PYTHON PROGRAMMING LANGUAGE

Halil Özmen

These Python notes are a brief summary of Python programming language, and contains information about some of the features of Python. These notes are intended to be used as a check list by the instructor who teaches programming courses to students.

The first ten chapters can be used in an introduction to programming course targetting students who has no programming knowledge, and the last four chapters can be used in more advanced Python programming courses.

TABLE OF CONTENTS

Version: 2026-01-01

1. Introduction to Computers and Programming

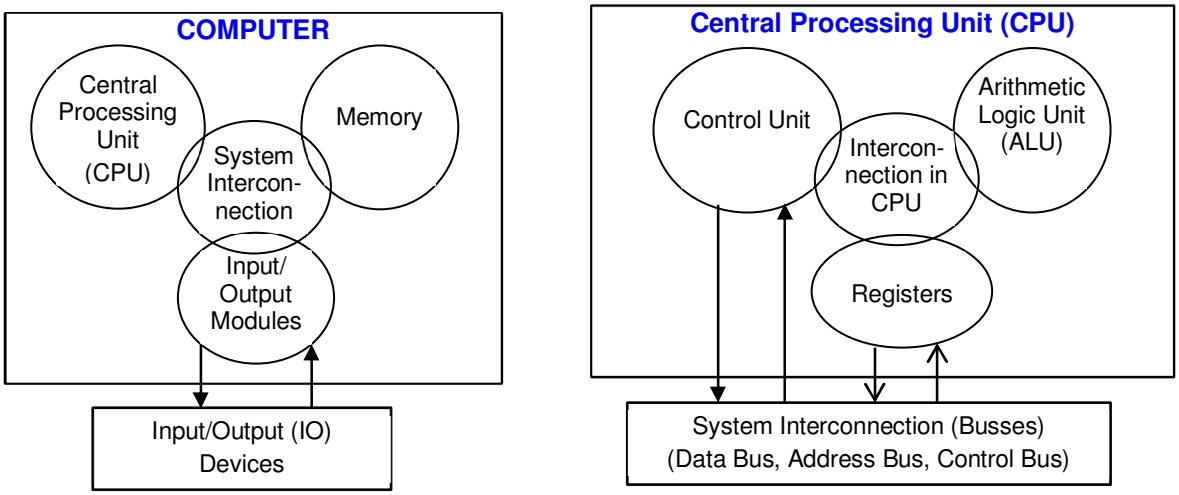
1.1. Introduction to Computers

Computer Components:

A computer consists of central processing unit (CPU), memory and input/output (IO) modules.

CPU controls the computer and instructions are run by the CPU.

CPU consists of control unit, arithmetic logic unit (ALU), registers and interconnections among these units. The control unit controls the CPU.



Input Devices: keyboard, mouse, harddisk, flash disk, microphone, modem, etc.

Output Devices: monitor, harddisk, flash disk, sound speaker, modem, printer, etc.

Fetch-Execute Cycle:

The execution of a machine code program on a von Neumann architecture computer occurs in a process called fetch-execute cycle:

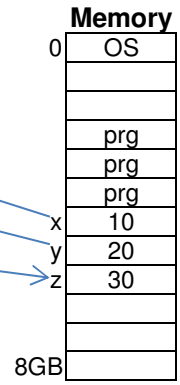
```
initialize the program counter
repeat forever
    fetch the instruction pointed by the program counter
    increment the program counter to point at the next instruction
    decode the instruction
    execute the instruction
end repeat
```

Running Programs in Computer:

While running programs, both the **program** (computer instructions) and **data** are stored in the **memory**.

Program statements and CPU Instructions:

Program Statements	CPU Instructions
z = x + y	Read location x Read location y Add Write to location z
print(z)	Read location z Send to output



The programming languages are easier to understand and write than CPU instructions.

Computer programs need to be translated to CPU instructions for the CPU to understand them and run them.

1.2. Introduction to Computer Programming

Computer Program:

A computer program is a sequence or set of instructions in a programming language for a computer to execute.

Computer Programming Process:

Computer programming is the process of performing a particular computation, usually by designing and building an executable computer program. Programming involves tasks such as analysis, generating algorithms (design and implementation), profiling algorithms' accuracy and resource consumption, and the implementation of algorithms.

Programming Languages:

A programming language is a system of notation for writing computer programs.

Assembly Language is any low-level programming language with a very strong correspondence between the instructions in the language and the architecture's machine code instructions.

A **high-level language** is any programming language that enables development of a program in a much more user-friendly programming context and is generally independent of the computer's hardware architecture.

Machine Language: 01101011 01010100 10111011 11101010 00100111 ....

Assembly Language:

```
LBL4:   MOV     AX,8
        ADD     R3, #32
        CMP     AL, BL
        JE      LBL4
```

High Level Language:

```
a = int(input("Enter an integer number: "))
if a % 2 == 0:
    a = a / 2
print(a)
```

History of Programming Languages: (only some major languages are listed)

1951 - Assembly Language	1978 - SQL (query language)
1954 - FORTRAN	1980 - C++
1958 - LISP	1984 - MATLAB
1958 - ALGOL	1990 - Python
1959 - COBOL	1993 - R
1964 - BASIC	1995 - Java
1964 - PL/I	1995 - PHP
1970 - Pascal	2002 - Scratch
1972 - C	....

Assembler, Compiler, Linker, Loader, Interpreter:

- **Assembler** translates a program in an assembly language to machine instructions.
- **Compiler** translates a program in a high-level language to machine instructions.
- **Linker** is a program which helps to link object modules of a program into a single object file. It performs the process of linking. Linking is a process of collecting and maintaining piece of code and data into a single file.
- **Loader** is a component of an operating system that is accountable for loading programs and libraries to computer memory.
- **Interpreter** is a computer program that directly executes instructions written in a programming or scripting language, without requiring them previously to have been compiled into a machine language program.

2. Introduction to Python

Python is an easy to learn, powerful programming language. It has efficient high-level data structures and a simple but effective approach to object-oriented programming.

Python's elegant syntax and dynamic typing, together with its interpreted nature, make it an ideal language for scripting and rapid application development in many areas on most platforms.

Some of Python's notable features:

- Python has an elegant syntax, making the programs you write easier to read.
- It is an easy-to-use language that makes it simple to get your program working. This makes Python ideal for prototype development and other ad-hoc programming tasks, without compromising maintainability.
- It comes with a large standard library that supports many common programming tasks such as connecting to web servers, searching text with regular expressions, reading and modifying files.
- Python's interactive mode makes it easy to test short snippets of code. There's also a bundled development environment called **IDLE** (Integrated Development and Learning Environment).
- It is easily extended by adding new modules implemented in a compiled language such as C or C++.
- It can also be embedded into an application to provide a programmable interface.
- It runs anywhere, including Windows, Linux, Unix and Mac OS, with unofficial builds also available for Android and iOS.
- It is free software in two senses. It doesn't cost anything to download or use Python, or to include it in your application. Python can also be freely modified and re-distributed, because while the language is copyrighted it's available under an open source license.

Some programming-language features of Python are:

- A variety of basic data types are available: numbers (floating point, complex, and unlimited-length long integers), strings (both ASCII and Unicode), lists, and dictionaries.
- Python supports object-oriented programming with classes and multiple inheritance.
- Code can be grouped into modules and packages.
- The language supports raising and catching exceptions, resulting in cleaner error handling.
- Data types are strongly and dynamically typed. Mixing incompatible types (e.g. attempting to add a string and a number) causes an exception to be raised, so errors are caught sooner.
- Python contains advanced programming features such as generators and list comprehensions.
- Python's automatic memory management frees you from having to manually allocate and free memory in your code.

2.1. Installation of Python

To install Python, download Python installation program (python-3.14.0-amd64.exe or similar) from web site **python.org**, and run it. This will install Python and its environment to your computer.

Online Python compilers: **https://onecompiler.com/python**
https://www.programiz.com/python-programming/online-compiler/
https://www.w3schools.com/python/trypython.asp?filename=demo_compiler

2.2. Running Python Environment

Under Windows, run "IDLE (Python 3.x ... bit)" from Start menu.

The "Python 3.?.? Shell" window will be opened.

The first few lines will be similar to:

```
  Python 3.13.3 (tags/v3.13.3:6280bb5, Apr  8 2025, 14:47:33) [MSC v.1943
  64 bit (AMD64)] on win32
  Enter "help" below or click "Help" above for more information.
>>>
```

After the prompt **>>>** the user can enter Python commands (statements).

Wing Personal IDE can be used to write and run Python programs, and can be downloaded from: https://wingware.com/downloads/wing-personal.

Creating, Saving and Running Python Programs:

Create and save new Python program in Python IDLE:

In IDLE, click on "New File" in "File" menu to create an empty Python script (program) file. Enter Python statements to this new file, then click "Save" in "File" menu of the new file window, select a directory, enter a file name that ends with "**.py**", and click "Save" button.

Open, update and save an existing Python program:

In IDLE, click on "Open..." in "File" menu. In "Open" window, select the directory and the Python program file, then click "Open" button. Change the program lines. Click "Save" in "File" menu.

Run a Python program:

Press "f5" from keyboard (or "Run Module" in "Run" menu), if "Save Before Run or Check" window appears, then click "OK". The program will be run and the output will appear in shell window.

In **Wing Personal**, click on  to create a new Python program file, write the program in the program window, then click on  to run the program. Wing will first ask you to save this program file, (save it to your Python program directory), then it will run and show the results in "Debug I/O" window.

The "Debug I/O" window may be moved to an appropriate position.

2.3. Basic Python Syntax

In Python statements are not terminated with any symbol, whereas in most programming languages statements end with a special symbol, for example, statements are terminated with ';' in C and Java.

Anything written after a hash # sign on a program line becomes a comment.

Sample Python statements:

```
# This is a comment line
a = 48 / 10                    # A division, then an assignment
pi = 3.1415926535              # A constant value is assigned to a variable
area = pi * (radius ** 2)      # Comment can be written after command
print(a)                       # Value of a variable is printed to screen
print(pi, area)                # Value of two variables are printed to screen
print("Hello Galaxy!")         # A text is printed
```

Spaces at the beginning of Python lines are important, but not elsewhere.

Inside the same block, lines must be preceeded by equal number of tabs or spaces.

```
a = 48 / 10
  print(a)                     # ERROR for this statement: indent error
```

2.4. Data Types

Data Types in Python:

Integer: 4 -19 0 10741 -20186 12

Float: 3.14159 0.008 12.0 12. -801.5 -14.0 4.1e10

Complex Number: 2+3j 4.8-2.1j

String: "Antalya" "7" 'Antalya is beatiful!' '3.14159' "1254"

Boolean: True False (ATTENTION: not TRUE or true or FALSE or false)

type() function returns the type of its parameter.

```
>>> type(7)
<class 'int'>
>>> type(7 * 8)
<class 'int'>
>>> type(7.0)
<class 'float'>
```

```
>>> type(8 // 2)
<class 'int'>
>>> type(8 / 2)
<class 'float'>
```

```
>>> type("07 Antalya")
<class 'str'>
>>> type(True)
<class 'bool'>
>>> type(7.2 + 4j)
<class 'complex'>
```

Strings:

Strings will be covered more deeply in subsequent chapters, here this is an introduction to strings.

String constants are delimited with a single quote ('), double quote ("), triple single quote (''') or triple double quote (""").

```
a = 'Antalya'
ab = "Antalya is beautiful"
```

```
s1 = '40' + "8"           # "408"
t1 = """Ali'yi Kaan"a, \nabcd 123456 efgh"""        # \n : newline.
print(t1)
Ali'yi Kaan"a,
abcd 123456 efgh
```

2.5. Variables

In Python, variable names must start with an alphabetical character (A-Z, a-z) or underscore '_', and can contain alphanumeric character (A-Z, a-z, 0-9) or underscore '_'.

Variable names are case sensitive. Example: a and A are two different variables.

Variable names can not start with a digit and can not contain space or other special characters.

Valid variables: a z A Z a8 _a8 _ _0 a1 _1a c3po C3pO

Invalid variable names: 8a a-8 a 8 a&b a^b

Good variable names make programs more readable by humans. Since programs often contain many variables, well-chosen variable names can make a program more easily understandable.

```
a = 7
pi = 3.1415962535
city = "Ankara"
x = 10
print('x =', x)
x = 20                    # value of x is changed
print('x =', x)
x = 3.14159               # value of x is changed again
print('x =', x)
x = 'Antalya'             # x has been changed one more time
print('x =', x)
x = '30' + "4444"         # x has been changed once more
print('x =', x)
```

Expressions:

An expression contains one or more variables or constants, and may contain operators and/or parenteses.

```
7 + 8                     10741 // (2 ** 8)          a = b + (c * 2) / (a - 1)
7.2 ** 2                  c = (a + 2) / b            b * b - (4 % a * c / (d - 4)) ** 0.5
```

A user variable or function name can not be same as a Python keyword, or an existing Python function name.

Python Keywords (Reserved Words):

False, None, True, and, as, assert, async, await, break, class, continue, def, del, elif, else, except, finally, for, from, global, if, import, in, is, lambda, nonlocal, not, or, pass, raise, return, try, while, with, yield.

2.6. Assignments

The equal symbol "**=**" is assignment operator.

First the expression at the right hand side of = is evaluated, then the result is stored in (assigned to) the variable (or object) on the left side.

```
variable = expression       # expression is computed, result is assigned to variable.
```

There are other assignment operators such as: **+= -= *= /= %= //= **=**

```
a = 7              # 7 is assigned to variable a
pi = 3.14159       # 3.14159 is assigned to variable pi
b = a + 2          # expression a + 2 is computed, result is assigned to variable a.
x, y, z = 100 / 5, -45, 0.5   # Multiple assignment statement.
x, y = y, x                   # Swap (exchange) contents of two variables
n = 88
n += b + c         # Equivalent to:  n = n + (b+c)
n -= 7             # Equivalent to:  n = n - 7
n *= 4             # Equivalent to:  n = n * 4
n /= 10            # Equivalent to:  n = n / 10
n %= 10            # Equivalent to:  n = n % 10
n //= 10           # Equivalent to:  n = n // 10
n **= 2            # Equivalent to:  n = n ** 2
a = "Ant"
a += "alya"        # Equivalent to:  a = a + "alya" -> "Antalya"
```

2.7. Type Conversion

We may need to get a data of a type by converting data of other type. Sometimes type conversion occurs automatically, and in some cases, we may need to use type conversion functions.

Type Conversion Functions:

ascii()	Returns a string containing a printable representation of an object
bin()	Converts an integer to a binary string
bool()	Converts an argument to a Boolean value
chr()	Returns string representation of character given by integer argument
complex()	Returns a complex number constructed from arguments
float()	**Returns a floating-point object constructed from a number or string**
hex()	Converts an integer to a hexadecimal string
int()	**Returns an integer object constructed from a number or string**
oct()	Converts an integer to an octal string
ord()	Returns integer representation of a character
repr()	Returns a string containing a printable representation of an object
str()	**Returns a string version of an object**
type()	Returns the type of an object or creates a new type object

Expression	Result
int('4')	4
int('4.88')	Error
int(8.9876)	8
float(4)	4.0
float("4.88")	4.88

Expression	Result
str(4)	'4'
5 + int('10')	15
'5' + str(10)	'510'
5 + str(10)	Error

2.8. Basic Input and Output

Output is performed with the **print()** function. By default, a space is printed if there are more than one items as parameters of print function.

```
print("This is CS101 course.")
print(a, b, c)          # A space is printed between items, and new-line at the end.
print(a, b, c, sep="-", end="")     # "-" between items, nothing at the end.
```

The **input()** function produces a string from the user's keyboard input. If we wish to treat that input as a number, we can use the int or float function to make the necessary conversion:

```
a = input("Enter anything: ")          # a is a string
n = int(input("Enter a number: "))     # an integer must be entered, error otherwise
x = float(input("Enter a number: "))   # a number must be entered, error otherwise
```

```
name = input('Enter your name: ')
print("Hello", name)
n = int(input('Enter an integer: '))
print("One more is", n+1)
n2 = int(input('Enter an integer: '))
print("Remainder of", n2, "when divided by 10 is", n2 % 10)
x2 = float(input('Enter a number: '))
print("Half is", x2 / 2)
```

3. Operators and Control Flow Structures

3.1. Operators

Unary Operators	**+ - (Eg.: -2, +7.2, c = a * -b)**
Arithmetic Operators	**+ - * /** // (Floor division) **%** (Modulus) ****** (Power)
Comparison (Relational) Operators	**== != > >= < <=**
Logical (Boolean) Operators	**not and or**
Bitwise Operators	**<< >> ^ & |**
Assignment Operators	**= += -= *= /= //= %= **=**

Arithmetic Operators: Addition (+), Subtraction (-) and Multiplication (*) performs as expected.

Division (/): always returns a floating point number. 20 / 5 → 4.0, 17 / 3 → 5.66666666666667

Floor division (//): discards the fractional part. Eg: 48 // 10 will yield 4; 4.8 // 2 → 2.0

Modulus operator (%): computes remainder of an integer division. 17 % 10 → 7, 28 % 2 → 0.

Power operator (**): computes power. Eg: 7 ** 2 → 49, 2 ** 3 → 8.

Comparison (Relational) Operators:

When two values are compared with a comparison (relational) operator, the result is a boolean value: True or False.

==	equal to	`a == b`	`x == 99`
!=	not equal to	`a != 100`	`1919 / 101 != 19`
>	greater than	`kkk > 9`	`a + b > c - 10 / d`
>=	greater than or equal to	`day >= 30`	`v * 2 >= w + 8`
<	less than	`month < 6`	`c < d + e`
<=	less than or equal to	`year <= 2018`	`"Antalya" <= "Ankara"`

Logical (Boolean) Values and Operators:

Logical (Boolean) values are: **True**, **False**. Logical operators are: **not, and, or**.

Truth Tables of Logical (Boolean) Operators:

a	b	not a	a and b	a or b
False	False	True	False	False
False	True	True	False	True
True	False	False	False	True
True	True	False	True	True

Logical (Boolean) Expressions:

```
not t > u
t > u and v == w
t > u or v == w
(t > u and t <= v) or (w > t and w >= v)
```

```
not 0
not 1
not 5
not -17
```

```
not 0.0
not 32.4
not -0.01
```

Precedence of Operators:

Level	Operators	Meaning
1	**()**	Parentheses
2	**	Exponent (power)
3	**+x -x ~x**	Unary plus, Unary minus, Bitwise NOT
4	* / // %	Multiplication, Division, Floor division, Modulus
5	**+ -**	Addition, Subtraction
6	**<< >>**	Bitwise shift operators
7	**&**	Bitwise AND
8	**^**	Bitwise XOR
9	**|**	Bitwise OR
10	**==, !=, >, >=, <, <=, is, is not, in, not in**	Comparisons, Identity, Membership operators
11	**not**	Logical NOT
12	**and**	Logical AND
13	**or**	Logical OR

3.2. Selection Statements

if Statement:

The basic form of an if statement is as follows:

```
if condition:                    # condition is followed by colon ':'
    block of statements          # all block statements are indented equally
```

If the condition is **True**, then the block is executed, then execution continues after the block.
If the condition is **False**, the block is **not** executed, the execution continues directly after the block.

Rules:

The condition must be followed by a colon ':'.
All statements in a block must be equally indented.
I.e. Equal indentation is mandatory inside the same block.

```
x = 1
if x == 1:
    print("x is 1.")
# some Python statements
```

```
if n >= 10 and n <= 99:
    ....
    ....
# After if statement
```

```
a = 24
b = 60
if a < b:
    a = b + 1
    print("a =", a, " b =", b)
# some Python statements
```

```
if a >= b and c == d:
    ....
    ....    if-block statements
    ....
# After if statement
```

if - else Statement:

The basic form of an if - else statement is as follows:

```
if condition:                          # condition is followed by colon ':'
    if-block
else:                                  # else is followed by colon ':'
    else-block
```

If the condition is **True**, then if-block is executed.

If the condition is **False**, then else-block is executed.

It is same as:

```
if condition:
    if-block
if not condition:
    else-block
```

In either case, after if-block or else-block, the execution continues after the if-else statement.

```
a, b = 10, 20
if a > b:
    print("a > b")        # These two statements forms the if-block
    b = b * 2
else:
    print("a <= b")       # These two statements forms the else-block
    a *= 2
```

Nested if Statement:

Nested if statements are structures where there is (inner) if-statement inside another (outer) if-statement.

If there are nested if-statements only inside else-blocks, then if- elif statements can be used.

General Form of Nested if:

Now let's examine the general form of nested if's. There may be nested if inside if-blocks as well as inside else-blocks.

```
if condition-1:
    if condition-2:
        block-1
    else:
        block-2
else:
    if condition-3:
        block-3
    else:
        block-4
```

```
if gender == "F":
    if age < 20:
        # do things here
    else:
        # do other things here
else:        # case: gender == "M"
    if age < 20:
        # some actions here
    else:
        # other actions here
```

Example:

```
a = int(input("Enter a number: "))
b = int(input("Enter a number: "))
if a < 0:
    if b < 0:
        print("Both numbers are negative.")
    else
        print("Only the first number is negative.")
else
    if b < 0:
        print("Only the second number is negative.")
    else
        print("None of numbers are negative.")
```

if - elif Statement:

The basic form of an if - elif - else statement is as follows:

```
if condition1:
    block-1
elif condition2:
    block-2
elif condition3:
    block-3
elif condition-n:
    block-n
....(a series of elif statements)
else:
    else-block
........
```

```
if condition1:
    block-1
else:
    if condition2:
        block-2
    else:
        if condition3:
            block-3
        else:
            else-block
........
```

The condition are checked sequentially until the first **True** condition is found, then the block of the True condition will be executed, then the program will continue after the whole if-elif-else statement.

```
grade = float(input("Enter grade: "))
if grade >= 90:
    letter = "A"
elif grade >= 70:
    letter = "B"
elif grade >= 50:
    letter = "C"
elif grade >= 40:
    letter = "D"
else:
    letter = "F"
```

```
grade = float(input("Enter grade: "))
if grade < 40:
    letter = "F"
elif grade < 50:
    letter = "D"
elif grade < 70:
    letter = "C"
elif grade < 90:
    letter = "B"
else:
    letter = "A"
```

```
lang = input("Enter language (Tr/Eng/Fra/Ger/Ita/Spa): ")
if lang == "Tr":
    print("Günaydın")
elif lang == "Eng":
    print("Good morning")
elif lang == "Fra":
    print("Bonjour")
elif lang == "Ger":
    print("Guten morgen")
elif lang == "Ita":
    print("Buon giorno")
elif lang == "Spa":
    print("Buenos dias")
else:
    print("Error: Unknown language")
```

Exercices:

1) Write a Python program that inputs an integer number entered by user, and prints its double if it is less than 1000, or print half of it otherwise.
2) Write a Python program that inputs an integer number, then checks if the number is even or odd, and display an appropriate message.
3) Write a Python program that inputs two integer numbers, then checks if the first number is divisible by the second number or not, and display an appropriate message.
4) Write a Python program that inputs two integer numbers, then checks if the first number is greater than twice the second number or less than half of 2nd number, and display an appropriate message.
5) Write a Python program that inputs a grade entered by user, and then assign "A" to letter if grade ≥ 90, "B" to letter if 75 ≤ grade < 90, "C" if 60 ≤ grade < 75, "D" if 45 ≤ grade < 60, and "F" if grade < 45.
6) Write a Python program that inputs an integer, and depending on the value of the number, display "<num> has more than 6 digits", "<num> has 4 to 6 digits", "<num> has less than 4 digits".
7) Write a Python program than inputs an integer number, and prints an appropriate message for the following ranges: < -999, between -999 and -1 (inclusive), between 0 and 999 (inclusive), and > 999.

3.3. Loop Statements

The loop statements are used when there is a need for repeating tasks.

3.3.1. range() Function

Python's built-in function **range()** produces an object that will generate a sequence of integers.

Single argument	range(stop)	the sequence starts at 0 and continues to the integer before stop (0 1 2 ... stop-1). Identical with: range(0, stop)
Two arguments	range(start, stop)	the sequence starts at "start" and continues to the integer before "stop" (till [stop - 1]).
Three arguments	range(start, stop, increase/decrease)	Starts at "start", every time is increased (decreased) by "increase/decrease" and continues to the integer before "stop".

Example range() functions:

```
range(5)          # from 0 (inclusive) till 5 (exclusive):=> 0, 1, 2, 3, 4
range(2, 6)       # from 2 (inclusive) till 6 (exclusive):=> 2, 3, 4, 5
range(4, 10, 2)   # from 4 (inclusive) till 10 (exclusive) by 2:=> 4, 6, 8
range(4, 1, -1)   # from 4 (incl.) till 1 (excl.) by -1:=> 4, 3, 2
range(12, 4, -2) # from 12 (incl.) till 4 (excl.) by -2:=> 12, 10, 8, 6
```

3.3.2. for Loops

The basic forms of **for** statements are as follows:

```
# To repeat loop for each number generated by range() function:
for var in range(10):              # ends with colon ':'
    block of Python statements

for var in range(4, 10):           # ends with colon ':'
    block of Python statements

for var in range(2, 11, 2):        # ends with colon ':'
    block of Python statements
```

```
# To repeat loop for each character of a string
string = "..some string.."
for ch in string:
    block of Python statements         # eg.: print(ch)
```

```
# To repeat loop for each element of a list:
for var in list:               # ends with colon ':'
    block of Python statements
```

Examples:

```
# Prints out the numbers 0,1,2,3,4
for x in range(5):         # from 0 (inclusive) till 5 (exclusive)
    print(x)
```

```
# Prints out 3,4,5
for x in range(3, 6):      # from 3 (inclusive) till 6 (exclusive)
    print(x)
```

```
# Prints out 4, 6, 8
for x in range(4, 10, 2):      # from 4 till 10 (exclusive), 2 by 2
    print(x)
```

```
# Prints out 10, 8, 6
for x in range(10, 4, -2):     # from 10 till 4 (exclusive), by -2
    print(x)
```

```
print ("loop for elements of a list:")
primes = [2, 3, 5, 7, 11, 13, 17, 19]          # This is a list.
for prime in primes:
    print(prime)
```

Different solutions for the same problem:

Write a Python for loop that prints 4-digit positive numbers that ends with 77, in increasing order.
I.e. print 1077 1177 1277 .... 9877 9977

```
# Solution 1:
for n in range(1000, 10000):       # Generate all 4-digit positive numbers
    if n % 100 == 77:              # Check if last 2-digits is 77.
        print(n)
```

```
# Solution 2:
for n in range(1077, 10000, 100):      # Generate 1077 1177 1277 ... 9977
    print(n)
```

```
# Solution 3:
for k in range(10, 100):       # Generate left-most 2 digits: 10 11 ... 99
    n = k * 100 + 77
    print(n)                   # or: print(k * 100 + 77) without using n
```

Solution 2 is best, because it only generates the necessary numbers and doesn't perform any extra iteration or any extra computation.

Find the sum of numbers divisible by n from n1 to n2 (both inclusive). n1, n2 and n are integers.

```
# Solution 1:    Easy to think.
sum1 = 0
for k in range(n1, n2+1):      # Generate all numbers from n1 to n2
    if k % n == 0:             # Check if number is divisible by n
        sum1 += k                  # Add to sum
print(sum1)
```

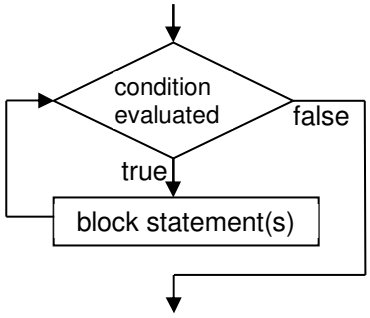
```
# Solution 2:    Have a much better performance
if n1 % n == 0:       # If n1 is divisible by n,
    k1 = n1           # then start number is n1
else:        # Find smallest number greater than n1 and divisible by n.
    k1 = n1 - n1 % n + n
sum2 = 0
for k in range(k1, n2+1, n):
    sum2 += k
print(sum2)
```

Nested for loops:

If there are loops inside other loops, then this structure is called "**nested loops**".
Nested for loop examples:

```
for n10 in range(10, 40, 10):      # Will produce values: 10 20 30
    for n1 in range(1, 4):         # Will produce values: 1 2 3
        print(n10 + n1)

for n10 in range(10, 40, 10):      # Will produce values: 10 20 30
    for n1 in range(1, 4):         # Will produce values: 1 2 3
        print(n10 + n1, end=" ")
    print()

for f in range(1,5):
    print("f=", f, "start")
    for n in range(1,4):
        print("n=", n)
    print("f=", f, "end")

print("Triangle of asterisks")
for r in range(1, 5):     # r: 1, 2, 3, 4
    for c in range(r):
        print("*", end="")
    print()
```

3.3.3. while Loops



Flowchart of while statement

The while loops are used for repeating tasks as long as a certain condition is true.

The basic form of **while** loop is as follows:

```
while condition:
    ....
    ....    while block (Python statements)
    ....
```

The while block is executed as long as the condition is true.

Examples:

```
# Sum of entered numbers until a non-positive number is entered.
sum1 = 0
n = int(input("Enter a number: "))
while n > 0:
    sum1 += n
    n = int(input("Enter a number: "))
print(sum1)
```

```
# while loop: prints out 0 1 2 3 4
count = 0
while count < 5:
    print(count)
    count += 1        # This is the same as count = count + 1
print("After while loop: count =", count)
```

```
# Print 10 9 8 7 6 ... 0
print("Countdown:")
seconds = 10
while seconds >= 0:
    print(seconds)
    seconds -= 1      # seconds = seconds - 1
```

```
# For a group of bacteria, with a given increase rate,
# find the time that will take to double the population.
time = 0
population = 1000         # 1000 bacteria to start with
growth_rate = 0.21        # 21% growth per hour
print("Bacteria population =", population)
while population < 2000:
    time += 1             # or:  time = time + 1
    population = int(population + growth_rate * population)
#   print("Hour:", time, " Population:",  population)
print("It took", time, "hours for the bacteria to double.")
print("After", time, "hours, there are", population , "bacteria.")
```

```
print("Fibonacci numbers < 100:")
fibo1 = 1
fibo2 = 1
print(fibo1, fibo2, end=" ")
fibo3 = fibo1 + fibo2
while fibo3 < 100:
    print(fibo3, end=" ")
    fibo1, fibo2 = fibo2, fibo3
    fibo3 = fibo1 + fibo2
```

3.3.4. break and continue Statements

break statement:

The break statement causes to terminate the loop before it has looped through all the items. The execution continues after the loop.

```
for n in range(1, 10):
    if n >= 5:
        break
    print(n)

k = 827419
while k > 0:
    if k % 2 == 0:
        break
    k //= 10  # k = k // 10
```

```
fruits = ["apple", "banana", "cherry"]
for x in fruits:
    print(x)
    if x == "banana":
        break
print (20 * "=")
for x in fruits:
    if x == "banana":
        break
    print(x)
```

continue statement:

The continue statement causes Python to skip immediately the rest of the current iteration (stop the current iteration), and continue with the next iteration.

```
for n in range(1, 10):
    if n >= 5 and n <= 7:
        continue          # continue with next iteration
    print(n)
```

```
for n in range(10, 41):
    if n % 2 == 0:        # if n is even
        continue          # then continue with next iteration
    print(n)
```

```
str = 'Oz48gur19luk'
total = 0             # The sum of the digits seen so far.
count = 0             # The number of digits seen so far.
for c in str:
    if not c.isdigit():   # If not digit,
        continue          # then continue with next character.
    count = count + 1
    total = total + int(c)
print("Number of digits =", count, "  Sum of digits =", total)
```

Exercises:

1. Write a program that inputs two integer numbers (n1, n2), finds and prints the sum of numbers between n1 and n2 (both inclusive).
2. Write a program that that inputs three integer numbers (n1, n2, n), prints the numbers divisible by n between n1 and n2 (both inclusive).
3. Write a program that that inputs two integer numbers (n1, n2), prints the numbers divisible by 10 between n1 and n2 (both inclusive).
4. Write a program that inputs an integer number and print the first digit of that number.
5. Write a program that inputs n integer numbers and print their average. Input first n.
6. Write a program that inputs integer numbers until 0 (zero) is entered and print their average.
7. Write a program that inputs positive integer numbers until a non-positive number is entered and prints the maximum number.

4. Functions

4.1. Basics of Functions

A function is a piece of program that takes zero or more parameters, and performs a well defined task.

Function Call:

The general form of a **function call** is as follows:

```
function_name(arguments)
```

Function Arguments:

Arguments are placed between parentheses and separated by a comma.
An argument is an expression that appears between the parentheses of a function call.

`int(7.8)`	`print(a, b+c)`	`abs(x)`	`abs(n - 7)`	`int(1.5**2)`	`round(a - b / c, 2)`

The rules to executing a function call:

1. Evaluate each argument one at a time, working from left to right.
2. Pass the resulting argument values into the function.
3. Execute the function.
4. When the function finishes, if it returns a value, then that value replaces the function call.

If a function returns a value, then its function call can be used in expressions:

```
aa = abs(-7 + 2) + abs(3.3 * -2) * 2        # 5 + 6.6 * 2 = 18.2
result = max(7, 12) * abs(-2)               # 12 * 2 = 24
for k in range(1, 40, 7):                   # => [1, 8, 15, 22, 29, 36]
```

We can also use function calls as arguments to other functions:

```
pow(abs(-2), round(4.3))           # pow(2, 4) => 16
n = int(input("Enter an integer number: "))
R = float(input("Enter a number: "))
a = int(5 + float("4.8") * 2)      # --> 14
```

4.2. Built-in Python Functions

Python comes with many **built-in functions** that perform common operations.

Built-in Python Functions:

abs()	delattr()	globals()	locals()	property()	str()
all()	dict()	hasattr()	long()	range()	sum()
any()	dir()	hash()	map()	raw_input()	super()
basestring()	divmod()	help()	max()	reduce()	tuple()
bin()	enumerate()	hex()	memoryview()	reload()	type()
bool()	eval()	id()	min()	repr()	unichr()
bytearray()	execfile()	input()	next()	reversed()	unicode()
callable()	file()	int()	object()	round()	vars()
chr()	filter()	isinstance()	oct()	set()	xrange()
classmethod()	float()	issubclass()	open()	setattr()	zip()
cmp()	format()	iter()	ord()	slice()	__import__()
compile()	frozenset()	len()	pow()	sorted()	
complex()	getattr()	list()	print()	staticmethod()	

Examples of function calls:

```
print(a, b, c)
int(7.84)              # convert to int      Result: 7
int("1919")            # convert to int      Result: 1919
```

```
float(4)              # convert to float    Result: 4.0
float("2.74")         # convert to float    Result: 2.74
int(float("2.74"))    # 1. convert to float, 2. convert to int; Result: 2
round(4.499999)       # Result: 4
round(4.5)            # Result: 5
round(3.141592653, 4)         # Result: 3.1416
abs(-9)                       # => 9
pow(abs(-2), round(4.3))      # pow(2, 4)  =>  16
k = pow(abs(-2), round(4.3)) + 4       # pow(2, 4) + 4 =>  20
```

4.3. Defining Functions

The built-in functions are useful but pretty generic. Often there aren't builtin functions that do what we want, such as calculate the average of a list of numbers or count upper case characters in a string. When we want functions to do these sorts of things, we have to write them ourselves.

Function Definition:

The general form of a **function definition** is as follows:

```
def function_name(parameters):          # Colon ':' is necessary.
    statement(s)                        # One or more equally indented statements
```

def	Keyword indicating function definition
«function_name»	Same naming rules as variables.
«parameters»	Zero or more parameters, comma-separated.

return Statement:

```
    return expression
```

Returned value replaces the function call.

Assume that a function named square is defined as:

```
def square(num):
    result = num * num
    return result
```

```
a = b * square(c+d) - e
```

In the above function call, argument (c+d) is evaluated first and becomes the value of the parameter of square() function. square() function returns some value (e.g. the square of its parameter), and the returned value replaces the function call in the expression: b * square(c+d) - e.

Local Variables:

Local variables are the variables used in functions.

They do not exist outside the function that they are used.

```
def quadratic(a, b, c, x):    # quadratic: returns (a * x ** 2 + b * x + c)
    first_term = a * x ** 2           # local variable first_term
    second_term = b * x               # local variable second_term
    third_term = c                    # local variable third_term
    return first_term + second_term + third_term

q = quadratic(1, 2, 3, 4)
print(a)                  # ERROR: a is defined in function, not here
print(first_term)         # ERROR: first_term is defined in function
```

Examples of function definitions and Python programs that uses defined functions:

```
def convert_to_celsius(fahrenheit):
    return round((fahrenheit - 32) / 1.8, 2)
```

```
def quadratic(a, b, c, x):
    return a * x ** 2 + b * x + c
```

```
def print_n(a, n): # print_n: Prints first argument multiple times.
    for k in range(n):
        print (a)
# MAIN PROGRAM.
print_n(44, 7)
print_n(3.14159, 3)
print_n("Bilim", 4)
```

```
# isPrime: Returns True if parameter is a prime number, False otherwise.
def isPrime(num):
     if num < 2:
          return False
     for j in range(2, num//2+1):# or: in range(2, round(num**0.5)+1)
          if num % j == 0:            # if j is divisor of n,
               return False           # then num is not a prime number
     return True
# MAIN PROGRAM.
a = int(input("Enter an integer number: "))
if isPrime(a):
     print(a, "is a prime number.")
else:
     print(a, "is NOT a prime number.")
# Find prime numbers from 100 to 200.
print("Prime numbers from 100 to 200 are:")
for n in range(100, 201):
     if isPrime(n):
          print(n , end = ' ')
```

Global Variables:

Variables that are created outside of a function (as in all of the examples in the previous pages) are known as global variables.

Global variables can be used anywhere, both inside of functions and outside.

```
x = "awesome"
def myfunc():
     print("Python is " + x) # global

myfunc()
```

```
Python is awesome
```

```
x = "awesome"
def myfunc():
     x = "fantastic"          # local
     print("Python is " + x)
myfunc()
print("Python is " + x)
```

```
Python is fantastic
Python is awesome
```

The global Keyword

Normally, when you create a variable inside a function, that variable is local, and can only be used inside that function. To create a global variable inside a function, you can use the global keyword.

```
def myfunc():
     global x                # now x is a global variable
     x = "fantastic"
myfunc()
print("Python is " + x)      # since x is a global variable, it is available here
```

Exercises:

1. Write a function that gets an integer as parameter and returns the factorial of the parameter.
2. Write a function that gets an integer as parameter and returns True if the parameter is a prime number, and returns False otherwise.
3. Write a function that gets two integers as parameters, and returns the sum of all integer numbers between these two numbers (both inclusive).
4. Write a function that gets three integers (say n1, n2, k) as parameters, and returns the sum of all integer numbers divisible by k from n1 to n2 (both inclusive).
5. Write a function that gets three integers (say n1, n2, k) as parameters, and returns the average of all integer numbers divisible by k from n1 to n2 (both inclusive).
6. Write a function that returns the list of prime factors of a given integer number.
7. Write a function that gets a string as parameter, and returns True if the alpha numeric characters forms a palindrome. Example: "Madam, I'm Adam!" is a palindrome.

   Hint: step 1: create a string that contains only alphanumeric characters; step 2: convert all to uppercase; step 3: check if it is symmetric.
8. Write a function that returns the number of occurences of a number in a list.
9. Write a function that returns the number of peak (deep) values in a list.

5. Strings

In **computer science**, a **string** is a finite sequence of characters (i.e., letters, numerals, symbols and punctuation marks, etc.). An important characteristic of each **string** is its length, which is the number of characters in it. The length can be any natural number (i.e., zero or any positive integer).

Each character is represented by a number using ASCII encoding (ASCII Code).

ASCII TABLE

Dec.	+0	+1	+2	+3	+4	+5	+6	+7	+8	+9	+10	+11	+12	+13	+14	+15
32	space	!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
48	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
64	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
80	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
96	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
112	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	

Some important ASCII codes: ' ' (space): 32 (20H), '0' : 48 (30H), 'A': 65 (41H), 'a': 97 (61H).

' ' (space) < '0' .... '9' < 'A' .... 'Z' < 'a' .... 'z'

Strings can be assigned and can be compared to other strings.

String Length: len() function returns the number of characters in a string.

```
len("Albert Einstein")    -> 15          len('123!')  -> 4
```

String Constants

String constants are delimited with either a single quote ('), double quote ("), triple single quote (''') or triple double quote (""""). The opening and closing quotes must match.

```
""       # Null string or empty string
"?"      "k"      "7"      "\n"         # Strings with a single character
'Antalya'         "Antalya is beautiful"
"""Jane's friend asked:\n"Are you alien?"."""           # \n : newline.
"René Descartes'          # ERROR: opening and closing quotes do not match
print('That\'s beautiful.\n')          # "That's beautiful.\n"
print("A single \\ back slash. Time is\nnot\t\"linear\".")
print(1, "\t", 2, "\n", 4, "\t", 8)
```

String Operators:

Operator	Description	Examples

+	Concatenation operator '+' concatenates two strings.	`"I love" + " nature!"`
		`"Good" + " morning" + "!"`
		`surname + ", " + name`

*	Repeats a string.	`"At" * 4  # "AtAtAtAt"` `10 * '*'  # "**********"`

Comparing Strings:

Case Sensitive Comparison: (ASCII codes are used)

"ZZZ"	<	"aaa"	"Abcd"	<	"Abcd "
"aaZZZZ"	<	"aaa"	"CS99"	>	"CS8888"
"KZBZHGV"	<	"KZa"	"CS9999"	<	"Cs88"
"abcd"	>	"ABCD"	"ABA"	>	"AB9999"

Case Insensitive Comparison: (Upper and lower case of same letter are treated as equal.)

"ZZZ"	>	"aaa"	"CS990a"	==	"Cs990A"
"KZZZH8"	>	"KZa"	"CS990"	>	"Cs8888"
"abcd"	==	"ABCD"	"CS990a "	>	"Cs990A"

Concatenation with other data types, and str() function:

```
"Antalya" + 'is' + "beautiful!"            # 'Antalyaisbeautiful!'
'Antalya' + 7             # ERROR: Can't convert 'int' object to str implicitly
9 + ' planets'            # ERROR: unsupported operand type(s) for +: 'int' and 'str'
"How far is a " + str(7) + " minute walk?"      # Number converted to string
"Use " + str(3.14159265359) + " as pi."         # Number converted to string
"Alan Turing" + ''                 # 'Alan Turing'
"" + 'John von Neumann'            # 'John von Neumann'
"At" * 4                           # "AtAtAtAt"
```

Get all characters in a string one-by-one:

```
s = "Antalya"
for c in s:
    print(c)
```

```
s = "Antalya"
for j in range(len(s)):
    print(s[j])
```

```
s = "Antalya"; count = 0
for c in s:
    if c == "A" or c == "a":
        count += 1
print ("There are", count, "A's.")
```

5.1. String Indexing

Any character in a string can be accessed using the index of that character.

Index values of characters in a string start from 0. The index value of last character of a string s is -1. Negative indexes are used to designate characters from the end of strings.

```
s = "Abcd 1234567 => Xyz"
#    0123456789012345678  Index values.
print(s[0])               # first (0'th) character of s      A
print(s[6])               # character with index 6 of s      1
print(s[len(s)-1])        # last character of s              z
print(s[-1])              # last character of s              z
print(s[-2])              # 2nd character from end of s      y
```

```
# Program that counts the number of adjacent same characters in a string.
s1 = 'Abba is good!'
count = 0            # counter for equal adjacent characters
for j in range(len(s1) - 1):
    if s1[j] == s1[j+1]:      # if j'th and (j+1)st chars are same, then
        count += 1            # increment counter
print("There are", count, " adjacent same characters.")
```

5.2. String Slicing

String slicing produces a new string that only consists of a portion of related string.

```
s[startIndex:stopIndex]
s[startIndex:stopIndex:increment]
```

In slicing, startIndex is included but the stopIndex is excluded.

If startIndex is not used, it will be treated as the start of string (0). Example: **`s[:8]`**

If stopIndex is not used, it will be treated as the end of string. Example: **`s[8:]`**

If the increment is negative, then startIndex must be greater than stopIndex. In this case, if startIndex is omitted then it will be treated as the end of string, and if stopIndex is omitted then it will be treated as the start of string.

Examples:

```
print("s[0:4] =>", s[0:4])     # first 4 chars of s (indexes 0 1 2 3)
print("s[:4] =>", s[:4])       # first 4 chars of s (indexes 0 1 2 3)
print("s[4:8] =>", s[4:8])     # 4th thru 7th characters of s
print("s[10:] =>", s[10:])     # 10th character till end of s
```

```
print("s[::2] =>", s[::2])     # chars with index 0, 2, 4, ... of s
print("s[4::3] =>", s[4::3])   # chars with index 4, 7, 10, ... of s
print("s[4:12:3] =>", s[4:12:3])   # chars with index 4, 7, 10 of s
print("s[::-1] =>", s[::-1])   # the reverse of s
print("s[::-2] =>", s[::-2])   # from the end of s every 2nd char
print("s[3::-1] =>", s[3::-1])# first 4 chars of s in reverse order
print("s[10:2:-3] =>", s[10:2:-3])   # chars with index 10, 7, 4 of s
```

5.3. String Methods

Methods:

Methods are like functions, but they are associated with a data type/structure or object.

Called by putting a period (.) after the data item then the method name and any parameters.

```
cnt = string1.count(string2)
```

In IDLE, to learn all methods of a certain data type using:

```
dir(<<type>>)                # E.g.  dir(str)
```

In IDLE, to learn about a method using:

```
help(<<type>>.<<method>>)
```

Example:

```
help(str.isupper)            # help about isupper string method
```

Method	Description	Return type
s.lower()	Returns a copy of the string with all letters converted to lowercase.	str
s.upper()	Returns a copy of the string with all letters converted to uppercase.	str
s.islower()	Returns True if all letter characters in the string are lowercase	bool
s.isupper()	Returns True if all letter characters in the string are uppercase	bool
s.isalpha()	Returns True if all characters of s are alphabetic (letter characters)	bool
s.isdigit()	Returns True if all characters of s are digits (numeric characters)	bool
s.isalnum()	Returns True if all characters of s are alphanumeric (letter or digit)	bool
s.isspace()	Returns True if all characters of s are whitespace characters	bool
s.startswith(s1)	Returns True if s starts with s1	bool
s.endswith(s1)	Returns True if s ends with s1	bool
s.swapcase()	Returns a copy of the string with all letters swapped the case.	str
s.count(s1)	Returns the number of non-overlapping occurrences of s1 in the string s	int
s.find(s1)	Returns the index of the first occurrence of s1 in the string s, or -1 if s1 doesn't occur. (case sensitive)	int
s.find(s1, begin)	Returns the index of the first occurrence of s1 at or after index begin in the string, or -1 if not found.	int
s.find(s1, begin, end)	Returns the index of the first occurrence of s1 between index begin (inclusive) and end (exclusive) in the string, or -1 if not found.	int
s.index(..,..,..)	Very similar to find() method. Returns the index of s1 in the string s if found; but it gives an error if s1 doesn't occur.	int
s.replace(f, r)	Returns a copy of the string with all occurrences of substring f replaced with string r	str
s.strip()	Returns a copy of the string with leading and trailing whitespace removed	str
s.lstrip()	Returns a copy of the string with leading whitespaces removed	str
s.rstrip()	Returns a copy of the string with trailing whitespaces removed	str
s.strip(s2)	Returns a copy of the string with leading and trailing characters in s2 removed. a = "aaaBolazaza"; x = a.strip("za") → "Bol"	str
s.split(separator, maxsplit)	Returns a list of strings after breaking the given string by the specified separator string. text.split() text.split(', ') text.split('\t')	list

Strings are immutable: String indexing, slicing and methods do not change the original string.

```
city = "Antalya"
city[3] = "o"              # ERROR: strings are immutable!
city.replace('a', 'b')             # does not change city!
city = city.replace('a', 'b')      # assignment changes string!
```

Chaining method calls are done in order from left to right:

```
city = "Antalya"
print(city.upper().replace('L', 'K'))
```

5.4. String Formatting

f"..." formatting:

A new string is created by placing the values of the variables between { } in the format string after the f symbol.

{ } Expression	Description	Examples

{variable}	The variable is placed in the result string as it is.	`a = 42; b = "Eda"`
		`print(f"zz{a}tt{b}ee")`
		`Output:  "zz42ttEdaee"`

{variable:width}	The variable is placed in the result string with the given width reserved for it.
	Numbers are right aligned, string left!
		`a = 42; b = "Eda"`
		`print(f"zz{a:4}tt{b:6}ee")`
		`Output:  "zz  42ttEda   ee"`

{variable:0width}	The variable is placed in the result string with the given width reserved for it and the extra spaces are replaced with 0.	`a = 42; b = 4.16`
		`print(f"zz{a:04}tt{b:07}ee")`
		`Output:  "zz0042tt0004.16ee"`

Examples:

`f"....{var1}....{var2}...."`	`....value1....value2....`
`f"....{var1:width}....{var2}...."`	`....  value1....value2....`
`a, b, c = "Antalya", 7, 1.618` `print(f"City: {a}  Num: {b}  GR: {c}")`	City: Antalya Num: 7 GR: 1.618
`a, b, c = "Antalya", 7, 1.618` `print(f"City: {a:12} Num: {b:03} GR: {c}")`	City: Antalya Num: 007 GR: 1.618

Format placeholders (format identifiers):

%d	int	`print("There are %d seconds in %d minutes" % (5*60, 5))` `print("There are %5d seconds in %3d minutes" % (5*60, 5))`
%f	float	`print("Radius: %f, Area: %f" % (4, 3.14159 * 4**2))` `print("Radius: %.1f, Area: %.2f" % (4, 3.14159 * 4**2))`

%s	str	`a = "Antalya"`
		`print("City is %s." % a)`
		`print("City is %-20s." % a)`

%c	character	`print("4th char is %c" % a[4])` `print("ASCII code = %d, character = %c" % (72, 72))`

`a, b, c = "Antalya", 7, 1.618`
`line1 = "%s;%d;%f\n" % (a, b, c)`
`line2 = "%-10s;%02d;%8.2f\n" % (a, b, c)`
`print ("City: %s, Plate no: %02d" % (a, b))`
	
	`'Antalya;7;1.618000\n'`
	`'Antalya   ;07;    1.62\n'`
	`City: Antalya, Plate no: 07`

```
# Formatted print and parallel lists.
names = ["Jane", "Paulo", "Juan Carlos", "Raquel"]
distance = [7, 34, 4287, 749]
prep = [284.2468, 23.4, -122.0, 0.1238]
print("Unformatted listing:")
for j in range(len(names)):
    print(names[j], distance[j], prep[j])
print("\nFormatted listing:")
for j in range(len(names)):
    print("%-16s %5d %10.2f" % (names[j], distance[j], prep[j]))
```

Output of the above program:

```
Unformatted listing:
Jane 7 284.2468
Paulo 34 23.4
Juan Carlos 4287 -122.0
Raquel 749 0.1238
```

```
Formatted listing:
Jane                7     284.25
Paulo              34      23.40
Juan Carlos      4287    -122.00
Raquel            749       0.12
```

format() method:

```
"...{}...{}...{}...{}...".format(a, b, c, d, ....)          # a b c d
"...{0}...{1}...{2}...{3}...".format(a, b, c, d, ....)      # a b c d
"...{0}...{2}...{1}...{2}...".format(a, b, c, ....)         # a c b c
"...{:d}...{:10s}...{:.2f}...{:04d}...".format(a, b, c, d)  # a b c d
```

format() method examples:	**Result**
`a, b, c = "Antalya", 7, 1.618` `'3 items {0}, {1} and {2}'.format(a, b, c)`	"3 items Antalya, 7 and 1.618"
`str = "{0}{1}{2}".format(8, "ab"+"cd", 4 * 7)`	"8abcd28"

`a = 7;  b = "x";  c = 222`
`str = "{0}{1}{2}".format(a, b, c)`
`str1 = "{0}++{1}++{2}".format(a, b, c)`
`str2 = "{}++{}++{}".format(a, b, c)`
	
	"7x222"
	"7++x++222"
	"7++x++222"

`# arguments' indices can be repeated` `'{0}{1}{0}'.format('abra', 'cad')`	 'abracadabra'

Exercices:

1. Write a function countChar() that gets a string and a character as parameters, and returns the number of occurences of that character in the string. Do not use count() method.
2. Write a function to determine if a given string contains a numeric type (either integer or decimal).
3. Count how many times a substring occurs in a string including overlapping occurences: (“banana”, “ana”) -> 2.
4. Write a function that tests if a given string is a palindrome.
5. Write a function countUp() that returns the number of upper case letters in a given string.
6. Write a function upLow() that returns a new string made by converting all upper case letters to lower and lower case letters to upper in a given string.
7. Write a function shuffle() that gets two strings as parameters, returns a new string made by shuffling as follows: 1st char from string 1, then 1st char from string 2, then 2nd char from string 1, then 2nd char from string 2, etc. Remaining chars of longer string are placed at end.

6. Lists

Lists contain zero or more objects and are used to keep collections of data.

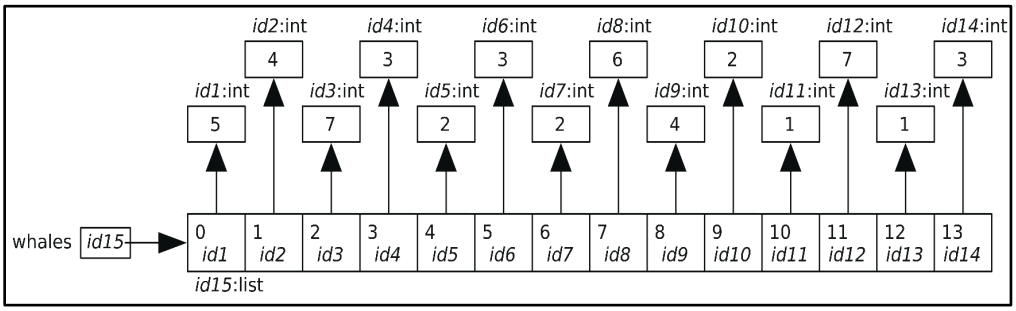
Creating Lists:

```
listName = []          # Create empty list
listName = list()      # Create empty list
listName = list(argument)      # Create list from argument
listName = [expression1, expression2, ... , expressionN]
```

```
monthDays = [31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31]
#   indexes:  0   1   2   3   4   5   6   7   8   9  10  11
#   indexes:                                      -3  -2  -1
```

Consider the following list:

```
whales = [5, 4, 7, 3, 2, 3, 2, 6, 4, 2, 1, 7, 1, 3]
```



Reference and Assignment: **IMPORTANT!**

```
a = [2, 4, 8, 16]
print(a)                                # -->  [2, 4, 8, 16]
b = a             # b is an alias to a
print(b)                                     [2, 4, 8, 16]
a.append(32)
print(b)                                # -->  [2, 4, 8, 16, 32]
b.append(64)
print(a)                                # -->  [2, 4, 8, 16, 32, 64]
```

In this example there is only one list, a and b both refer to this same list.

List Indexing: The index of list elements starts from 0 (zero). Negative indexes are used to refer to elements at the end of a list: [-1] for the last element, [-2] for one before the last, etc.

```
values = [4, 10, 3, -6, 8]
print(values[3])      # will print -6
for j in range(len(values)):
    print(j, values[j])
    values[j] += 1          # changes list
```

```
# Process each element with for loop:
values = [4, 10, 3, -6, 8]
for n in values:
    print(n, n ** 2)
    n += 1    # doesn't change list elements
```

```
list1 = [7, 4, 16, 3, 2, -6, 15, 8, 7, -11, 10, -2]
cnteven, toteven, cntodd, totodd = 0, 0, 0, 0
for n in list1:             # get to n each element one by one
    if n % 2 == 0:          # if n is even, then
        toteven += n            # add element to toteven
        cnteven += 1            # increment cnteven
    else:                   # n is not even, so n is odd, then
        totodd += n             # add element to totodd
        cntodd += 1             # increment cntodd
aveeven = toteven / cnteven
aveodd = totodd / cntodd
print("Number of even numbers=", cnteven, " Sum=", toteven, " Average=", aveeven)
print("Number of odd numbers=", cntodd, " Sum=", totodd, " Average =", aveodd)
```

Adding and Modifying Elements:

Adding value to a list is done by **append()** method: **`values.append(7)`** # appends 7 at end of list.

List elements can modified using indexing: `monthDays[1] = 29`

When `whales[8] = 12` is executed, the content of id9 will be changed to 12.

```
values = [4, 10, 7, -6, 8, 7]
values.append(2)
for j in range(len(values)):       # for all indexes of values list
    values[j] = values[j] * 2      # Double each value
print(values)         # [8, 20, 14, -12, 16, 14]
fruits = ["apple", "banana", "cherry", "dewberry", "eggfruit"]
fruits[0] = "apricot"          # element with index 0 is modified.
print(fruits)
```

Deleting Elements:

del construct is used to delete an element with a given index or slice.

```
del list1[2]                   # remove item with index 2
del list1[-1]                  # remove the last item.
del list1[2:5]                 # remove items with index 2, 3, 4
```

Functions:

Function	Description	Examples
len(L)	Returns the number of items in list L	n = len(list1)
list(seq)	Creates a list from elements of a range or a tuple.	a = list(range(10, 100, 2)) a = list((2, 3, 5, 7))
max(L)	Returns the maximum value in list L	nmax = max(list1)
min(L)	Returns the minimum value in list L	nmin = min(list1)
sum(L)	Returns the sum of the values in list L	sum1 = sum(list1)
sorted(L)	Returns a copy of list L where the items are ascendingly sorted. (This does not mutate L.)	list2 = sorted(list1)

List Operators:

+	produces a new list composed by the concatenation of its argument lists.	ls = ls1 + ls2
*****	produces a new list that consists of repeated elements of the multiplied list	ls = ls1 * 4

```
a = [2, 4, 8] + [16, 32, 64]  # -> [2, 4, 8, 16, 32, 64]
original = ['H', 'He', 'Li']
final = original + ['Be', 'B']     # A list can be added to another list
# final = original + 'Be'      # ERROR! An item cannot be added to a list.
h4 = [7, 2] * 4           # Repetition. -> [7, 2, 7, 2, 7, 2, 7, 2]
```

in operator: checks if an item exists in a list.

```
fruits = ["apple", "banana", "cherry", "dewberry", "eggfruit"]
a = "cherry"
if a in fruits:
    print("Yes,", a, "is in the fruits list")
else:
    print("No,", a, "is NOT in the fruits list")
```

List Slicing

Because lists are sequences, indexing and slicing work the same way for lists as they do for strings.
Slicing returns a new list. However lists can be modified through slicing.

```
list1 = [4, 10, 3, -6, 8, 4, 9, 7, 1, 6]
print(list1)
```

```
print("[0] [-1]\t", list1 [0], list1[-1])
print("[2:]\t", list1[2:])        # from index 2 to the end
print("[:4]\t", list1[:4])        # from start to index 4 (exclusive)
print("[2:4]\t", list1[2:4])      # from index 2 (inclusive) to index 4 (exclusive)
print("[-3:]\t", list1[-3:])      # the last 3 elements
print("[::2]\t", list1[::2])      # even indexed elements
print("[::-1]\t", list1[::-1])    # reverse of list

list1[2:5] = [10, 20]     # replaces elements indexed 2-3-4 with [10, 20]
list1[2:5] = []           # removes elements indexed 2-3-4
list1[2:5] = []           # removes elements indexed 2-3-4
list1[:4] = [12, 16]      # replaces first 4 elements with [12, 16]
list1[-3:] = [2, 7, 10, 8, 5] # replaces last 3 elements with 2, 7, 10, 8, 5
```

List Methods:

Method	Description
L.append(v)	Adds an element at the end of list L
L.insert(i, v)	Inserts value v at index i in list L, shifting subsequent items to make room
L.extend(seq)	Appends the items in seq to the end of L (seq can be a list)
L.count(v)	Returns the number of occurrences of v in list L
L.index(v)	Returns the index of the first occurrence of v in L - an error is raised if v doesn't occur in L.
L.index(v, beg)	Returns the index of the first occurrence of v at or after index beg in L - an error is raised if v doesn't occur in that part of L.
L.index(v, beg, end)	Returns the index of the first occurrence of v between indices beg (inclusive) and end (exclusive) in L; an error is raised if v doesn't occur in that part of L.
L.pop()	Removes and returns the last item of L (which must be nonempty)
L.pop(index)	Removes and returns item with given index of L (which must be nonempty)
L.remove(v)	Removes the first occurrence of value v from list L
L.copy()	Returns a copy of the list L
L.clear()	Removes all items from list L
L.reverse()	Reverses the order of the values in list L
L.sort()	Sorts the values in list L in ascending order
L.sort(reverse=True)	Sorts the values in list L in descending order

Examples:

```
lst = [72, 48, 96, 60, 77, 84, 41, 60, 80, 51]
lst.append(92)          # add to end
lst.insert(4, 57)       # insert to index 4
n = lst.count(60)       # number of 60's in lst
j = lst.index(60)       # index of first 60 in lst
k = lst.index(60,5)     # index of first 60 at or after index 5 in lst
lst.remove(60)          # removes first occurence of 60 in lst
x = lst.pop(6)          # Deletes element with index 6 from list and returns it
lst2 = lst.copy()       # create a copy of lst and assign it to lst2, lst and lst2 are two distinct lists
lst3 = [64, 86]         # create a list named lst3
lst.extend(lst3)        # add elements of lst3 at the end of lst
lst.reverse()           # Reverses the order of elements of lst
lst.sort()              # sort elements of lst in ascending order
lst.sort(reverse=True)    # sort elements of lst in descending order
del lst[5]              # removes lst[5]
del lst[-5:]            # removes the last 5 elements of lst
lst.clear()             # Removes all elements, i.e. it empties the list.
```

List Comprehension

List comprehensions provide a concise way to create lists. Common applications are to make new lists where each element is the result of some operations applied to each member of another sequence or iterable, or to create a subset of those elements that satisfy a certain condition.

A list comprehension consists of brackets containing an expression followed by a for clause, then zero or more for or if clauses. The result will be a new list resulting from evaluating the expression in the context of the for and if clauses which follow it.

```
lst = [<expression> for <var> in <iterable>]
# is identical to:
lst = []
for <var> in <iterable>:
    lst.append(<expression>)
```

```
lst = [<expression> for <var> in <iterable> if <condition>]
# is identical to:
lst = []
for <var> in <iterable>:
    if <condition>:
        lst.append(<expression>)
```

Examples:

```
squares = [x**2 for x in range(1, 11)]     # 1 4 9 16 25 36 49 64 81 100
a = [-4, -2, 0, 2, 4]
print(a)     # [-4, -2, 0, 2, 4]
# create a new list with the values doubled
b = [x*2 for x in a]
print(b)     # [-8, -4, 0, 4, 8]
# filter the list to exclude negative numbers
c = [x for x in a if x >= 0]
print(c)     # [0, 2, 4]
# apply a function to all the elements
d = [abs(x) for x in a]
print(d)     # [4, 2, 0, 2, 4]

# nested for:
a = [x*y for x in [1, 2] for y in [7, 8]]
print(a) # [7, 8, 14, 16]

# nested for:
a = [(x, y) for x in [1, 2] for y in [10, 20]]
print(a) # [(1, 10), (1, 20), (2, 10), (2, 20)]
# is equivalent to:
a = []
for x in [1, 2]:
    for y in [10, 20]:
            a.append((x, y))

a = [(x, y) for x in [1, 2, 3] for y in [3, 1, 4] if x != y]
print(a)     # [(1, 3), (1, 4), (2, 3), (2, 1), (2, 4), (3, 1), (3, 4)]
# is equivalent to:
a = []
for x in [1,2,3]:
    for y in [3,1,4]:
        if x != y:
            a.append((x, y))
print(a)     # [(1, 3), (1, 4), (2, 3), (2, 1), (2, 4), (3, 1), (3, 4)]
```

Lists of Lists (Nested Lists):

Lists can contain any type of data. That means that they can contain other lists.

The general form of a nested list:

```
nestedList = [ [list1], [list2], [list3], ...., [listN] ]
```

```
life = [['Canada', 76.5], ['United States', 75.5], ['Mexico', 72.0]]
print(life[2])
print(life[0][0])
print(life[0][1])
canada = life[0]
canada[1] = 80.0
print(life)
```

Matrices:

The general form of a matrix:

```
matrix = [[Row1], [Row2], [Row3], ...., [RowN]]
```

Examples

```
m1 = [[2, 4, 3, 7], [5, 1, 0, 6], [7, 9, 1, 3], [4, 8, 2, 3]]
m2 = [[6, 2, 0, 5], [4, 9, 1, 7], [8, 2, 3, 5], [7, 0, 5, 7]]
# Add two matrices:
m3 = m1.copy()        # Assign copy of m1 to m3.
for row in range(len(m1)):
    for col in range(len(m1[0])):
        m3[row][col] += m2[row][col]  # Add elements of m2 to m3
print(m3)

print("Matrix transpose:")
def transposeMatrix(m):
    tm = []
    for col in range(len(m[0])):
        trow = []
        for row in range(len(m)):
            trow.append(m[row][col])
        tm.append(trow)
    return tm

m = [[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]]   # 3 x 4 matrix
print(m)
m2 = transposeMatrix(m)
print(m2)
```

Important note: If you will write Python programs that involve matrices and vectors, using the NumPy library is strongly recommended. See: 11. NumPy Arrays

Exercises:

1. Write a function sumEvens() that gets as parameter a list of integers and returns the sum of even numbers in the list.
2. Write a function aveOdds() that gets as parameter a list of integers and returns the average of odd numbers in the list.
3. Write a function that returns a new list containing non-negative items of a given list.
4. Write a function that gets as parameter a list and two values (v1 and v2), returns a new list that contains the elements of the given list which are between v1 and v2 (both inclusive).
5. Write a function that gets as parameter a list and a value, returns a new list that contains the elements of list which has the same type as value.

7. Tuples

A **Tuple** is a collection of Python objects separated by commas which is ordered and unchangeable. Python tuple is similar to list except that the objects in tuple are **immutable** which means we cannot change the elements of a tuple once assigned.

Tuples are written using parentheses instead of brackets.

Tuples can be subscripted (indexed), sliced, and looped over like strings and lists.

A tuple with one element is not written as (x) but as (x,) (with a trailing comma).

The number of elements of a tuple can be obtained by calling **len** function: tlen = len(myTuple)

```
weekDays = ("monday", "tuesday", "wednesday", "thursday", "friday")
tp1 = ('physics', 'chemistry', 1997, 2000)
tp2 = "a", "b", "c", "d"           # parenthesis may be omitted
tp3 = (7,)                         # tuple with one element.
q = (7)                            # ATTENTION: this is NOT a tuple.
```

```
a, b = 7, 12
c = (a, b)                         # c is a tuple
```

Conversion to tuple:

```
tpl1 = tuple(range(8, 20, 4))      # Converts range to tuple: (8, 12, 16)
tpl2 = tuple(somelist)             # Converts a list to tuple
tpl3 = tuple("Python")             # ('P', 'y', 't', 'h', 'o', 'n')
```

Accessing Values in Tuples: indexing.

```
p10 = (2, 3, 5, 7, 11, 13, 17, 19, 23, 29)
print(p10[0], p10[7])
x = p10[3] * p10[6]
```

Tuples are immutable:

Tuples are immutable, which means that they can not be modified in any way.

- Their elements can not be changed. tup1[2] = 24 is not valid.
- They can not be extended. tup1.append(44) append function is undefined for tuples.
- Their elements can not be deleted.

However, new tuples can be created using existing tuples.

```
tup1 = (12, 34.56, 7, 89.01)
tup2 = ('abc', "d", "ef", "klm-opq", 'xyz')
# A new tuple can be created as follows:
tup1 = tup1 + tup2        # a new tuple is created, then assigned to tup1
```

Although tuples are immutable, the objects inside them can still be mutated:

```
life = (['Canada', 76.5], ['United States', 75.5], ['Mexico', 72.0])
life[0][1] = 80.0
print(life)
(['Canada', 80.0], ['United States', 75.5], ['Mexico', 72.0])
```

Tuple Operators:

+ operator produces a new tuple whose value is the concatenation of its arguments.

*** operator** produces a new tuple that consists of repeated elements of the multiplied tuple.

in operator checks if an item exists in a tuple.

```
tp3 = tp1 + tp2
tp4 = tp1 * 4
if x in tp1:      # True if value of x exists in tuple tp1
```

Slicing Tuples:

Slicing tuples is very similar to slicing lists and strings.

```
print(tup1[2:])           # elements with index 2 to end.
print(tup1[::-1])         # reverse of the tuple.
print(tup1[0:3])          # elements with index 0 to 2.
tup2 = tup1[:3:-1] + tup1[:4]      # new tuple
```

Tuple Methods:

T.count(v)	Returns the number of occurrences of v in tuple T
T.index(v)	Returns the index of the first occurrence of v in T - an error is raised if v doesn't occur in L.

8. Sets

A set is an **unordered** and **unindexed** collection with **no duplicate** elements.

Basic uses include membership testing and eliminating duplicate entries. Sets cannot have two items with the same value. If there are duplicate values, they will be ignored.

Curly braces or the set() function can be used to create sets.

Note: to create an empty set you have to use set(), not {}. Once a set is created, its items cannot be changed, but new items can be added to a set, or items can be removed.

```
set1 = set()          # Create an empty set
set2 = set(list1)     # Create a set from elements of a list
set3 = set(range(4,9))    # Create a set from values generated by range
set4 = {"abc", 32, True, 4.8, "city"}      # A set with different data types
set5 = {"apple", "banana", "cherry", "apple"}   # Duplicates are ignored
print(set5)       # will print: {'banana', 'cherry', 'apple'}
```

len() function returns the length of a set, i.e. the number of elements in a set.

```
n = len(set1)
```

Set Methods:

Set objects support mathematical operations like union, intersection, difference, and symmetric difference.

Method	Description
s.add(e)	Adds an element to the set.
s.remove(e)	Removes the specified element (returns None).
s.discard(e)	Removes the specified element (returns None).
s.intersection(s2, ...)	Returns a set, that is the intersection of two or more sets.
s.union(s2, ...)	Return a set containing the union of sets. Set s is not modified.
s.difference(s2, ...)	Returns a set containing the difference between two or more sets.
s.symmetric_difference(s2)	Returns a set with the symmetric differences of two sets.
s.copy()	Returns a copy of the set.
s.update(s2, ...)	Update the set with the union of this set and others.
s.isdisjoint(s2)	Returns True if intersection is empty (i.e. no common elements).
s.issubset(s2)	Returns True if this set is a subset of another set.
s.issuperset(s2)	Returns True if this set is a superset of another set.
s.clear()	Removes all the elements from the set.
s.difference_update(s2)	Removes the items in this set that are also included in another set.
s.intersection_update(s2, ...)	Removes the items in this set that are not present in other set(s).
s.pop()	Removes randomly an element from the set and returns it.

Examples:

```
set1 = {2, 4, 8, 16}
set1.add(32)
s1 = {"apple", "banana", "cherry"}
s2 = {"banana", "apricot", "carrot"}
s1.update(s2)         # Add elements of s2 to s1 (duplicates ignored)
s1.remove("carrot")       # Remove "carrot" from s1
s4 = s1.union(s2)         # union of s1 and s2
s5 = s1.intersection(s2)  # intersection of s1 and s2
if s1.issubset(s2):       # True if all elements of s1 exist in s2
if "banana" in s1:        # True if "banana" exists in s1
```

9. Dictionaries

A **dictionary** is a collection of **key-value pairs**.

All keys in a dictionary must be unique.

The key & value pairs are listed between curly brackets "{ }".

In a dictionary, a key and its value are separated by a colon. The key + value pairs are separated with commas.

Keys can be of any singular data type (int, float, string, boole).

Values can be of any data type or collection (int, float, string, boole, complex, list, tuple, dictionary, ...).

```
{key1:value1, key2:value2, key3:value3, ...., keyN:valueN}
d = {}                  # Create an empty dictionary.
d = dict()              # Create an empty dictionary.
d[key1] = value1        # If key1 does not exist, adds new key + value pair.
d[key1] = value1        # If key1 exists, then value of d[key1] is changed.
```

Functions:

Function	Description
len(D)	Returns the number of key+value pairs (elements) in dictionary D.
max(D)	Returns the **maximum key** in dictionary D if all keys are numeric or string. Error otherwise.
min(D)	Returns the **minimum key** in dictionary D if all keys are numeric or string. Error otherwise.
sum(D)	Returns the **sum of the keys** in dictionary D if all keys are numeric. Error otherwise.
sorted(D)	Returns a list containing **the keys** of D where the keys are in order from smallest to largest, if all keys are numeric or string.

Dictionary Methods:

Method	Description
D.keys()	Returns an immutable object containing all the keys of D.
D.values()	Returns an immutable object containing all the values of D.
D.copy()	Returns a copy of dictionary D.
D.update(d)	Adds element(s) of d to the dictionary D if the key is not in the dictionary. If the key is in the dictionary D, it updates the element with the new value.

D.get(key, value)	If key exists in D then returns D[key], otherwise returns value.
	`a = dd.get(k, x)  # is identical to:`
	`if k in dd:  # if k is a key in dd`
	`  a = dd[k]  # then get the value of that element`
	`else:  # otherwise (no key is equal to k)`
	`  a = x  # get 2nd argument`

D.items()	Can be used to iterate a dictionary to get both key and value pairs.
	`for key, value in myDict.items():`
	`  print (key, value)`

D.popitem()	Removes the last key-value pair, returns (key,value) tuple.
D.pop(key)	Removes the element with the specified key, returns value of removed element.
del D[key]	Removes element with the given key
D.clear()	Removes all elements

in operator: The existence of a key in a dictionary is checked using the in operator.

```
if x in dd:             # if x exists as a key in dd
if x not in dd:         # if x doesn't exist as a key in dd
dd = {..............}
if k in dd:             # if k exists as a key in dd,
    dd[k] = ....            # then change the value of element with key k.
else:                   # if k doesn't exists as a key in dd,
    dd[k] = ....            # then add a new element with key k.
```

Dictionary Comprehension

Comprehension can be used to create dictionaries.

```
dct = [<key_expression>:<value_expression> for <var> in <iterable>]
# is identical to:
dct = {}
for <var> in <iterable>:
    dct[<key_expression>] = <value_expression>
```

```
dct = [<key_expression>:<value_expression> for <var> in <iterable> if <condition>]
# is identical to:
dct = {}
for <var> in <iterable>:
    if <condition>:
        dct[<key_expression>] = <value_expression>
```

dct = {n:n*n for n in range(1, 5)} will create: {1: 1, 2: 4, 3: 9, 4: 16}

dct = {n:n**3 for n in range(1, 10) if n%2 == 0} will create: {2: 8, 4: 64, 6: 216, 8: 512}

Examples:

```
monthDict={1:'Jan', 2:'Feb', 3:'Mar', 4:'Apr', 5:'May', 6:'Jun',
    7:'Jul', 8:'Aug', 9:'Sep', 10:'Oct', 11:'Nov', 12:'Dec'}
for key in monthDict:
    print(key, monthDict[key])
mdkeys = monthDict.keys()
print(mdkeys)
mdvals = monthDict.values()
print(mdvals)
```

```
# Alternative:
for key, value in monthDict.items():
    print(key, value)
```

```
monthDay={'Jan':31, 'Feb':28, 'Mar':31, 'Apr':30, 'May':31, 'Jun':30,
    'Jul':31, 'Aug':31, 'Sep':30, 'Oct':31, 'Nov':30, 'Dec':31}
for mon in monthDay:                    |    for mon, day in monthDay.items():
    print(mon, monthDay[mon])           |        print(mon, day)
```

```
months={1:['Jan', 31], 2:['Feb', 28], 3:['Mar', 31], 4:['Apr', 30],
    5:['May', 31], 6:['Jun', 30], 7:['Jul', 31], 8:['Aug', 31],
    9:['Sep', 30], 10:['Oct', 31], 11:['Nov', 30], 12:['Dec', 31]}
for j in months:                                |    for k, v in months.items():
    print(j, months[j])                         |        print(k, v)
for j in months:                                |    for k, v in months.items():
    print(j, months[j][0], months[j][1])  |        print(k, v[0], v[1])
```

```
a1 = {'a': 100, 'b': 200}
a2 = {'x': 300, 'y': 200}
newd = mergeDict(a1, a2)
print(newd)
```

```
def mergeDict(d1, d2):
    d = d1.copy()
    d.update(d2)
    return d
```

Dictionaries as counters:

Dictionaries can be used as counters of occurences of values.

```
# Use dictionary as counter:
votes = ["Kemal", "Raquel", "Ali", "Raquel", "Raquel", "Jane", "Jane",
    "Kemal", "Jane", "Kemal", "Raquel", "Jane", "Kemal", "Raquel", "Ali"]
counts = {}                         # Create an empty dictionary
for name in votes:
    if name in counts:      # if name exists as key in dict. counts,
        counts[name] += 1       # then increment that counter,
    else:                       # if name doesn't exist as key in counts,
        counts[name] = 1        # then add new element to counts.
print(counts)               # print the dictionary
for name in counts:
    print("%-12s :%3d" % (name, counts[name]))      # print nicely
# First for loop can be changed with the following:
cnt = {}      # Create an empty dictionary
for name in votes:
    cnt[name] = cnt.get(name, 0) + 1      # Add or update elements.
print(cnt)          # print the dictionary
```

10. File Input and Output

So far, we've only been able to get data into our programs from the user (using keyboard) with input() function. Also, we've only output data to the display using the print() function.

If there are large amounts of data, it can not be possible to get from user and to output to the monitor.

Large amounts of data can be read from files and can be written to files.

There are various types of files, some file types are: plain text files (.txt, .py, .c, .java, .html, .xml, etc.), various document files (.pdf, .docx, .xlsx, .pptx, etc.), executable program files (.exe, .dll, etc.), picture files (.jpg, .gif, etc.), audio files (.mp3, .wav, etc.), video files (.avi, .mp4, etc.), etc.

In this notes, we will explain only how to read from and write to **plain text files**.

Opening Files:

In order to read from or to write to a file, that file must be opened.

```
file = open(filename, mode)         # This is the general form.
```

```
file = open("data.txt", "r")        # Open file data.txt for reading
file = open("data2.txt", "w")       # Open file data2.txt for writing
file = open("data3.txt", "a")       # Open file data3.txt for appending
```

File modes:

"r"	Read mode. The file must exist.
"w"	Write mode. If the file exists then the file will be overwritten. I.e. the previous contents of the file will be destroyed, and the file will be re-created.
"a"	Append to the end of the file. Writes will write to the end of the file.

Closing Files:

When a file will not be processed anymore, then it must be closed.

```
file.close()
```

File Methods:

Method	Description
F.read()	Reads all contents of the file and returns as a string.
F.read(n)	Reads n characters from the file and returns as a string.
F.readlines()	Reads all lines of the file and returns the lines as a list.
F.readline()	Reads one line from the file and returns as a string.
F.write(string)	Writes a string to file. Argument must be a string.
F.writelines(list)	Writes a list of strings to the file by concatenating all.
F.close()	Closes the file.

Examples:

```
file = open("data.txt", "r")        # Open file data.txt
contents = file.read()              # Store all content into a string
file.close()                        # Close file
upcount = 0                         # Counter
for c in contents:                  # for each character of the string
    if c.isupper():                 # if character is uppercase
        upcount += 1                # then increment the counter
print(upcount)                      # print counter
```

```
fp = open("data.txt", "r")          # Open file data.txt
lines = fp.readlines()              # Read all lines into a list
fp.close()                          # Close file
for line in lines:                  # for each line in the list
    print(line.rstrip())            # right strip whitespaces and then print
```

```
file = open("data.txt", "r")   # Open file data.txt
for line in file:              # Read each line including \n at end of line
    print(line.rstrip())       # rstrip whitespaces and then print.
file.close()                   # Close file
```

```
file = open ("data.txt", "r")
contents = file.read()          # Store all content into a string
file.close()                    # Close file
strnums = contents.split()      # Split contents by whitespaces
total = 0
for s in strnums:               # for each element of list of strings
    total += float(s)           # convert element to number, add to total
print(total)                    # print total
```

The with Statement

Because every call on function open() should have a corresponding call on method close(), Python provides a with statement that **automatically closes a file** when the end of the block is reached.

The general form of a with statement is as follows:

```
with open(«filename», «mode») as «fileVariable»:
    «block»                     # After the block, the file is automatically closed.
```

Examples of **with** statement:

```
with open('file_example.txt', 'r') as file:
    contents = file.read()          # After this statement, the file is automatically closed.
print(contents)
```

```
with open('lorem_ipsum.txt', 'r') as file1:
    first_ten_chars = file1.read(10)       # Reads first 10 characters
    the_rest = file1.read()                # Reads until end-of-file
print("The first 10 characters:", first_ten_chars)
print("The rest of the file:", the_rest)
```

```
with open('courses.txt', 'r') as file2:
    lines = file2.readlines()     # Returns lines in a list
print(lines)
# The screen output will be similar to:  ['First line\n', 'Second line\n', 'Third line\n', ...]
```

```
with open('planets.txt', 'r') as data_file:
    for line in data_file:
        print(line.strip())
```

Writing to Files:

F.write() method can be used to write to text files.

```
# Create new file or if file exists then overwrite the file:
file1 = open('topics.txt', 'w')
file1.write('Computer Science')
file1.close()
```

```
# Append to end of file:
with open('topics.txt', 'a') as output_file:
    output_file.write('Software Engineering')
```

```
with open("numbers.txt", 'w') as output_file:
    a, b, c = 88, 7, 1.618
    line = '|%4d|%2d|%8.4f|\n' % (a, b, c)
    output_file.write(line)
```

```
lst = ["Aa", "777", "x"]
f1 = open("aa.txt", "w")
f1.writelines(lst)
f1.close()
```

File content will be:

Aa777x

Exercises:

1. Write a function **dispfile()** that takes a file name as parameter, reads all lines from this file and displays them on screen and returns the number of lines displayed.
2. A file contains only integer and float numbers. Write a function that takes a file name as parameter, reads all numbers in the file, returns the sum of the numbers.
3. Write a function **writeNumbers()** that takes a file name as parameter, write to the file each on a different line, the numbers from 100 to 1000 (100 200 ... 1000) incremented by 100.
4. The lines of a file contains only letters, digits, spaces, dot ('.') and comma (',') characters. Write a function **findNums()** that takes a file name as parameter, reads the contents of this file, replaces dots and commas with space, finds and prints the words that consists only of digits.

11. NumPy Arrays

Installing NumPy: Run `pip install numpy` command from command prompt.

NumPy (**Numerical Python**) is an open source Python library consisting of multidimensional array objects and a collection of routines for processing those arrays. NumPy was created in 2005 by Travis Oliphant. It is an open source project and can be used freely.

Using NumPy, mathematical and logical operations on arrays can be easily performed.

- NumPy arrays (called **ndarray**) have a fixed size at creation, unlike Python lists (which can grow dynamically). Changing the size of an ndarray will create a new array and delete the original.
- The elements in a NumPy array are all required to be of the **same data type**, and thus will be the same size in memory. The exception: one can have arrays of (Python, including NumPy) objects, thereby allowing for arrays of different sized elements.
- NumPy arrays facilitate advanced mathematical and other types of operations on large numbers of data.

NumPy Data Types:

Each element in **ndarray** is an object of data-type object (called **dtype**).

Data types are: bool, int, int8, int16, int32, int64, uint8, uint16, uint32, uint64, float, float16, float32, float64, complex, complex64, complex128

NumPy is usually imported under the **np** alias: `import numpy as np`

NumPy Methods:

Method	Description
array()	Constructor. Can be used to convert a list or tuple to NumPy array (ndarray).
arange()	Similar to Python range() function. Accepts int and float parameters.
reshape()	Reshape a NumPy array. Used to create multi dimensional arrays.
zeros(n) zeros((r, c))	Array of zeros (0.0). (Can be one or more dimensional.)
ones(n) ones((r, c))	Array of 1.0's. (Can be one or more dimensional.)
full(n, v) full((r, c), v)	Array of v values. (Can be one or more dimensional.)
identity(n) or eye(n)	Identity matrix of n x n dimensions
transpose()	Returns a new matrix that is transpose of a matrix
dot()	Returns result of matrix multiplication
tile(arr, n)	Tile array elements. **np.tile(np.arange(1, 5), 2) --> [1 2 3 4 1 2 3 4]**
repeat(arr, n)	Repeat array elements. **np.repeat(np.arange(1, 5), 2) --> [1 1 2 2 3 3 4 4]**
append(arr, arr2, axis=None)	Returns a new array by appending values (or another array) to the end of an array (at the given axis). Example for 1D arrays: ar3 = np.append(ar1, ar2)
insert(arr, idx, arr2, axis=None)	Returns a new array by inserting values (or another array) at the given index to an array, along the given axis. Example (1D): ar3 = np.insert(ar1, 4, ar2)
delete(arr, obj, axis=None)	Return a new array with sub-arrays along an axis deleted. For a 1-D array, this returns those entries not returned by *arr[obj]*.
sum(axis=None)	Returns sum of values (along the given axis)
max(axis=None)	Returns maximum of values (along the given axis)
min(axis=None)	Returns minimum of values (along the given axis)
copy()	Returns a copy of the array.
random.randint(n, size=())	Returns an array containing random numbers from 0 to (n-1). a = np.random.randint(100, size=(3, 4)) # 3x4 matrix with values 0-99. b = np.random.randint(40, 101, size=(3, 4)) # 3x4 matrix with values 40-100
random.choice(iterable, size=())	Returns an array containing randomly selected numbers from the iterable. x = np.random.choice(range(10, 101, 10), size=(4, 5)) # This gives 4x5 matrix consisting of randomly selected numbers from 10, 20, ..., 100
random.rand(n)	Returns an array of n random numbers from 0.0 to 1.0 (1.0 excluded).

NumPy Array (ndarray) Attributes:

Attribute	Description	Usage Example	Sample Results	
shape	Array shape (tuple with # of rows, columns, ...)	arr.shape arr.shape[0]	(20,)	(4, 8)
ndim	Array dimensions	arr.ndim	1	2
size	Array size (total number of elements)	arr.size	20	32
dtype	Data type of array elements	arr.dtype	int32	float64

Arithmetic Operations

Arithmetic operations on NumPy arrays perform that operation on every array value. When an array is added/subtracted/multiplied/divided/modulus by a number, that operation is performed on every element of the array and a new array of same size is created.

E.g.: If m is a NumPy array (1D or 2D), m*n will give a new NumPy array of same size where all values are multiplied by n.

Same way, when two arrays have an arithmetic operator between them, that operation is performed between elements with same index. The arrays must have same size and shape.

```
m1 = np.arange(....).reshape(3, 4)
m2 = np.arange(....).reshape(3, 4)
m3 = m1 * m2        # elements are multiplied
```

NumPy has a rich collection of mathematical functions:

https://numpy.org/doc/2.3/reference/routines.math.html

NumPy Array Indexing and Slicing

Slicing NumPy arrays have similar rules as slicing lists or strings.
There is multi-dimensional indexing and slicing with 2+D NumPy arrays.

A single element:	m[rowindex, colindex]
A row:	m[rowindex]
A column:	m[: , colindex]
Multiple rows:	m[startRow:stopRow:rowIncrement]
Multiple rows + cols:	m[startRow:stopRow, startCol:stopCol] m[startRow:stopRow:rowIncrement, startCol:stopCol:colIncrement]

```
[[ 1  2  3  4  5]
 [ 6  7  8  9 10]
 [11 12 13 14 15]
 [16 17 18 19 20]]
m[1:3] :
[[ 6  7  8  9 10]
 [11 12 13 14 15]]
m[1:3, 2:4] :
[[ 8  9]
 [13 14]]
```

Examples:

`ls = [42, 7, 84, 5, 12, 24]` `a = np.array(ls)`	`[42, 7, 84, 5, 12, 24]` `[42  7 84  5 12 24]`
`a = np.array(ls).reshape(2, 3)`	`[[42  7 84]` ` [ 5 12 24]]`

`a = np.arange(1, 9)`
`a = np.arange(1.0, 9)  # default increment = 1`
`a = np.arange(10, 21, 2)`
`a = np.arange(1.2, 2.5, 0.2)`
	`[1 2 3 4 5 6 7 8]`
	`[1. 2. 3. 4. 5. 6. 7. 8.]`
	`[10 12 14 16 18 20]`
	`[1.2 1.4 1.6 1.8 2.  2.2 2.4]`

`a = np.arange(1, 13).reshape(3, 4)`	`[[ 1  2  3  4]`
	` [ 5  6  7  8]`
	` [ 9 10 11 12]]`

`a = np.arange(1, 13).reshape(3, 4) * 10`
 	
`b = a[1]  # Indexing`
`c = a[1:, 2:]  # Slicing`
	`[[ 10  20  30  40]`
	` [ 50  60  70  80]`
	` [ 90 100 110 120]]`
	`[ 50  60  70  80]`
	`[[ 70  80]`
	` [110 120]]`

`a = np.arange(10, 130, 10).reshape(3, 4)`	`[[ 10  20  30  40]`
	` [ 50  60  70  80]`
	` [ 90 100 110 120]]`

`a = np.arange(1, 13).reshape(4, 3).transpose()`	`[[ 1  4  7 10]`
	` [ 2  5  8 11]`
	` [ 3  6  9 12]]`

`a = np.eye(3)`	`[[1. 0. 0.]`
	` [0. 1. 0.]`
	` [0. 0. 1.]]`

`a = np.eye(2, dtype=int)`	`[[1 0]` ` [0 1]]`
`a = np.ones(4, dtype=int)`	`[1 1 1 1]`
`a = np.ones((2,3))`	`[[1. 1. 1.]` ` [1. 1. 1.]]`
`a = np.full((2,3), 7)  # or:` `a = np.ones((2,3), dtype=int) * 7`	`[[7 7 7]` ` [7 7 7]]`

a1 = np.arange(1, 5) a2 = np.arange(24, 19, -2) a3 = np.append(a1, a2) print(a3) a4 = np.insert(a1, 1, a2) print(a4)	[1 2 3 4] [24 22 20] [1 2 3 4 24 22 20] [1 24 22 20 2 3 4]

a = np.arange(1, 10).reshape(3, 3).transpose()
 	
# Flatten NumPy matrix: 	
b = a.reshape(a.size)
	[[1 4 7]
	[2 5 8]
	[3 6 9]]
	[1 4 7 2 5 8 3 6 9]

# Sum, Max, Min: 	
m1 = np.arange(1, 13).reshape(3, 4)
print(m1)
 	
sm = m1.sum()
print("m1.sum() =", sm)
mx = m1.max()
print("m1.max() =", mx)
mn = m1.min()
print("m1.min() =", mn)
# Sum, max, min of columns: 	
sumcols = m1.sum(0) # or: m1.sum(axis=0)
print("m1.sum(0) =", sumcols)
mxcols = m1.max(0) # or: m1.max(axis=0)
print("m1.max(0) =", mxcols)
mncols = m1.min(0) # or: m1.min(axis=0)
print("m1.min(0) =", mncols)
# Sum, max, min of rows: 	
sumrows = m1.sum(1) # or: m1.sum(axis=1)
print("m1.sum(1) =", sumrows)
mxrows = m1.max(1) # or: m1.max(axis=1)
print("m1.max(1) =", mxrows)
mnrows = m1.min(1) # or: m1.min(axis=1)
print("m1.min(1) =", mnrows)
	[[1 2 3 4]
	[5 6 7 8]
	[9 10 11 12]]
	
	m1.sum() = 78
	
	m1.max() = 12
	
	m1.min() = 1
	
	m1.sum(0) = [15 18 21 24]
	
	m1.max(0) = [9 10 11 12]
	
	m1.min(0) = [1 2 3 4]
	
	m1.sum(1) = [10 26 42]
	
	m1.max(1) = [4 8 12]
	
	m1.min(1) = [1 5 9]

a = np.random.randint(20, size=(3, 4)) + 1 print(a) x = np.random.choice(range(10, 101, 10), size=(4, 5)) print(x) r = np.random.rand(6).reshape(3, 2) print(r)	[[16 19 17 13] [20 10 3 8] [9 17 5 17]] [[40 30 50 100 40] [60 90 100 80 90] [10 100 80 50 60] [40 100 60 20 50]] [[0.87066238 0.74545205] [0.4120916 0.06714363] [0.13951804 0.55617267]]

```
import numpy as np
m1 = np.arange(1, 7).reshape(2, 3)    # matrix 2x3: [[1 2 3] [4 5 6]]
m2 = m1.reshape(3, 2)              # matrix 3x2: [[1 2] [3 4] [5 6]]
m3 = np.array([[1,2,3],[4,5,6]]) # convert nested list to numpy matrix
zz = np.zeros((2, 3), dtype=int) # 2 x 3 matrix of 0's
a1 = np.ones((3, 4))                    # 3 x 4 matrix of 1.0's
a1 = np.ones((3, 4), dtype=int)  # 3 x 4 matrix of 1's
im = np.identity(4)          # 4 x 4 identity matrix
a1 = a1 + 7         # add a number to every element of array
m4 = 4 * m2         # multiply every element of array by a number
m4 = m2 ** 2        # square of every element of array
m7 = m5 + m6        # element-by-element addition (equal dimensions)
m7 = m5 * m6        # element-by-element multiplication (equal dimensions)
m7 = m5 // m6 # element-by-element floor division (equal dimensions)
m7 = m5 % m6        # element-by-element modulus (equal dimensions)
mc = ma.dot(mb)          # matrix multiplication of ma and mb
md = ma.transpose()      # Transpose of a matrix
```

References: https://numpy.org/doc/ https://www.tutorialspoint.com/numpy/ https://www.w3schools.com/python/numpy/

12. 2D Graphics in Python with Matplotlib

Matplotlib is a low level graph plotting library in Python that serves as a visualization utility.
Matplotlib was created by John D. Hunter.

Installation of Matplotlib

If you have Python and PIP already installed on a system, then installation of **Matplotlib** is very easy. Install it using this command in command line under Windows cmd:

```
pip install matplotlib
```

If this command fails, then use a python distribution that already has Matplotlib installed, like Anaconda, Spyder etc.

Installation of pip: Examine https://pip.pypa.io/en/stable/installation/ for installation of pip.

Import Matplotlib

Once Matplotlib is installed, import it in your applications by adding the import module statement:

```
import matplotlib
```

Matplotlib Pyplot

Most of the Matplotlib utilities lies under the **pyplot** submodule, and are usually imported under the plt alias:

```
import matplotlib.pyplot as plt
```

Now the Pyplot package can be referred to as plt.

Draw a line:

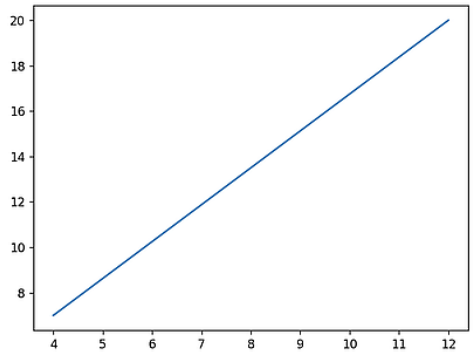
To draw a line, **plot()** method is used with two lists or NumPy arrays as parameters, the first contains the start and end x values, and the second contains the start and end y values.

Draw a line in a diagram from x-y coordinate (4, 7) to x-y coordinate (12, 20):

```
import matplotlib.pyplot as plt

xpoints = [4, 12]
ypoints = [7, 20]
plt.plot(xpoints, ypoints)
plt.show()
```

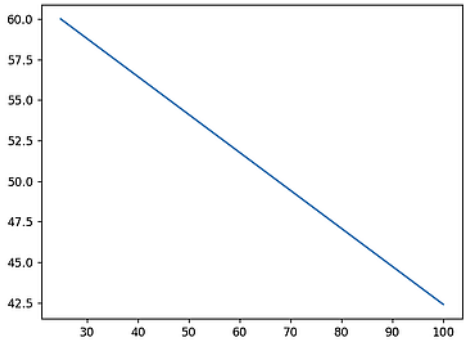
```
# or:
```

```
import matplotlib.pyplot as plt

plt.plot([4, 12], [7, 20])
plt.show()        # Show graph window
```



Draw a line from position (24.8, 60) to position (100, 42.4):

```
# Draw a line
import matplotlib.pyplot as plt

xpoints = [24.8, 100]
ypoints = [60, 42.4]
plt.plot(xpoints, ypoints)
plt.show()
```

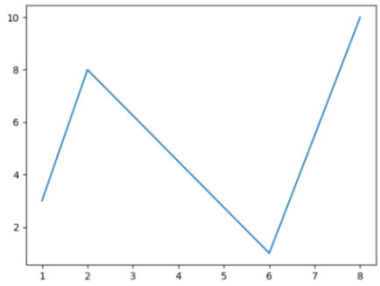


Draw Line Among Multiple Points:

You can plot as many points as you like, just make sure there are the same number of points in both x-axis and y-axis.

Draw a line in a diagram from coordinate (1, 3) to (2, 8) to (6, 1) and finally to (8, 10):

```
import matplotlib.pyplot as plt

xpoints = [1, 2, 6, 8]
ypoints = [3, 8, 1, 10]

plt.plot(xpoints, ypoints)
plt.show()        # Display graph window
```

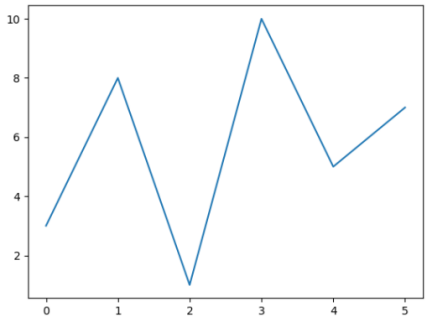


Draw Multiple Points with default X-Points:

If we do not specify the points in the x-axis, they will get the default values 0, 1, 2, 3, etc. depending on the length of the y-points.

So, if x-points are omitted in plt.plot() method call, then x points will be 0, 1, 2, ... and the diagram will look like this:

```
import matplotlib.pyplot as plt

ypoints = [3, 8, 1, 10, 5, 7]

plt.plot(ypoints)
plt.show()
```



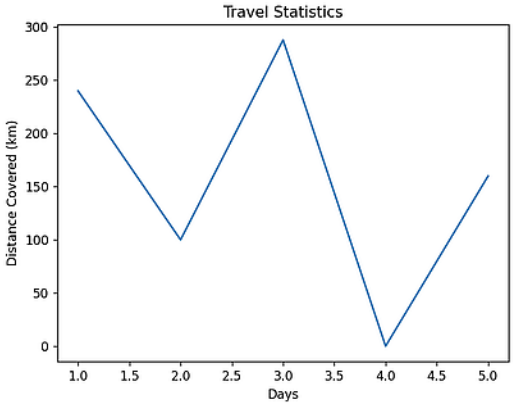
Title, Labels, Markers, Formatting, Grid and Legend

Title and Labels in a Plot:

With Pyplot, **title()**, **xlabel()** and **ylabel()** functions are used to set title and labels for the x- and y-axis.

Add title and labels to the x- and y-axis:

```
# Title and Labels in a Plot
import matplotlib.pyplot as plt

plt.title("Travel Statistics")
plt.xlabel("Days")
plt.ylabel("Distance Covered (km)")
x = list(range(1, 6))
y = [240, 100, 288, 0, 160]
plt.plot(x, y)
plt.show()
```

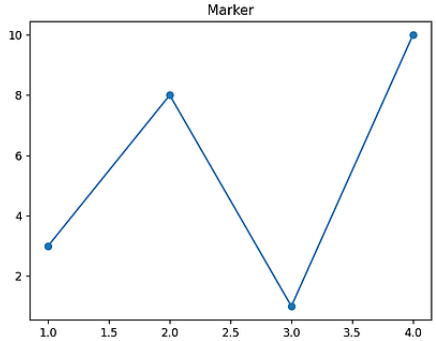


Markers:

The keyword argument marker can be used to emphasize each point with a specified marker:

Mark each point with a circle:

```
import matplotlib.pyplot as plt

plt.title("Marker")

xp = list(range(1, 5))
yp = [3, 8, 1, 10]
plt.plot(xp, yp, marker = 'o')
plt.show()
```



The markers are: `'o', '*', '.', ',', 'x', 'X', '+', 'P', 's', 'D', 'd', 'p', 'H', 'h', 'v', '^', '<', '>', '1', '2', '3', '4', '|', '_'`.

To mark each point with a star: `plt.plot(xs, ys, marker = '*')`

See https://www.w3schools.com/python/matplotlib_markers.asp for a list of markers.

Format Strings fmt:

A shortcut string notation parameter can be used to specify the marker, line type and color.

This parameter is also called **fmt**, and is written with this syntax: marker | line | color.

If format parameter is placed as 3rd parameter (after x and y points), then fmt= can be omitted:

`plt.plot(x, y, fmt="*:b")` or: `plt.plot(x, y, "*:b")`

Line Reference:

Line Syntax	Description
'-'	Solid line
':'	Dotted line
'--'	Dashed line
'-.'	Dashed/dotted line

Color Reference:

Color Syntax	Description
'r'	Red
'g'	Green
'b'	Blue
'c'	Cyan
'm'	Magenta
'y'	Yellow
'k'	Black
'w'	White

Try:

```
plt.plot(x, y, "o--b")
plt.plot(x, y, "+:r")
plt.plot(x, y, "*-.g")
plt.plot(x, y, "x-m")
```

Grid:

With Pyplot, grid() function is used to add grid lines to the plot.

Legend:

To place a legend:

- labels need to be specified in plot() method calls,
- plt.legend() must be called to display those labels.
- to choose legend location, use plt.legend(loc=....).

See the example in "Multiple Lines" below.

Ref: https://matplotlib.org/stable/api/_as_gen/matplotlib.pyplot.legend.html

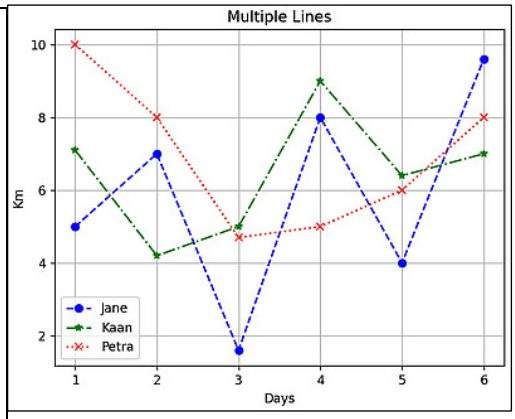
best	0
upper right	1
upper left	2
lower left	3
lower right	4
right	5
center left	6
center right	7
lower center	8
upper center	9
center	10

Multiple Lines:

You can plot as many lines as you like by simply adding more plt.plot() functions:

Draw three lines by specifying a plt.plot() function for each line:

```
import matplotlib.pyplot as plt

plt.title("Multiple Lines")
plt.xlabel("Days")
plt.ylabel("Km")

xs = list(range(1, 7))  # 1 to 6 (common)
y1 = [5, 7, 1.6, 8, 4, 9.6]
y2 = [7.1, 4.2, 5, 9, 6.4, 7]
y3 = [10, 8, 4.7, 5, 6, 8]

plt.plot(xs, y1, "o--b", label="Jane")
plt.plot(xs, y2, "*-.g", label="Kaan")
plt.plot(xs, y3, "x:r", label="Petra")
plt.grid()
plt.legend()      # default: (loc='best')
plt.show()
```

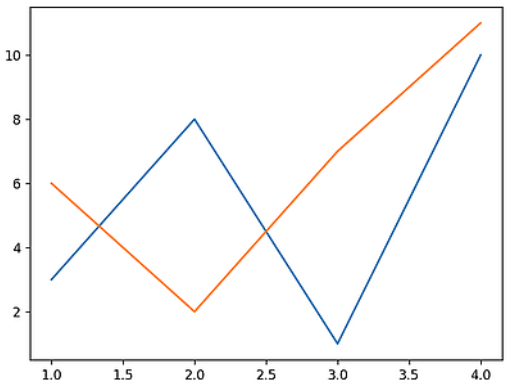


Multiple Lines with a single plot():

You can plot many lines by adding the points for the x- and y-axis for each line in the same plt.plot() function.

The x and y values come in pairs:

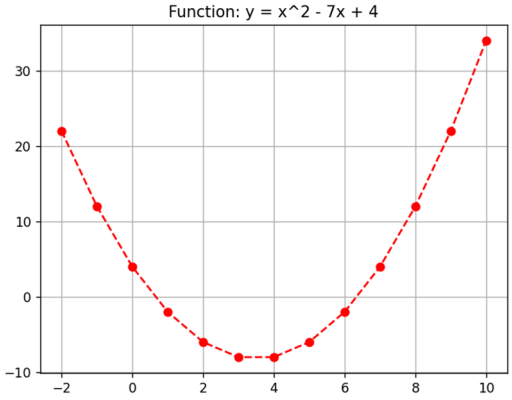
```
import matplotlib.pyplot as plt

x1 = list(range(1, 5))
y1 = [3, 8, 1, 10]
x2 = x1
y2 = [6, 2, 7, 11]

plt.plot(x1, y1, x2, y2)
plt.show()
```



Plotting Functions:

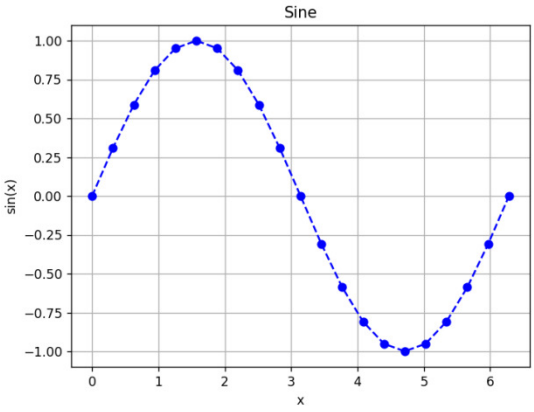
A function can be plotted by creating y-points of the function for given x points.

```
# Draw an x-y plot of a function.

# Function: y = ax^2 + bx + c
def func1(xs, a, b, c):
    return [(a*x**2 +b*x +c) for x in xs]

plt.title("Function: y = x^2 - 7x + 4")
xs = list(range(-2, 11))  # -2 -1 0 ... 10
ys = func1(xs, 1, -7, 4)
plt.plot(xs, ys, "o--r")
plt.grid()
plt.show()
```

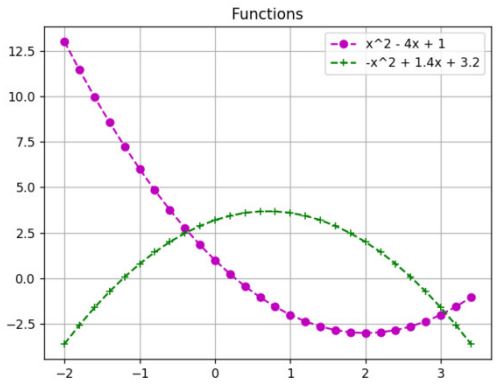


Another example that plots a function: sine function from 0 to 2Π.

```
# Draw an x-y plot of sine function.
import matplotlib.pyplot as plt
import numpy as np
import math

x = np.linspace(0, 2 * math.pi, 21)
y = np.sin(x)     # NumPy has sin() func.

plt.title("Sin")
plt.xlabel("x")
plt.ylabel("sin(x)")
plt.plot(x, y, "o--")
plt.show()
```



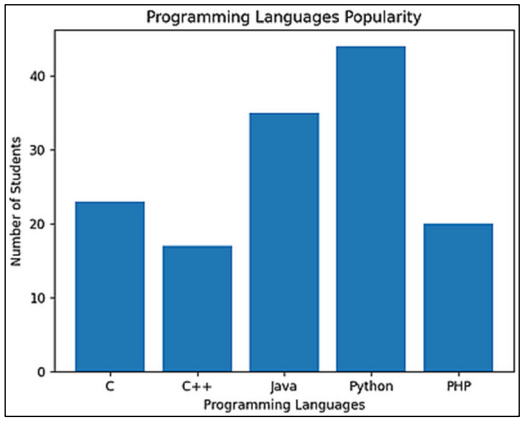
A plot containing two functions:
(See A12_2D_Plot_12_func.py program.)



Bar Charts:

With Pyplot, the bar() function is used to draw bar charts.

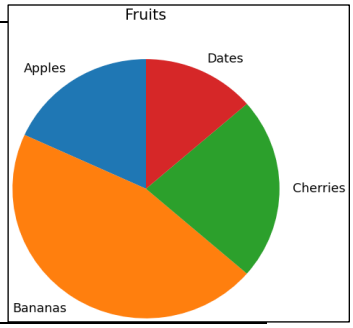
```
# Draw a bar chart.
import matplotlib.pyplot as plt

plt.title("Programming Languages Popularity")
plt.xlabel("Programming Languages")
plt.ylabel("Number of Students")
langs = ['C', 'C++', 'Java', 'Python', 'PHP']
students = [23, 17, 35, 44, 20]
plt.bar(langs, students)
plt.show()
```



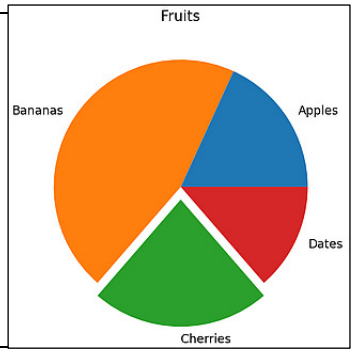
Pie Charts:

With Pyplot, the pie() function is used to draw pie charts.

```
# Pie chart
import matplotlib.pyplot as plt

plt.title("Fruits")

y = [20, 50, 25, 15]
fruits = ["Apples", "Bananas", "Cherries", "Dates"]
plt.pie(y, labels= fruits, startangle= 90)
plt.show()
```




```
# Pie chart
import matplotlib.pyplot as plt

plt.title("Fruits")

y = [20, 50, 25, 15]
labels = ["Apples", "Bananas", "Cherries", "Dates"]
expList = [0, 0, 0.1, 0]
plt.pie(y, labels= labels, explode= expList)
plt.show()
```



References:

https://www.w3schools.com/python/matplotlib_intro.asp

https://matplotlib.org/3.1.1/api/_as_gen/matplotlib.pyplot.html (complete list of methods)

13. Classes and Objects

Python is an object oriented programming language.
In Object Oriented Programming, there are classes, and objects created based on these classes.
A Class is a "blueprint" for creating objects.
Almost everything in Python is an object. An object has its properties and methods.

Class

A Class is a "blueprint" for creating objects. To create a class, the keyword class is used:

```
class MyClass:
```

Object

Objects can be created from defined classes.

```
p1 = MyClass(....)    # Create an object named p1 from class MyClass
```

The Constructor Function __init__()

All classes have a function called __init__().

The __init__() function is called automatically every time a new object is created from a class.

The __init__() function is used to assign values to object properties, and perform other operations that are necessary to do when the object is being created.

```
class Student:
    def __init__(self, id, name, dept):    # Constructor method (function)
        self.id = id               # Assignment to object attribute
        self.name = name           # Assignment to object attribute
        self.dept = dept           # Assignment to object attribute

    def __str__(self):        # String representation of objects
        return f"{self.id:9}: {self.name:24} {self.dept}"

    # Other object methods (functions)

# Main:
s1 = Student(200201777, "Jane Fonda", "CS")    # Create a Student object
print(s1)             # automatically calls s1.__str__()
```

The self Parameter

The self parameter is a reference to the current instance of the class, and is used to access variables that belongs to the class.

It has to be the first parameter of any function (method) in the class.

It does not have to be named self, you can call it whatever you like.

Object Methods

Classes can also contain methods.

Methods in classes are functions that belong to the objects of that class.

Using and Changing Object Attributes

The attributes of objects can be accessed and changed.

```
total += obj.price    # Get an attribute of object obj and use it
print(obj.name)       # Get an attribute of object obj and print it
obj.weight = 60       # Change an attribute of object obj
```

Deleting an object and the destructor function __del__()

del can be used to delete an object:

```
del obj1
```

When an object is deleted, if there is a __del__() method in the class, then this method is automatically executed.

```
import datetime       # Date and time library

class Person:
    perCount = 0      # Class Attribute. Number of Person objects

    def __init__(self, name, byear, weight):
        self.name = name          # Assignment to object attribute
        self.birthYear = byear    # Assignment to object attribute
        self.weight = weight      # Assignment to object attribute
        Person.perCount += 1      # Increment objects counter

    def __del__(self):            # Executed when an object is deleted
        Person.perCount -= 1      # Decrement objects counter

    def greet(self):      # Prints a greeting
        print("Hello my name is " + self.name + "!")

    def getAge(self):     # Returns age
        return datetime.datetime.now().year - self.birthYear

    def __str__(self):    # Returns string representation of object
        return self.name + " (" + str(self.birthYear) + ") " + str(self.weight) + "kg."

# Main:
p1 = Person("Jane", 1987, 56)
p1.greet()            # Call an object method
p1.weight = 58        # Change an attribute
age = p1.getAge()     # Call an object method to get age
print(p1.name, "is", age, "years old.")    # Print name attribute and age
print(p1)             # Print string representation of object

p2 = Person("Peter", 1999, 74)
p2.greet()                # Call an object method
p2.birthYear = 1998       # Change an attribute
print(p2.name, "is", p2.getAge(), "years old.")

print(Person.perCount)    # Print number of Person objects

del p2       # Delete p2 object
print(Person.perCount)    # Print number of Person objects
```

List of objects example:

```
class Car:
    def __init__(self, plate, mark, year)
        ....
    #....

# Main:
cars = []             # List of Car objects, initially empty
cars.append(Car("07ABC124", "Renault", 2016))   # Create a Car obj and append to list
cars.append(Car("35KML24", "Audi", 2019))       # Create a Car obj and append to list
```

Encapsulation in Python:

Encapsulation is about protecting data inside a class.

It means keeping data (properties) and methods together in a class, while controlling how the data can be accessed from outside the class.

This prevents accidental changes to the data and hides the internal details of how the class works.

In Python, encapsulation is implemented by preceeding attribute name by double underscore (__).

Encapsulation example:

```
class Car:
    def __init__(self, plate, mark, model, color, km)
        self.__plate = plate
        self.__mark = mark
        self.__model = model
        self.__color = color
        self.__km = km

    # Accessors (get methods):
    def get_plate(self):
        return self.__plate
    def get_mark(self):
        return self.__mark
    .... ....
    def get_km(self):
        return self.__km

    # Mutators (set methods):
    def set_color(self, color):
        self.__color = color
    def set_km(self, km):
        self.__km = km

# Main:
car1 = Car("07ABC124", "Renault", "Megane", 64000, 2022)
car1.set_km(64800)
print("Plate:", car1.get_plate(), " Mark:", car1.get_mark() )
```

14. Python GUI Programming

Tkinter is the standard GUI library for Python. Tkinter provides a powerful object-oriented interface to the Tk GUI toolkit.

To create a GUI application using Tkinter, the following steps needs to be performed:

- Import the Tkinter module (from tkinter import *, import tkinter, or import tkinter as tk),
- Create the GUI application main window,
- Add one or more widgets to the window.
- Enter the main event loop to take action against each event triggered by the user.

```
import tkinter as tk           # Usually tk is used as alias.

win = tk.Tk()                  # Create window
win.title("First Window")      # Set window name
# Add widgets:
tk.Label(win, text="Hi, I am a label!").pack()
ent1 = tk.Entry(win, width=20)
ent1.pack()
win.mainloop()                 # Display window and enter event loop
```

```
from tkinter import *          # Easier to use (no need for package name)

win = Tk()                     # Create window
win.title("Second Window")     # Set window name
win.geometry("600x400")        # Set window size
# Add widgets:
Label(win, text="Hi, I am a label!").pack()
ent1 = Entry(win, width=20)
ent1.pack()
win.mainloop()                 # Display window and enter event loop
```

14.1. Layout and Geometry Management

All Tkinter widgets have access to specific geometry management methods, which have the purpose of organizing widgets throughout the parent widget area. Tkinter exposes the following geometry manager classes: **pack**, **grid**, and **place**.

Pack:

This geometry manager organizes widgets in blocks before placing them in the parent widget.

- Places each widget below the previous one.
- pack(side = TOP), pack(side = LEFT), pack(side = RIGHT), pack(side = BOTTOM) can be used for alignment.
- To make the widgets as wide as the parent widget, use the fill=X option.

Grid:

This geometry manager organizes widgets in a table-like structure in the parent widget.

- The widgets are placed in a matrix-like structure in **rows and columns**.

```
Label(win, text="Name:").grid(row=1, column=1, sticky=W)          # W == "w" in tkinter
ent1 = Entry(win, width=20)
ent1.grid(row=1, column=2, sticky="w")
Label(win, text="Address:").grid(row=2, column=1, sticky=NW)      # NW == "nw" in tkinter
txt1 = Text(win, height=3, width=40)
txt1.grid(row=2, column=2, sticky="w")
Button(win, text="Save").grid(row=3, column=1, columnspan=2)
```

Positioning in Grid:

NW	N	NE
W		E
SW	S	SE

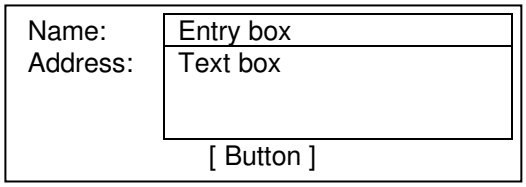
Name:	Entry box
Address:	Text box
[Button]	

Place:

This geometry manager organizes widgets by placing them in a specific horizontal and vertical positions in the parent widget.

Used to organize by specifying the exact x and y coordinates in pixels.

```
Label(win, text="Name:").place(x=4, y=4)
ent1 = Entry(win, width=20)
ent1.place(x=60, y=4)
Label(win, text="Address:").place(x=4, y=28)
txt1 = Text(win, height=3, width=40)
txt1.place(x=60, y=28)
Button(win, text="Save").place(x=180, y=84)
```

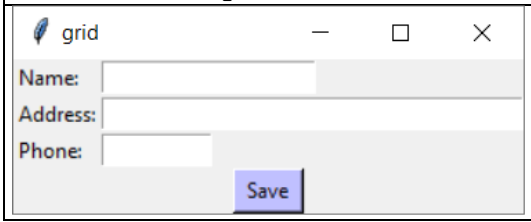


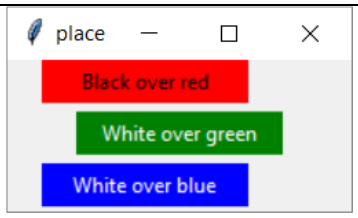
Layout management examples: (All assumes: **`from tkinter import *`**)

```
# Pack
win1 = Tk()                 # Create window
win1.title("pack")          # Set window name
win1.geometry("200x64")     # Set window size

lbl1 = Label(win1, text="Black text over red", bg="red", fg="black")
lbl1.pack()       # lbl1.pack(fill=X)
lbl2 = Label(win1, text="White text over green", bg="green", fg="white")
lbl2.pack()       # lbl2.pack(fill=X)
lbl3 = Label(win1, text="White text over blue", bg="blue", fg="white")
lbl3.pack()       # lbl3.pack(fill=X)

win1.mainloop()
```

with ...pack(fill=X)


```
# Grid
win1 = Tk()  # Create window
win1.title("grid")    # Set window name
#win1.geometry("300x88")  # Set window size

Label(win1, text="Name:").grid(row=0, column=0, sticky="w")
ent1 = Entry(win1, width=20)
ent1.grid(row=0, column=1, sticky="w")
Label(win1, text="Address:").grid(row=1, column=0, sticky="w")
ent2 = Entry(win1, width=40)
ent2.grid(row=1, column=1, sticky="w")
Label(win1, text="Phone:").grid(row=2, column=0, sticky="w")
ent3 = Entry(win1, width=10)
ent3.grid(row=2, column=1, sticky="w")
btn1 = Button(win1, text=" Save ", bg="#C0C0FF")
btn1.grid(row=4, columnspan=2)

win1.mainloop()
```





```
# Place
win1 = Tk()                 # Create window
win1.title("place")         # Set window name
win1.geometry("200x88")     # Set window size

lbl1 = Label(win1, text="Black over red", bg="red", fg="black")
lbl1.place(x = 20, y = 0, width=120, height=25)
lbl2 = Label(win1, text="White over green", bg="green", fg="white")
lbl2.place(x = 40, y = 30, width=120, height=25)
lbl3 = Label(win1, text="White over blue", bg="blue", fg="white")
lbl3.place(x = 20, y = 60, width=120, height=25)

win1.mainloop()
```

14.2. Tkinter Widgets:

Widgets are elements in a window. The most frequently used widgets are:

Widget	Description
Frame	Frame widget is used as a container widget to organize other widgets.
Label	Label widget is used to provide a single-line caption for other widgets. It can also contain images. To create a label widget: lbl1 = Label(win, text="Student Name:") To get label text: str2 = lbl1["text"] To change text in a label widget: lbl1["text"] = str1
Entry	Entry widget is used to display a single-line text field for accepting values from a user. To create an entry widget: ent1 = Entry(win, width=10) To get string entered to entry widget: str1 = ent1.get() To change text in an entry widget: ent1.insert(0, str1) To clear contents of an entry widget: ent1.delete(0, END)
Text	Text widget is used to display or get text in multiple lines. To create a text widget: txt1 = Text(win, height=3, width=40) To get contents of a text widget: str1 = txt1.get("0.0", END) # ....., tk.END) To change text in a text widget: txt1.insert("0.0", contents) To clear contents of a text widget: txt1.delete("0.0", END)
Combobox	A combobox allows user to select one value in a list of values. In addition, it allows to enter a custom value. It is in ttk in tkinter: from tkinter import ttk To create a combobox: combo1 = ttk.Combobox(win, width=8) To set values of a combobox: combo1["values"] = ("...", "...", "...", "...") To get value from a combobox widget: current_value = combo1.get()
Button	Button widget is used to display buttons in a window. To create a button: btn1 = Button(win, text="Do ....", command=func1)
Canvas	Canvas widget is used to display images or to draw shapes, such as lines, ovals, polygons and rectangles, in your application.
Checkbutton	Checkbutton widget is used to display a number of options as checkboxes. The user can select multiple options at a time.
Listbox	Listbox widget is used to provide a list of options to the user.
Menubutton	Menubutton widget is used to display menus in an application.
Menu	Menu widget is used to provide various commands to a user. These commands are contained inside Menubutton.
Message	Message widget is used to display multiline text fields for accepting values from a user.
Radiobutton	Radiobutton widget is used to display a number of options as radio buttons. The user can select only one option at a time.
Scrollbar	Scrollbar widget is used to add scrolling capability to various widgets, such as list boxes.

Standard attributes of widgets:

There are some common attributes, such as sizes, colors and fonts.

Text color	Default color is black.	fg="..."	fore="..."	or foreground="..."
Background	Default is the color of parent.	bg="..."	back="..."	or background="..."
Font	Font can be changed by font=.... parameter. Font value can be a list or tuple. Eg: ("Arial", 12, "bold")			

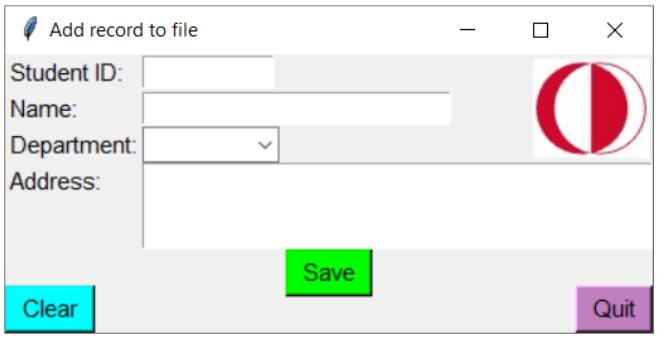
Example:

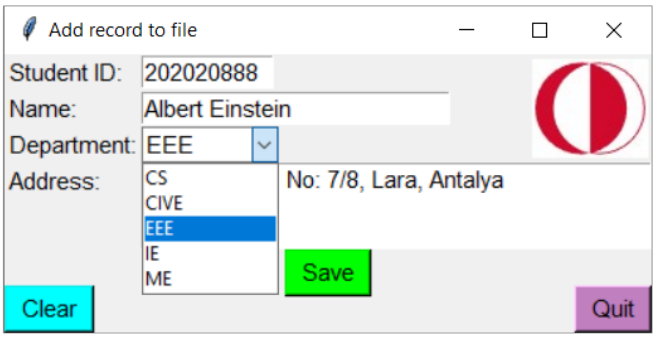
```
Arial11 = ("Arial", 11, "normal")      # Define a font tuple
Arial12b = ("Arial", 12, "bold")       # Define another font tuple
Label(win, text="Abcd", font=Arial11, fg="blue", bg="orange")
ent1 = Entry(win, width=10, font=Arial12B, fg="red", bg="cyan")
txt1 = Text(win, height=8, width=72, font=Arial11, fg="purple")
Button(win, text="Next", font=Arial12b, bg="red", width=20, height=2)
```

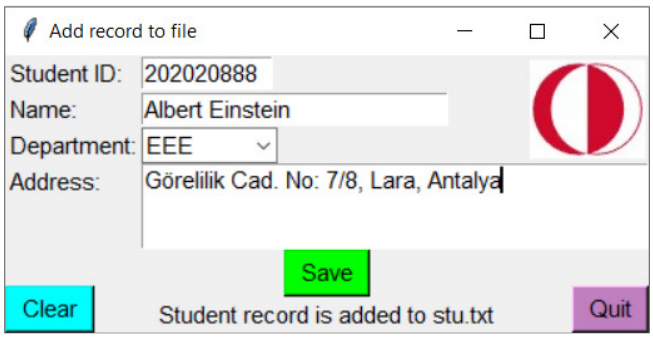
Example: A GUI application that appends records to a text file.

```
# Add record to file         (see: A14_GUI_7_add_record.py)
# Author: Halil Özmen

from tkinter import *
from tkinter import ttk
from PIL import ImageTk, Image

def save():
     # First, get data and correct them:
     stuid = ent1.get().replace(";", " ")
     name = ent2.get().replace(";", ",")
     dept = combo1.get().replace(";", "")
     address = txt1.get("1.0", END).replace(";", ",").replace("\n", " ")
     # Check if there is error:
     if len(stuid) == 0 or len(name) == 0 or len(dept) == 0 or len(address) < 2:
          lbl4["text"] = "Missing data."
          return
     # Create data line:
     line = stuid + ";" + name + ";" + dept + ";" + address + "\n";
     file1 = open("stu.txt", "a") # Open file in append mode
     file1.write(line)  # Write (append) line to end of file.
     file1.close()
     lbl4["text"] = "Student record is added to stu.txt"

def clear():       # Clear fields.
     ent1.delete(0, END)
     ent2.delete(0, END)
     combo1.set("")
     txt1.delete("0.0", END)
     lbl4["text"] = ""

def quit():
     win.destroy()

# Main:
def main():
     global win, ent1, ent2, combo1, txt1, lbl4

     win = Tk()              # Create window
     win.title("Add record to file")  # Set window name
     Arial11 = ("Arial", 11, "normal")# Font tuple

     # Place an image
     img = ImageTk.PhotoImage(Image.open("univ_icon.jpg"))
     Label(win, image=img).grid(row=1, column=1, rowspan=3, sticky=E)

     Label(win, text="Student ID:", font=Arial11).grid(row=1, column=0, sticky=W)
```

```
    ent1 = Entry(win, width=10, font=Arial11)
    ent1.grid(row=1, column=1, sticky=W)

    Label(win, text="Name:", font=Arial11).grid(row=2, column=0, sticky=W)
    ent2 = Entry(win, width=24, font=Arial11)
    ent2.grid(row=2, column=1, sticky=W)

    Label(win, text="Department:", font=Arial11).grid(row=3, column=0, sticky=W)
    combo1 = ttk.Combobox(win, width=8, font=Arial11)
    combo1["values"] = ("CS", "CIVE", "EEE", "IE", "ME")
    combo1.grid(row=3, column=1, sticky=W)

    Label(win, text="Address:", font=Arial11).grid(row=4, column=0, sticky=NW)
    txt1 = Text(win, height=3, width=40, font=Arial11)
    txt1.grid(row=4, column=1)

    btnsave = Button(win, text=" Save ", command=save, font=Arial11, bg="lime")
    btnsave.grid(row=8, columnspan=2)

    btnclear = Button(win, text=" Clear ", command=clear, font=Arial11, bg="cyan")
    btnclear.grid(row=8, rowspan=2, columnspan=2, sticky=SW)

    btnquit = Button(win, text=" Quit ", command=quit, font=Arial11, bg="#C080C0")
    btnquit.grid(row=8, rowspan=2, columnspan=2, sticky=SE)

    lbl4 = Label(win, text="", font=Arial11)
    lbl4.grid(row=9, columnspan=2)

    win.mainloop()

# Run main
if __name__ == '__main__':
    main()
```







References:

https://www.edureka.co/blog/tkinter-tutorial/ http://www.pythonlake.com/python/tkinter
https://www.python-course.eu/python_tkinter.php
https://realpython.com/python-gui-tkinter/
https://www.tutorialspoint.com/python/python_gui_programming.htm

A. Random Number Generation

The "**random**" library is used to generate pseudo-random numbers.

Some random library functions:

Function	Description
random.seed()	Initializes the random number generator.
random.random()	Returns a value between 0.0 (inclusive) and 1.0 (exclusive).
random.randint(n1, n2)	Returns a random integer n so that n1 <= n <= n2
random.randrange(n)	Returns a random integer between 0 (inclusive) and n (exclusive)
random.randrange(n1, n2)	Returns a randomly selected element from range(n1, n2), n2 exclusive
random.randrange(n1, n2, step)	Returns a randomly selected element from range(n1, n2, step), n2 exclusive
random.choice(seq)	Returns a random element from the non-empty sequence seq (list, set, tuple, string, range()). If seq is empty, raises IndexError.
random.sample(seq, k)	Returns a particular length (k) list of items chosen from the sequence i.e. list, tuple, string, set or range().

Example:

```
import random

random.seed()    # Initialize the random number generator.

# Random values from 0.0 (inclusive) to 1.0 (exclusive)
for j in range(10):                # Repeat 10 times.
    value = random.random()        # [0.0 ... 1.0)
    print(value)
print(40 * "=")

# 10 pair of dice values:
for j in range(10):                # Repeat 10 times.
    die1 = random.randint(1, 6)    # [1 ... 6]
    die2 = random.randint(1, 6)    # [1 ... 6]
    print(die1, die2, sep="", end=" ")

k = random.randrange(10, 100, 10))     # randomly from 10 20 30 ... 90
print(k)
m = random.randrange(28, 100, 7))      # randomly from 28 35 42 ... 98
print(m)

lst = [.., .., .., .., ....]
x = random.choice(lst)         # A random element from the list lst

s = "abcdefgh12345678"
c = random.choice(s)           # A random char from string s

# Create a list of 8 random elements from (10 20 30 ... 400)
a = random.sample(range(10, 401, 10), 8)
```