if x > y:  
 print('x > y')  
 elif x < y:  
 print('x > y')  
 else:  
 print('x and y are equal')

Out [59]: x > y

## **Nested conditional**

*One if statement can be nested within another. In the following example, the first branch contains a print statement. The second branch contains another if statement which has two branches of its own. These two are print statements.*

In [62]: x = y = 10  
 if x < y :  
 print('x < y')  
 elif x > y:  
 print('x > y')  
 else:  
 print('x = y')  
x = y

## **Iterations or Loop**

1. **For Loop**

*FOR loop is used for iterating a variable over a sequence (a list or string).*

In [60]: for x in range(10, 20):  
 # range of numbers from 10 to 19 last number 20 is not included  
 print(x)  
 # print each element of the list one by one

10  
11  
12  
13  
14  
15  
16  
17  
18  
19

1. **While Loop**

* *If the condition is false, exit the while statement and continue execution at the next statement*
* *Execute number of statements in the body till the condition passed in while is true.*

In [61]: x = 10  
 while x < 20:  
 print(x)   
 if x % 2 == 0:  
 # when x is odd number skip  
 print(' Even number found')  
 x += 1  
 if x == 19:  
 break  
 # When x is 18 stop the iteration  
 print('\nProgram ends')  
 10  
 Even number found  
11  
12  
 Even number found  
13  
14  
 Even number found  
15  
16  
 Even number found  
17  
18  
 Even number found  
  
Program ends

***Continue and break***

***How to break the loop and come out under some conditions? Use break.***

***How do you skip further processing and force the next iteration? Use continue.***

In [63]: for x in range(10, 20):  
 # range of numbers from 10 to 19 last number 20 is not included  
 print(x)  
 if x % 2 == 0:  
 # when x is odd number skip  
 print(' Even number found')  
 continue  
 if x == 19:  
 break  
 # When x is 19 stop the iteration  
 print('\nProgram ends')  
10  
 Even number found  
11  
12  
 Even number found  
13  
14  
 Even number found  
15  
16  
 Even number found  
17  
18  
 Even number found  
19  
  
Program ends

Example: Draw an inverted triangle with \* - first line with 5 stars, second line with 4 stars and so on.

In [64]: for i in range(1, 6):  
 for j in range(5, i - 1, -1):  
 print('\*', end = ''),  
 print()  
\*\*\*\*\*  
\*\*\*\*  
\*\*\*  
\*\*  
\*

# **Standard Modules**

*A module is a file that contains a collection of related functions grouped together. Before we use the function from a module, we need to import them.*

*For example, to use the function from os module, you need to specify the following: import os*

*To specify a function in located in the module, we need to specify the name of the module and the name of the function, separated by a dot.*

*For example, os.getcwd() which gives the current working directory*

In [65]: import os  
 os.chdir('D:/RRD/Courseware')  
 print(os.getcwd())

D:\RRD\Courseware

*Get the contents of the current working directory, use listdir.*

In [66]: print(os.listdir(os.getcwd()), sep = '\n')  
['Chapter\_1\_Python\_Basics.ipynb', 'Chapter\_1\_Python\_Basics.pdf']

*Change working directory by using chdir()*

In [67]: os.chdir('D:\RRD\RRD\_March\_2019')  
 print(os.getcwd())  
D:\RRD\RRD\_March\_2019

In [68]: os.chdir('D:\RRD\RRD\_March\_2019\Workbook')  
 print(os.getcwd())  
D:\RRD\RRD\_March\_2019\Workbook

*To create a directory*

In [69]: os.mkdir('Delete')  
In [70]: os.getcwd()  
Out[70]: 'D:\\RRD\\RRD\_March\_2019\\Workbook'

In [71]: directories = [ x for x in os.listdir('.') if os.path.isdir(x)]  
 directories  
['.ipynb\_checkpoints', 'data', 'Delete', 'images']

*How do you delete a directory using Python?*

In [72]: os.removedirs('Delete')  
In [73]: directories = [x for x in os.listdir('.') if os.path.isdir(x)]  
 directories  
['.ipynb\_checkpoints', 'data', 'images']

We observe that the directory, Delete is deleted and not appearing in the list of directories above.

To check if a file exists in the current directory, use os.path.isfile

In [74]: os.path.isfile('./images/RAF.jpg')  
Out[74]: *True*

To check what are the modules installed with their version number

In [75]: import pkg\_resources  
 dists = [str(d).replace(" "," Version: ") for d in pkg\_resources.working\_set]  
 for i in dists:  
 print(i)

wordcloud Version: 1.5.0  
wit Version: 5.1.0  
wincertstore Version: 0.2  
win-unicode-console Version: 0.5  
…

tables Version: 3.4.2  
sympy Version: 1.1.1  
statsmodels Version: 0.9.0  
SQLAlchemy Version: 1.1.13  
spyder Version: 3.2.4  
…

seaborn Version: 0.8  
scipy Version: 0.19.1  
scikit-learn Version: 0.19.1  
scikit-image Version: 0.13.0  
ruamel-yaml Version: 0.11.14  
…

pywin32 Version: 221  
…  
pytest Version: 3.2.1  
…

How do you check which version of numpy is installed in my Computer? This is already available in the above list.

In [76]: import numpy as np  
 print(np.\_\_version\_\_)  
1.15.4

How do you check the version of Python I am using?

The platform module includes tools to see the platform, hardware, operating system, and interpreter version information where the program is running.

platform.python\_version() returns the Python version as string 'major.minor.patchlevel'

In [77]: from platform import python\_version  
 print(python\_version())  
3.6.3

# **Date Time Operations**

<https://docs.python.org/3/library/datetime.html>

We require datetime module to work with dates as date objects.

Print current time in year (4 digit), month, day, hour, minute, second and microsecond.

In [78]: import datetime  
 print(datetime.datetime.now())

Out [78]: 2019-03-28 15:38:29.806057



## **Create a date object**

To create a date, we use the datetime() class (constructor) of the datetime module. The datetime() class requires only three parameters such year, month and day to create a date. Rest of the six parameters such as hour, minute, second, microsecond have (0 as default value) and TZ (time zone) ( None as default value) are optional.

In [79]: import datetime

dt = datetime.datetime(2019,5,4)  
 print(dt)

2019-05-04 00:00:00

**strftime()** method is used for formatting date objects into readable strings. This takes one parameter, format which specified the format of the returned string. For example, display the week day name of 1/6/2019

In [80]: import datetime  
 dt = datetime.datetime(2019, 6, 1)  
 print(dt.strftime("%A"))

Saturday

strptime() **method parses a string representing a time according to a format. The return value is a struct\_time as returned by gmtime() or localtime().**

In [81]: import time  
 struct\_time = time.strptime('30 April 2019', "%d %B %Y")  
 print(struct\_time)

time.struct\_time(tm\_year=2019, tm\_mon=4, tm\_mday=30, tm\_hour=0, tm\_min=0, tm\_sec=0, tm\_wday=1, tm\_yday=120, tm\_isdst=-1)

Get the difference between two dates in python

In [82]: import datetime

dt1 = datetime.datetime(2019, 6, 1)  
 dt2 = datetime.datetime(2019, 7, 1)  
 diff = dt2 - dt1  
 print("\nDifference between two dates is", diff)

Difference between two dates is 30 days, 0:00:00

Add or subtract a specified number of days to a date

In [83]: import datetime from datetime  
 import timedelta  
 dt2 = datetime . datetime(2019, 7, 1)  
 d = timedelta(days = -60)  
 # Subtract 60 days  
 dt3 = dt2 + d  
 print("\nDate before 60 days to 2019-07-01 ")  
 print(dt3.strftime("%A-%m-%d-%Y"))

Date before 60 days to 2019-07-01   
Thursday-05-02-2019

If a person's date of birth is 1979-07-01 and retirement age is 65 years, how long (years, months and days) the person has to wait for his / her retirement?

In [84]: import datetime from datetime  
 import timedelta  
 dob = datetime.datetime(1979, 7, 1)  
 sixty\_five\_years = timedelta(days = 365.25) \* 65  
 retirement\_date = dob + sixty\_five\_years  
 print("\nRetirement date ", retirement\_date)  
 diff = retirement\_date - datetime.datetime.now()  
 print("\nDays before retirement ")  
 print(diff)

Retirement date 2044-06-30 06:00:00  
  
Days before retirement   
9225 days, 14:21:29.563554

Get the name of the customer by function, input(). This takes input from the user we make use of a built-in function input().

Greet the customer based on the time of the day; Good morning, Good Afternoon, Good evening.

In [85]: customer\_name = input()  
Robert  
In [86]: currentTime = datetime.datetime.now()  
 # Since no time zone (tz) is specified, it uses local timezone  
 #   
 if currentTime.hour < 12:  
 greet\_msg = 'Good morning '  
 elif 12 <= currentTime.hour < 18:  
 greet\_msg = 'Good afternoon '

else:  
 greet\_msg = 'Good evening '

print('\nHi %s %s' % (customer\_name, greet\_msg))  
 print('\nDate / Time is ' + currentTime.strftime("%A: %m-%d-%Y %H:%M Hours"))

Hi Robert Good afternoon   
  
Date / Time is Thursday: 03-28-2019 15:39 Hours

# **User defined functions**



## **Function**

A *function* is a named sequence of statements that performs a specified operation in the *function definition*.

Keyword **def** is used to start the function.

**def** is followed by **function name** followed by *()* - parenthesis

Within the parenthesis, we can specify the arguments for the function.

A colon(:) marks the end of function header

Indentation (space - normally four spaces) is given to the body of the function.

Syntax for a function definition:

def function\_name (list of parameters):

''' Description about the function '''

return

A docstring is a string literal that occurs as the first statement in a module, function, class, or method definition. Such a docstring becomes the doc special attribute of that object.

All modules should normally have docstrings, and all functions and classes exported by a module should also have docstrings.

## **Statements**

There can be one or more statements inside the function, but they have to indented to the left margin. You may use four spaces of indentation.

The function is named specified by you in the def statement. Creating a function can make a program smaller by eliminating the repeated code.

You create a function, prior to its execution. Statements inside a function are not executed inside the function until the function is called.

The *return* statement terminates the program execution before it reaches its end.

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

You can define a function inside another.

Functions can take zero or more arguments. Inside the function, the values that are passed are assigned to variables called *parameters*.

For example, in the following example, you have two arguments and assigned to parameters x and y respectively.

When you create a local variable inside a function, it only exists inside the function and not outside. For example, in the following function, add\_numbers, we have created a variable, sum. When add\_numbers is terminated, the variable, sum is destroyed.

Parameters are also local. Outside the function, add\_numbers(), both x and y don't exist.

**Points to remember**

The colon (:) indicates the beginning of the block.

The indented statements that follow are called block.

In [87]: def add\_numbers(x , y):  
 '''  
 This function accepts two numbers x and y then adds these  
 two numbers and return their sum   
 '''  
 if isinstance(x, float) & isinstance(y, float):  
 sum = x + y  
 return sum  
 return 0

### End of function   
  
Print the doc string of the function, add\_numbers

In [88]: print('\ndocstring')  
 print(add\_numbers.\_\_doc\_\_)

docstring  
  
 This function accepts two numbers x and y then adds these  
 two numbers and return their sum

Add two numbers 5\_000, 6\_000.

In [89]: print('\nSum of two numbers')  
 x = 5000  
 y = 6000  
 sum = add\_numbers (float(x), float(y))  
 print(int(sum))

Sum of two numbers  
11000

## **ANONYMOUS FUNCTION, LAMBDA**

In Python, Lambda is an expression. Lambda's body is a single expression, not a block of statements.

In [90]: square = lambda x : x \*\* 2  
 square(4)  
Out[90]:

16

## **Scope of variables**












### **Local variable**

This is declared inside a function body

This cannot be accessed outside the function

### **Global variable**

This is declared outside a function body

Can be accessed outside the function, anywhere in the program

In [91]: x = 10  
 # global   
 def change\_x(x):  
 x += 5  
 # Now x is a local variable  
 print('Inside the function input value of x is %d ' % x)  
 y = x % 3  
 print('Value of y is %d ' % y )  
 change\_x(x)  
 print('Outside the function x is %d' % x)

Inside the function input value of x is 15  
Value of y is 0  
Outside the function x is 10

Define a function, check\_even()

In [92]: def check\_even (x):  
 if isinstance (x, int):  
 print('x is an integer')  
 if x % 2 == 0:  
 print('x is even', x)  
 else :  
 print('x is odd', x)  
 else:  
 print('Please enter an integer value')  
In [93]: x = 5  
 check\_even(x)

x is an integer  
x is odd 5

In [94]: def withinRange (x, y, z):  
 return z >= x & z <= y

## **Recursion**

A function can call itself. A process of function calling itself is recursion and such functions are called recursive.

If you write a function in such a way that the program never terminates, you have a situation of infinite recursion. Python detects such recursions when the maximum recursion depth is reached and throws an error message.

Write a function to find the factorial of a given number.

In [95]: def factorial (x):  
 '''  
 This function computes the factorial of a given number, x by using definition,   
 0! = 1 and n! = n (n-1)!  
 '''  
 if isinstance(x, int):  
 if x == 0:  
 return 1  
 else:  
 result\_recurse = factorial(x-1)  
 result = x \* result\_recurse  
 return result  
 print('Please give integer values only')  
  
In [96]: factorial(5)  
Out[96]: 120

In [97]: factorial('x')  
Please give integer values only

In [98]: print(factorial.\_\_doc\_\_)

This function computes the factorial of a given number, x by using definition,   
 0! = 1 and n! = n (n-1)!

## **Boolean function**

Functions can return Boolean values. Consider the following example, consider the function withinRange() written earlier in this section.

In [99]: withinRange(1, 9, 6)  
Out[99]: True

# **Random numbers**

There are times when we want the computer to be non-deterministic (not predictable) like games. One of the ways of making a program to be truly non-deterministic is to generate and random numbers.

The **random** module contains a function called random that returns a floating point number between 0.0 and 1.0. So each time the function **random** is invoked, a new random number is obtained.

In [100]: import random  
 for i in range(5):  
 ### 5 iterations  
 x = random.random()

print('\n i = %d ===>> x = %0.6f ' % (i, x))

i = 0 ===>> x = 0.344128  
  
 i = 1 ===>> x = 0.687977  
  
 i = 2 ===>> x = 0.974008  
  
 i = 3 ===>> x = 0.563206  
  
 i = 4 ===>> x = 0.120864

To generate a random number between low and high, multiply x with high.

In [101]: import random

Low = 0; high = 300  
 print("\nGenerate random numbers between %d and %d " % (low, high))  
 for i in range (5):  
 ### 5 iterations  
 x = round(random.random() \* high, 0)  
 print('\n i = %d ===>> x = %3.0f' % (i, x))  
  
Generate random numbers between 0 and 300  
  
 i = 0 ===>> x = 70  
  
 i = 1 ===>> x = 73  
  
 i = 2 ===>> x = 251  
  
 i = 3 ===>> x = 91  
  
 i = 4 ===>> x = 105

How will you include upper limit in the random number range of 0 to 100?

In [102]: import random   
 n = 100  
 # the number of random numbers  
 low = 0  
 high = 100  
 ## Set seed, a starting point to get the same list  
 random.seed = 1823333563433434343  
 print("\nGenerate random numbers between %d and %d" % (low, high))  
 random\_no\_list = [0] \* n  
 # Initialize the list with 0s  
 for i in range (100):  
 ### 5 iterations  
 random\_no\_list[i] = random.randint(low,high)  
 print('\nFirst 100 Random numbers from the list')  
 print(random\_no\_list[:100])  
 high\_random\_values = [100 in random\_no\_list ]  
 print(high\_random\_values)  
  
Generate random numbers between 0 and 100  
  
First 100 Random numbers from the list  
[94, 36, 82, 64, 21, 40, 47, 33, 85, 8, 71, 78, 65, 5, 69, 66, 4, 59, 86, 90, 15, 97, 25, 8, 15, 45, 93, 99, 23, 91, 6, 61, 39, 27, 28, 31, 17, 15, 59, 17, 66, 49, 96, 43, 83, 98, 94, 22, 76, 71, 39, 15, 48, 96, 6, 52, 46, 44, 53, 5, 16, 63, 71, 82, 40, 29, 13, 83, 25, 80, 49, 85, 32, 67, 77, 21, 66, 67, 18, 57, 17, 98, 54, 66, 37, 14, 14, 96, 48, 65, 92, 20, 9, 89, 4, 77, 59, 34, 64, 17]  
[False]

In [103]: import random  
 n = 100  
 # the number of random numbers  
 low = 0  
 high = 1000  
 print("\nGenerate random numbers between %d and %d" % (low, high))  
 random\_no\_list = [0] \* n  
 # Initialize the list with 0s  
 for i in range (100):  
 ### 5 iterations  
 random\_no\_list[i] = random.randrange(low, high)  
 print('\nFirst ten random numbers from the list')  
 print(random\_no\_list[:10])

Out [103]: Generate random numbers between 0 and 1000  
 First ten random numbers from the list  
 [283, 990, 354, 135, 228, 185, 471, 470, 280, 65]

# **Error handling in Python**

Error handling is done through the use of exceptions.

In the event of an error occurring, a try block stops execution and hands over the same to except blocks.

You can use finally block to execute the statements regardless of whether an exception occurs.

You can raise an exception such as xx in your program by using raise exception[,] statement. It breaks the current code execution and returns the exception block until it is handled.

There are common exception errors that can be trapped.

|  |  |  |
| --- | --- | --- |
| # | Error type | When Exception is raised |
| 1 | IOError | if the file cannot be opened |
| 2 | ImportError | if Python cannot find the module |
| 3 | ValueError | if a built-in operation or function receives an argument that has the right type but an inappropriate value |
| 4 | KeyboardInterrupt | when the user hits the interrupt key (Ctrl - C or Del key) |
| 5 | EOFError | when end-of-file condition has reached without reading any data |
| 6 | ZeroDivisionError | when an attempt is made to divide a number by zero |
| 7 | TypeError | if a wrong type is given for a calculation or a built-in function |
| 8 | NameError | If a variable's name can't be found |

In [1]: *### General exception*

try:

x = 's'  
x = x / 5

except:

print("\Error in the value given ")

finally:

print("\nExecuted the code with try - exception block")

Error in the value given

Executed the code with try - exception block

In [2]: ### Example of trapping ZeroDivisionError  
 try:  
 x = 1 / 0  
 except ValueError:  
 print("\nError in the value given ")  
 except ZeroDivisionError:  
 print("\nAttempt to divide by zero")  
 finally:  
 print("\nExecuted the code with try - exception block")

Attempt to divide by zero

Executed the code with try - exception block

In [3]: ### Example of TypeError  
 try:  
 x = 's'  
 x1 = x / 5  
 except TypeError:  
 print("\nError in the type of value given ")  
 except ZeroDivisionError:  
 print("\nAttempt to divide by zero")  
 finally:  
 print("\nExecuted the code with try - exception block")

Error in the type of value given

Executed the code with try - exception block

*For more details, refer https://www.pythonforbeginners.com/error-handling/python-try-and-except*

# **Regular expression**

https://docs.python.org/3.4/library/re.html

Regular expressions, regex patterns are essentially a tiny, highly specialized programming language contained in Python.

The re module provides regular expression matching operations.

The re module functions fall into three categories: pattern matching, substitution and splitting. A regex describes a pattern to locate in the text.

You can compile the regex with re.compile forming a reusable regex object. This is highly recommended if you intend to apply the same expression to many strings since this will save time of execution.

**Regular expression methods**

|  |  |  |
| --- | --- | --- |
| **#** | **Argument** | **Description** |
| 1 | findall | Return all non-overlapping matching patterns in a string as a list |
| 2 | finditer | Like findall, but returns an iterator |
| 3 | match | Match pattern at start of string and optionally segment pattern components into groups; if the pattern matches, returns a match object, and otherwise None |
| 4 | search | Scan string for match to pattern; returning a match object if so; unlike match, the match can be anywhere in the string as opposed to only at the beginning |
| 5 | split | Break string into pieces at each occurrence of pattern |
| 6 | sub, subn | Replace all (sub) or first n occurrences (subn) of pattern in string with replacement expression; use symbols \1, \2, ... to refer to match group elements in the replacement string |

The function findall returns all matches in a string.

The functions, match and search are closely related to findall.

The function, search returns only the first match.

The function, match only matches at the beginning of the string.

Assume you have a string with variable number of white spaces, tabs and new lines.

\s+ is the regex for describing one or more whitespace characters.

\t is for tabs and \n is for new line

In [4]: import re  
 txt = 'Arun Arjun Babitha Charan\t Dominic\nEbinezer'  
 print('\nBefore the split')  
 print(txt)  
 print('\nAfter the split')  
 regex = re.compile('\s+')  
 print(regex.split(txt))  
 print('\nTo get a list of all patterns matching the regex')

regex.findall(txt)

Before the split

Arun Arjun Babitha Charan Dominic Ebinezer

After the split ['Arun', 'Arjun', 'Babitha', 'Charan', 'Dominic', 'Ebinezer']

To get a list of all patterns matching the regex

Out[4]: [' ', ' ', ' ', '\t ', '\n']

**Note the regular expression symbols**

[ ] - indicates a set of characters and characters can be listed individually or collectively as in a range

The regular expression [A-Z0-9-] indicates the pattern of any letter (alphabet) A to Z or any number 0 to 9, underscore() or hyphen (-)

Special characters lose their special meaning inside [] sets.

+ symbol causes the regular expression to match 1 or more repetitions of the preceding regular expression. For example xy+ will match 'x' followed by any non-zero number of 'y', but it will not match just 'x'.

@ matches @ in the string

.[A-Z]{3,4} causes the regular expression to match

'.' dot outside the set [], matches any character except the newline.

'.' escapes the special characters , allowing to match characters like \*, ? and (.) - See the above point. We need to differentiate from the above.

[A-Z]{3,4} causes the resulting regular expression to match from 3 to 4 repetitions of the preceding RE, attempting to match as many repetitions as possible. For example, [A-Z]{2,4} will match any Alphabet (letter) from 2 to 4.

In [5]: email\_ID = """Arun arun22@gmail.com\Bharath bharath380@gmail.com\Gughan gughan@yahoo.co.in [abc\_91@yahoo.co.in](mailto:abc_91@yahoo.co.in)"""

pat = r'[A-Z0-9.\_-]+@[A-Z0-9.-]+\.[A-Z]{2,4}' *# regex pattern*  
 regex = re.compile(pat, flags = re.IGNORECASE)

*# flags = re.IGNORECASE make it case insensitive to consider both*

*# upper case and lower case*  
 regex.findall(email\_ID)

Out[5]: ['arun22@gmail.com',

'bharath380@gmail.com',

'gughan@yahoo.co.in',

'abc\_91@yahoo.co.in']

In [6]: mo = regex.search(email\_ID)  
 mo

Out[6]: <\_sre.SRE\_Match object; span=(5, 21), match='arun22@gmail.com'>

The search returns a special match object for the first email address in the text informing us the start and end position of the pattern in the string.

In [7]: email\_ID[mo.start(): mo.end()]

*### get the first email address at the specified range*

*### (mo.start() and mo.end())*

Out[7]: 'arun22@gmail.com'

regex.match returns will match if the pattern occurs at the start of the string.

In [8]: print(regex.match(email\_ID))

None

In the string, email\_ID the pattern (of email address) does not occur at the start of the string. Hence None is displayed.

In [9]: cust\_name = "Peter Reynolds"  
 mg = re.match(r"(\w+) (\w+)", cust\_name)

*### \w matches includes most characters that can be part of a word*

*### in any language, as well as numbers and the underscore*  
 print("\nFull name")  
 print(mg.group(0))  
 print("\nFirst name")  
 print(mg.group(1))  
 print("\nSurname")  
 print(mg.group(2))

Full name Peter Reynolds First name Peter Surname Reynolds

In [10]: mg = regex.findall(email\_ID)  
 print(mg)

['arun22@gmail.com', 'bharath380@gmail.com', 'gughan@yahoo.co.in', 'abc\_91@yahoo.co.in']

The argument sub will return a new string with occurrences of the pattern replaced by a new string.

The argument, sub also has access to groups in each match using the special symbols like \1 , \2, \3 corresponding to the first matched group , second matched group and third matched group respectively.

An example of using the sub() method. It replaces colour names with the word colour:

In [11]: rt = re.compile('white|black|blue|red')  
 rt.sub('colour', 'white socks and black shoes')

Out[11]: 'colour socks and colour shoes'

If you want to split a sentence into words, you can use split()

In [12]: pr = re.compile(r'[^a-zA-Z]', flags = re.IGNORECASE)

*# ^ indicates expression beginning with*  
 pr.split('This is a test sentence')

Out[12]: ['This', 'is', 'a', 'test', 'sentence']

Extract numbers from the text such as description. Get the age in number occuring after the key word 'age is'.

In [13]: s = 'My age is 31 years and income is Rs 43000'  
 words = s.split('age is',2)  
 print('\nAfter splitting the sentence My age is 31 years \

and income is Rs 43000')  
 print('\nWe get the two strings, before and after "age is" ', words)  
 print('Again split to get only age')  
 words\_age = s.split('years',2)

*# Split the words into two before the word years and after*  
 pr = re.compile(r'\d+')  
 print('After extracting the numerical value of age, we get')  
 print(pr.findall(words\_age[0]))

*# from the words before the word years*  
 print('After extracting the numerical value of income, we get')  
 print(pr.findall(words\_age[1])) *# from the words after the word years*

After splitting the sentence My age is 31 years and income is Rs 43000

We get the two strings, before and after "age is" ['My ', ' 31 years and income is Rs 43000']

Again split to get only age

After extracting the numerical value of age, we get ['31']

After extracting the numerical value of income, we get ['43000']

For more details on regular expression, refer to the following links:

https://www.programiz.com/python-programming/regex https://docs.python.org/3/howto/regex.html https://docs.python.org/3.4/library/re.html

# **File handling**



## **Text files**

**Write and read text file**

**Syntax to open a file object in Python is:**

fp = open('filename', 'mode') where fp is the file pointer.

mode informs us which way the file will be used

**Some useful modes:**

* r read mode for read only
* w write mode - existing file overwritten
* a append mode - add new data at the end
* r+ read and write mode
* write() statements write into a file while read() reads from the existing file
* close() to close the file completely and terminating the resources in use.

**Write numbers from 10 to 20 in a file names as new\_data.txt in the current working directory**

In [14]: fp = open('new\_data.txt', mode = 'w')

for i in range(10,20):

string = str(i) + '\n'

fp.write(string)

fp.close()

**Read the data file, new\_data.txt and print the contents.**

In [15]: fp = open('new\_data.txt', mode = 'r')

for line in fp:

print(line)

fp.close()

10

11

12

13

14

15

16

17

18

19

**Statement with will automatically open and close the file.**

In [16]: with open("new\_data.txt") as f:

for line in f:

print(line)

10

11

12

13

14

15

16

17

18

19

For more details, please refer to <https://www.geeksforgeeks.org/file-handling-python/ss>

**Exercise 1**

Write a function and test the function for a recursively defined mathematical function, fibonacci.

Definition of finonacci of n is given below:

fibonacci(0) = 1

fibonacci(1) = 1

fibonacci(n) = fibonacci(n-1) + fibonacci(n-2)

**Exercise 2**

Generate a list of 200 random numbers between 10 and 1000 (1000 inclusive)

**Exercise 3**

Write a function and test the function for checking if the input number x is a prime number or not. A number is a prime number, when it is not divisible by any other number other than 1 and itself. For example, numbers 3, 5, 7 are prime numbers.

**Exercise 4**

Generate a list of 100 random numbers between 0 and 1000

**Exercise 5**

Find the date away from 30 days from this day.

**Exercise 6**

We notice that error messages will identify what kind of error happened on python statement. Give an example of a NameError occurred using try ... except block.

**Exercise 7**

Extract the user id, domain name and suffix from the following email address list.

email\_address\_list = ['abc123@hotmail.com', 'xyz123@gmail.com', 'pqrs\_12@yahoo.com']

**Exercise 8**

Get the name from five customers and write to a text file and then print the name of those five customers.