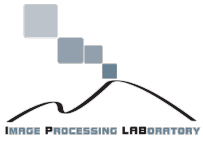

Recall of basic concepts of the Python language

1. Using Python as a Calculator

1.1 Numbers

Expression syntax is straightforward: the operators +, -, * and / can be used to perform arithmetic; parentheses (()) can be used for grouping. For example:

In [2]:
```
2 + 2
```

Out[2]: 4

In [3]:
```
50 - 5*6
```

Out[3]: 20

In [4]:
```
(50 - 5*6) / 4
```

Out[4]: 5.0

In [5]:
```
8 / 5
```

Out[5]: 1.6

With Python, it is possible to use the ** operator to calculate powers:

In [6]:
```
5 ** 2  # 5 squared
```

Out[6]: 25

In [7]:
```
2 ** 7
```

Out[7]: 128

The equal sign (=) is used to assign a value to a variable. Afterwards, no result is displayed before the next interactive prompt:

In [8]:
```
width = 20
height = 5 * 9
width * height
```

Out[8]: 900

In interactive mode, the last printed expression is assigned to the variable _. This means that when you are using Python as a desk calculator, it is somewhat easier to continue calculations, for example:

In [9]:
```
tax = 12.5 / 100
```

Out[9]: 900

In [10]:
```
price = 100.50
```

In [11]:
```
price * tax
```

Out[11]: 12.5625

In [12]:
```
price + _
```

Out[12]: 113.0625

In [13]:
```
round(_, 2)
```

Out[13]: 113.06

This variable should be treated as read-only by the user. Don't explicitly assign a value to it — you would create an independent local variable with the same name masking the built-in variable with its magic behavior.

In addition to int and float, Python supports other types of numbers, such as Decimal and Fraction. Python also has built-in support for complex numbers, and uses the j or J suffix to indicate the imaginary part (e.g. 3+5j).

2. Text

Python can manipulate text (represented by type str, so-called "strings") as well as numbers. This includes characters "!", words "rabbit", names "Paris", sentences "Got your back.", etc. "Yay! :)". They can be enclosed in single quotes ('...') or double quotes ("...") with the same result

In [14]:
```
'spam eggs'  # single quotes
```

Out[14]: 'spam eggs'

In [15]:
```
"Paris rabbit got your back :)! Yay!"  # double quotes
```

Out[15]:
```
'Paris rabbit got your back :)! Yay!'
```

In [16]:
```
'1975'  # digits and numerals enclosed in quotes are also strings
```

Out[16]:
```
'1975'
```

In [17]:
```
'doesn\'t'  # use \' to escape the single quote...

"doesn't"  # ...or use double quotes instead

'"Yes," they said.'

"\"Yes,\" they said."

'"Isn\'t," they said.'
```

Out[17]:
```
'"Isn\'t," they said.'
```

In the Python shell, the string definition and output string can look different. The print() function produces a more readable output, by omitting the enclosing quotes and by printing escaped and special characters:

In [18]:
```
s = 'First line.\nSecond line.'  # \n means newline
s  # without print(), special characters are included in the string

print(s)  # with print(), special characters are interpreted, so \n produces new
```

```
First line.
Second line.
```

If you don’t want characters prefaced by \ to be interpreted as special characters, you can use raw strings by adding an r before the first quote:

In [19]:
```
print('C:\some\name')  # here \n means newline!
```

```
C:\some
ame
```

In [20]:
```
print(r'C:\some\name')  # note the r before the quote
```

```
C:\some\name
```

There is one subtle aspect to raw strings: a raw string may not end in an odd number of \ characters; see the FAQ entry for more information and workarounds.

String literals can span multiple lines. One way is using triple-quotes: """..."""" or '''...'''. End of lines are automatically included in the string, but it’s possible to prevent this by adding a \ at the end of the line. The following example:

In [21]:
```
print("""\
Usage: thingy [OPTIONS]
     -h                        Display this usage message
     -H hostname               Hostname to connect to
""")
```

```
Usage: thingy [OPTIONS]
     -h                        Display this usage message
     -H hostname               Hostname to connect to
```

If you want to concatenate variables or a variable and a literal, use +:

In [23]:
```
prefix = 'x'
prefix + 'thon'
```

Out[23]: 'xthon'

Strings can be indexed (subscripted), with the first character having index 0. There is no separate character type; a character is simply a string of size one:

In [24]:
```
word = 'Python'
word[0]  # character in position 0

word[5]  # character in position 5
```

Out[24]: 'n'

Indices may also be negative numbers, to start counting from the right:

In [25]:
```
word[-1]  # last character

word[-2]  # second-last character

word[-6]
```

Out[25]: 'P'

Note that since -0 is the same as 0, negative indices start from -1.

In addition to indexing, slicing is also supported. While indexing is used to obtain individual characters, slicing allows you to obtain a substring:

In [26]:
```
word[0:2]  # characters from position 0 (included) to 2 (excluded)

word[2:5]  # characters from position 2 (included) to 5 (excluded)
```

Out[26]: 'tho'

Slice indices have useful defaults; an omitted first index defaults to zero, an omitted second index defaults to the size of the string being sliced.

In [27]:
```
word[:2]   # character from the beginning to position 2 (excluded)

word[4:]   # characters from position 4 (included) to the end

word[-2:]  # characters from the second-last (included) to the end
```

Out[27]: 'on'

Note how the start is always included, and the end always excluded. This makes sure that s[:i] + s[i:] is always equal to s:

In [28]:
```
word[:2] + word[2:]

word[:4] + word[4:]
```

Out[28]: 'Python'

The function len() returns the length of a string:

In [29]:
```
s = 'supercalifragilisticexpialidocious'
len(s)
```

Out[29]: 34

3. List

Python knows a number of compound data types, used to group together other values. The most versatile is the list, which can be written as a list of comma-separated values (items) between square brackets. Lists might contain items of different types, but usually the items all have the same type.

In [1]:
```
squares = [1, 4, 9, 16, 25]
squares
```

Out[1]: [1, 4, 9, 16, 25]

In [2]:
```
squares[0]  # indexing returns the item
```

Out[2]: 1

In [3]:
```
squares[-1]
```

Out[3]: 25

In [4]:
```
squares[-3:]  # slicing returns a new list
```

Out[4]: [9, 16, 25]

Lists also support operations like concatenation:

In [5]:
```
squares + [36, 49, 64, 81, 100]
```

Out[5]: [1, 4, 9, 16, 25, 36, 49, 64, 81, 100]

Unlike strings, which are immutable, lists are a mutable type, i.e. it is possible to change their content:

In [11]:
```
cubes = [1, 8, 27, 65, 125]  # something's wrong here
4 ** 3  # the cube of 4 is 64, not 65!

cubes[3] = 64  # replace the wrong value
cubes
```

Out[11]:
```
[1, 8, 27, 64, 125]
```

You can also add new items at the end of the list, by using the append() method (we will see more about methods later):

In [12]:
```
cubes.append(216)  # add the cube of 6
cubes.append(7 ** 3)  # and the cube of 7
cubes
```

Out[12]:
```
[1, 8, 27, 64, 125, 216, 343]
```

Assignment to slices is also possible, and this can even change the size of the list or clear it entirely:

In [13]:
```
letters = ['a', 'b', 'c', 'd', 'e', 'f', 'g']
letters
```

Out[13]:
```
['a', 'b', 'c', 'd', 'e', 'f', 'g']
```

In [14]:
```
# replace some values
letters[2:5] = ['C', 'D', 'E']
letters
```

Out[14]:
```
['a', 'b', 'C', 'D', 'E', 'f', 'g']
```

In [15]:
```
# now remove them
letters[2:5] = []
letters
```

Out[15]:
```
['a', 'b', 'f', 'g']
```

In [16]:
```
# clear the list by replacing all the elements with an empty list
letters[:] = []
letters
```

Out[16]:
```
[]
```

The built-in function len() also applies to lists:

In [17]:
```
letters = ['a', 'b', 'c', 'd']
len(letters)
```

Out[17]:
```
4
```

It is possible to nest lists (create lists containing other lists), for example:

In [20]:
```
a = ['a', 'b', 'c']
n = [1, 2, 3]
x = [a, n]
x
```

Out[20]:
```
[['a', 'b', 'c'], [1, 2, 3]]
```

In [21]:
```
x[0]
```

Out[21]:
```
['a', 'b', 'c']
```

In [22]:
```
x[0][1]
```

Out[22]:
```
'b'
```

3. First Steps Towards Programming

Of course, we can use Python for more complicated tasks than adding two and two together. For instance, we can write an initial sub-sequence of the Fibonacci series as follows:

In [23]:
```
# Fibonacci series:
# the sum of two elements defines the next
a, b = 0, 1
while a < 10:
    print(a)
    a, b = b, a+b
```

```
0
1
1
2
3
5
8
```

This example introduces several new features.

- The first line contains a multiple assignment: the variables a and b simultaneously get the new values 0 and 1. On the last line this is used again, demonstrating that the expressions on the right-hand side are all evaluated first before any of the assignments take place. The right-hand side expressions are evaluated from the left to the right.
- The while loop executes as long as the condition (here: a < 10) remains true. In Python, like in C, any non-zero integer value is true; zero is false. The condition may also be a string or list value, in fact any sequence; anything with a non-zero length is true, empty sequences are false. The test used in the example is a simple comparison. The standard comparison operators are written the same as in C: < (less than), > (greater than), == (equal to), <= (less than or equal to), >= (greater than or equal to) and != (not equal to).
- The body of the loop is indented: indentation is Python's way of grouping statements. At the interactive prompt, you have to type a tab or space(s) for each indented line. In practice you will prepare more complicated input for Python with a text editor; all decent text editors have an auto-indent facility. When a compound statement is entered interactively, it must be followed by a blank line to indicate completion (since the parser cannot guess when you have typed the last line). Note that each line within a basic block must be indented by the same amount.
- The print() function writes the value of the argument(s) it is given. It differs from just writing the expression you want to write (as we did earlier in the calculator examples) in the way it handles

multiple arguments, floating point quantities, and strings. Strings are printed without quotes, and a

In [24]:
```
i = 256*256
print('The value of i is', i)
```

```
The value of i is 65536
```

The keyword argument end can be used to avoid the newline after the output, or end the output with a different string:

In [25]:
```
a, b = 0, 1
while a < 1000:
    print(a, end=',')
    a, b = b, a+b
```

```
0,1,1,2,3,5,8,13,21,34,55,89,144,233,377,610,987,
```

4. Control Flow Tools

4.1 if Statements

In [1]:
```
x = int(input("Please enter an integer: "))

if x < 0:
    x = 0
    print('Negative changed to zero')
elif x == 0:
    print('Zero')
elif x == 1:
    print('Single')
else:
    print('More')
```

```
Please enter an integer: 2
More
```

4.2. for Statements

The for statement in Python differs a bit from what you may be used to in C or Pascal. Rather than always iterating over an arithmetic progression of numbers (like in Pascal), or giving the user the ability to define both the iteration step and halting condition (as C), Python's for statement iterates over the items of any sequence (a list or a string), in the order that they appear in the sequence. For example (no pun intended):

In [2]:
```
# Measure some strings:
words = ['cat', 'window', 'defenestrate']
for w in words:
    print(w, len(w))
```

```
cat 3
window 6
defenestrate 12
```

4.3. The range() Function

If you do need to iterate over a sequence of numbers, the built-in function range() comes in handy. It generates arithmetic progressions:

In [3]:
```
for i in range(5):
    print(i)
```

```
0
1
2
3
4
```

The given end point is never part of the generated sequence; range(10) generates 10 values, the legal indices for items of a sequence of length 10. It is possible to let the range start at another number, or to specify a different increment (even negative; sometimes this is called the 'step'):

In [4]:
```
list(range(5, 10))


list(range(0, 10, 3))


list(range(-10, -100, -30))
```

Out[4]:
```
[-10, -40, -70]
```

To iterate over the indices of a sequence, you can combine range() and len() as follows:

In [5]:
```
a = ['Mary', 'had', 'a', 'little', 'lamb']
for i in range(len(a)):
    print(i, a[i])
```

```
0 Mary
1 had
2 a
3 little
4 lamb
```

4.4. break and continue Statements, and else Clauses on Loops

The break statement breaks out of the innermost enclosing for or while loop.

A for or while loop can include an else clause.

In a for loop, the else clause is executed after the loop reaches its final iteration.

In a while loop, it's executed after the loop's condition becomes false.

In either kind of loop, the else clause is not executed if the loop was terminated by a break.

This is exemplified in the following for loop, which searches for prime numbers:

In [6]:
```
for n in range(2, 10):
    for x in range(2, n):
        if n % x == 0:
            print(n, 'equals', x, '*', n//x)
            break
    else:
        # loop fell through without finding a factor
        print(n, 'is a prime number')
```

```
2 is a prime number
3 is a prime number
4 equals 2 * 2
5 is a prime number
6 equals 2 * 3
7 is a prime number
8 equals 2 * 4
9 equals 3 * 3
```

(Yes, this is the correct code. Look closely: the else clause belongs to the for loop, not the if statement.)

When used with a loop, the else clause has more in common with the else clause of a try statement than it does with that of if statements: a try statement's else clause runs when no exception occurs, and a loop's else clause runs when no break occurs. For more on the try statement and exceptions, see Handling Exceptions.

The continue statement, also borrowed from C, continues with the next iteration of the loop:

In [7]:
```
for num in range(2, 10):
    if num % 2 == 0:
        print("Found an even number", num)
        continue
    print("Found an odd number", num)
```

```
Found an even number 2
Found an odd number 3
Found an even number 4
Found an odd number 5
Found an even number 6
Found an odd number 7
Found an even number 8
Found an odd number 9
```

4.7. Defining Functions

We can create a function that writes the Fibonacci series to an arbitrary boundary:

In [8]:
```
def fib(n):    # write Fibonacci series up to n
    """Print a Fibonacci series up to n."""
    a, b = 0, 1
    while a < n:
        print(a, end=' ')
        a, b = b, a+b
    print()

# Now call the function we just defined:
fib(2000)
```

```
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597
```

The keyword def introduces a function definition. It must be followed by the function name and the parenthesized list of formal parameters. The statements that form the body of the function start at the next line, and must be indented.

The first statement of the function body can optionally be a string literal; this string literal is the function's documentation string, or docstring. (More about docstrings can be found in the section Documentation Strings.) There are tools which use docstrings to automatically produce online or printed documentation, or to let the user interactively browse through code; it's good practice to include docstrings in code that you write, so make a habit of it.

The execution of a function introduces a new symbol table used for the local variables of the function. More precisely, all variable assignments in a function store the value in the local symbol table; whereas variable references first look in the local symbol table, then in the local symbol tables of enclosing functions, then in the global symbol table, and finally in the table of built-in names. Thus, global variables and variables of enclosing functions cannot be directly assigned a value within a function (unless, for global variables, named in a global statement, or, for variables of enclosing functions, named in a nonlocal statement), although they may be referenced.

It is also possible to define functions with a variable number of arguments. There are three forms, which can be combined.

4.7.1. Default Argument Values

The most useful form is to specify a default value for one or more arguments. This creates a function that can be called with fewer arguments than it is defined to allow. For example:

In [10]:
```
def ask_ok(prompt, retries=4, reminder='Please try again!'):
    while True:
        ok = input(prompt)
        if ok in ('y', 'ye', 'yes'):
            return True
        if ok in ('n', 'no', 'nop', 'nope'):
            return False
        retries = retries - 1
        if retries < 0:
            raise ValueError('invalid user response')
        print(reminder)
```

This function can be called in several ways:

giving only the mandatory argument: ask_ok('Do you really want to quit?')

giving one of the optional arguments: ask_ok('OK to overwrite the file?', 2)

or even giving all arguments: ask_ok('OK to overwrite the file?', 2, 'Come on, only yes or no!')

5. Data Structures

The list data type has some more methods. Here are all of the methods of list objects:

- list.**append**(x) Add an item to the end of the list. Equivalent to a[len(a):] = [x].
- list.**extend**(iterable) Extend the list by appending all the items from the iterable. Equivalent to a[len(a):] = iterable.
- list.**insert**(i, x) Insert an item at a given position. The first argument is the index of the element before which to insert, so a.insert(0, x) inserts at the front of the list, and a.insert(len(a), x) is equivalent to a.append(x).

- list.**remove**(x) Remove the first item from the list whose value is equal to x. It raises a ValueError if there is no such item.
- list.**pop**([i]) Remove the item at the given position in the list, and return it. If no index is specified, a.pop() removes and returns the last item in the list. (The square brackets around the i in the method signature denote that the parameter is optional, not that you should type square brackets at that position. You will see this notation frequently in the Python Library Reference.)
- list.**clear**() Remove all items from the list. Equivalent to del a[:].
- list.**index**(x[, start[, end]]) Return zero-based index in the list of the first item whose value is equal to x. Raises a ValueError if there is no such item.

  The optional arguments start and end are interpreted as in the slice notation and are used to limit the search to a particular subsequence of the list. The returned index is computed relative to the beginning of the full sequence rather than the start argument.
- list.**count**(x) Return the number of times x appears in the list.
- list.**sort**(*, key=None, reverse=False) Sort the items of the list in place (the arguments can be used for sort customization, see sorted() for their explanation).
- list.**reverse**() Reverse the elements of the list in place.
- list.**copy**() Return a shallow copy of the list. Equivalent to a[:].

In [12]:
```
fruits = ['orange', 'apple', 'pear', 'banana', 'kiwi', 'apple', 'banana']
fruits.count('apple')

fruits.count('tangerine')

fruits.index('banana')

fruits.index('banana', 4)  # Find next banana starting at position 4

fruits.reverse()
fruits

fruits.append('grape')
fruits

fruits.sort()
fruits

fruits.pop()
```

Out[12]:
```
'pear'
```

5.1. Using Lists as Stacks

The list methods make it very easy to use a list as a stack, where the last element added is the first element retrieved (“last-in, first-out”). To add an item to the top of the stack, use append(). To retrieve an item from the top of the stack, use pop() without an explicit index. For example:

In [13]:
```
stack = [3, 4, 5]
stack.append(6)
stack.append(7)
stack

stack.pop()

stack

stack.pop()

stack.pop()

stack
```

Out[13]:
```
[3, 4]
```

5.2. Using Lists as Queues

It is also possible to use a list as a queue, where the first element added is the first element retrieved ("first-in, first-out"); however, lists are not efficient for this purpose. While appends and pops from the end of list are fast, doing inserts or pops from the beginning of a list is slow (because all of the other elements have to be shifted by one).

To implement a queue, use collections.deque which was designed to have fast appends and pops from both ends. For example:

In [14]:
```
from collections import deque
queue = deque(["Eric", "John", "Michael"])
queue.append("Terry")           # Terry arrives
queue.append("Graham")          # Graham arrives
queue.popleft()                 # The first to arrive now leaves

queue.popleft()                 # The second to arrive now leaves

queue                           # Remaining queue in order of arrival
```

Out[14]:
```
deque(['Michael', 'Terry', 'Graham'])
```

5.3 List Comprehensions

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 subsequence of those elements that satisfy a certain condition.

For example, assume we want to create a list of squares, like:

In [15]:
```
squares = []
for x in range(10):
    squares.append(x**2)

squares
```

Out[15]:
```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Note that this creates (or overwrites) a variable named x that still exists after the loop completes. We can calculate the list of squares without any side effects using:

In [16]:
```
squares = [x**2 for x in range(10)]
```

which is more concise and readable.

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. For example, this listcomp combines the elements of two lists if they are not equal:

In [19]:
```
[(x, y) for x in [1,2,3] for y in [3,1,4] if x != y]
```

Out[19]:
```
[(1, 3), (1, 4), (2, 3), (2, 1), (2, 4), (3, 1), (3, 4)]
```

and it's equivalent to:

In [20]:
```
combs = []
for x in [1,2,3]:
    for y in [3,1,4]:
        if x != y:
            combs.append((x, y))

combs
```

Out[20]:
```
[(1, 3), (1, 4), (2, 3), (2, 1), (2, 4), (3, 1), (3, 4)]
```

5.4 The *del* statement

There is a way to remove an item from a list given its index instead of its value: the del statement. This differs from the pop() method which returns a value. The del statement can also be used to remove slices from a list or clear the entire list (which we did earlier by assignment of an empty list to the slice). For example:

In [21]:
```
a = [-1, 1, 66.25, 333, 333, 1234.5]
del a[0]
a

del a[2:4]
a

del a[:]
a
```

Out[21]:
```
[]
```

5.5 Dictionaries

Another useful data type built into Python is the dictionary (see Mapping Types — dict). Dictionaries are sometimes found in other languages as "associative memories" or "associative arrays". Unlike sequences, which are indexed by a range of numbers, dictionaries are indexed by keys, which can be any immutable type; strings and numbers can always be keys. Tuples can be used as keys if they contain only strings, numbers, or tuples; if a tuple contains any mutable object either directly or indirectly, it cannot be used as a key. You can't use lists as keys, since lists can be modified in place using index assignments, slice assignments, or methods like append() and extend().

It is best to think of a dictionary as a set of key: value pairs, with the requirement that the keys are unique (within one dictionary). A pair of braces creates an empty dictionary: {}. Placing a comma-separated list of key:value pairs within the braces adds initial key:value pairs to the dictionary; this is also the way dictionaries are written on output.

The main operations on a dictionary are storing a value with some key and extracting the value given the key. It is also possible to delete a key:value pair with del. If you store using a key that is already in use, the old value associated with that key is forgotten. It is an error to extract a value using a non-existent key.

Performing list(d) on a dictionary returns a list of all the keys used in the dictionary, in insertion order (if you want it sorted, just use sorted(d) instead). To check whether a single key is in the dictionary, use the in keyword.

In [22]:
```
tel = {'jack': 4098, 'sape': 4139}
tel['guido'] = 4127
tel

tel['jack']

del tel['sape']
tel['irv'] = 4127
tel

list(tel)

sorted(tel)

'guido' in tel

'jack' not in tel
```

Out[22]:
```
False
```

The dict() constructor builds dictionaries directly from sequences of key-value pairs:

In [23]:
```
dict([('sape', 4139), ('guido', 4127), ('jack', 4098)])
```

Out[23]:
```
{'sape': 4139, 'guido': 4127, 'jack': 4098}
```

In addition, dict comprehensions can be used to create dictionaries from arbitrary key and value expressions:

In [24]:
```
{x: x**2 for x in (2, 4, 6)}
```

Out[24]:
```
{2: 4, 4: 16, 6: 36}
```

When the keys are simple strings, it is sometimes easier to specify pairs using keyword arguments:

In [25]:
```
dict(sape=4139, guido=4127, jack=4098)
```

Out[25]:
```
{'sape': 4139, 'guido': 4127, 'jack': 4098}
```

5.6 Looping Techniques

When looping through dictionaries, the key and corresponding value can be retrieved at the same time using the items() method.

In [27]:
```
knights = {'gallahad': 'the pure', 'robin': 'the brave'}
for k, v in knights.items():
    print(k, v)
```

```
gallahad the pure
robin the brave
```

When looping through a sequence, the position index and corresponding value can be retrieved at the same time using the enumerate() function.

In [28]:
```
for i, v in enumerate(['tic', 'tac', 'toe']):
    print(i, v)
```

```
0 tic
1 tac
2 toe
```

To loop over two or more sequences at the same time, the entries can be paired with the zip() function.

In [29]:
```
questions = ['name', 'quest', 'favorite color']
answers = ['lancelot', 'the holy grail', 'blue']
for q, a in zip(questions, answers):
    print('What is your {0}?  It is {1}.'.format(q, a))
```

```
What is your name?  It is lancelot.
What is your quest?  It is the holy grail.
What is your favorite color?  It is blue.
```

6. Files

6.1 Reading and Writing Files

open() returns a file object, and is most commonly used with two positional arguments and one keyword argument: open(filename, mode, encoding=None)

In [30]:
```
f = open('workfile', 'w', encoding="utf-8")
```

The first argument is a string containing the filename. The second argument is another string containing a few characters describing the way in which the file will be used. mode can be 'r' when the file will only be read, 'w' for only writing (an existing file with the same name will be erased), and 'a' opens the file for appending; any data written to the file is automatically added to the end. 'r+' opens the file for both reading and writing. The mode argument is optional; 'r' will be assumed if it's omitted.

Normally, files are opened in text mode, that means, you read and write strings from and to the file, which are encoded in a specific encoding. If encoding is not specified, the default is platform dependent (see open()). Because UTF-8 is the modern de-facto standard, encoding="utf-8" is recommended

unless you know that you need to use a different encoding. Appending a 'b' to the mode opens the file in binary mode. Binary mode data is read and written as bytes objects. You can not specify encoding when opening file in binary mode.

In text mode, the default when reading is to convert platform-specific line endings (\n on Unix, \r\n on Windows) to just \n. When writing in text mode, the default is to convert occurrences of \n back to platform-specific line endings. This behind-the-scenes modification to file data is fine for text files, but will corrupt binary data like that in JPEG or EXE files. Be very careful to use binary mode when reading and writing such files.

It is good practice to use the with keyword when dealing with file objects. The advantage is that the file

In [45]:
```
with open('workfile', encoding="utf-8") as f:
    read_data = f.read()

# We can check that the file has been automatically closed.
#f.closed
```

If you're not using the with keyword, then you should call f.close() to close the file and immediately free up any system resources used by it.

***Warning*: Calling f.write() without using the with keyword or calling f.close() might result in the arguments of f.write() not being completely written to the disk, even if the program exits successfully.**

6.2 Methods of File Objects

The rest of the examples in this section will assume that a file object called f has already been created.

To read a file's contents, call f.read(size), which reads some quantity of data and returns it as a string (in text mode) or bytes object (in binary mode). size is an optional numeric argument. When size is omitted or negative, the entire contents of the file will be read and returned; it's your problem if the file is twice as large as your machine's memory. Otherwise, at most size characters (in text mode) or size bytes (in binary mode) are read and returned. If the end of the file has been reached, f.read() will return an empty string ('').

In [ ]:
```
f.read()

f.read()
```

f.readline() reads a single line from the file; a newline character (\n) is left at the end of the string, and is only omitted on the last line of the file if the file doesn't end in a newline. This makes the return value unambiguous; if f.readline() returns an empty string, the end of the file has been reached, while a blank line is represented by '\n', a string containing only a single newline.

In [ ]:
```
f.readline()

f.readline()

f.readline()
```

For reading lines from a file, you can loop over the file object. This is memory efficient, fast, and leads to simple code:

In [ ]:
```
for line in f:
    print(line, end='')
```

If you want to read all the lines of a file in a list you can also use list(f) or f.readlines().

f.write(string) writes the contents of string to the file, returning the number of characters written.

In [ ]:
```
f.write('This is a test\n')
```

Other types of objects need to be converted – either to a string (in text mode) or a bytes object (in binary mode) – before writing them:

In [ ]:
```
value = ('the answer', 42)
s = str(value)  # convert the tuple to string
f.write(s)
```

f.tell() returns an integer giving the file object's current position in the file represented as number of bytes from the beginning of the file when in binary mode and an opaque number when in text mode.

To change the file object's position, use f.seek(offset, whence). The position is computed from adding offset to a reference point; the reference point is selected by the whence argument. A whence value of 0 measures from the beginning of the file, 1 uses the current file position, and 2 uses the end of the file as the reference point. whence can be omitted and defaults to 0, using the beginning of the file as the reference point.

In [52]:
```
f = open('workfile', 'rb+')
f.write(b'0123456789abcdef')

f.seek(5)      # Go to the 6th byte in the file

f.read(1)

f.seek(-3, 2)  # Go to the 3rd byte before the end

f.read(1)
```

Out[52]:
```
b'd'
```

In text files (those opened without a b in the mode string), only seeks relative to the beginning of the file are allowed (the exception being seeking to the very file end with seek(0, 2)) and the only valid offset values are those returned from the f.tell(), or zero. Any other offset value produces undefined behaviour.

File objects have some additional methods, such as isatty() and truncate() which are less frequently used; consult the Library Reference for a complete guide to file objects.

ESERCIZI

1. Write a program that asks the user for two numbers using the input function and shows the larger of the two using the print function.
2. Write a program that asks the user for three numbers a, b, c and shows the largest among them.
3. Scrivi un programma che chieda all'utente una lista di numeri e fornisca in output il maggiore tra tutti.

Write a program that asks the user for a list of numbers and outputs the largest of all.

4. Write a simple program that, given a list of numbers, adds all the elements together. Tip: Even if the sum() function exists, you could use the for loop to solve the exercise.
5. Write a program that, starting from an element and a list of elements, tells the output whether the passed element is present in the list or not. If the element is present in the list, the program will have to communicate the index of the element via the index method.
6. Write a simple function that, given a list of numbers, outputs a histogram based on these numbers, using asterisks to draw it. Given for example the list [3, 7, 9, 5], the function must produce this sequence:
   ***
   ****
   ****
   *****
7. Write a function that, given as input a list A containing n words, returns as output a list B of integers representing the length of the words contained in A.
8. A MAC address (Media Access Control address) is a unique address associated by the manufacturer with a chipset for wireless communications (e.g. WiFi or Bluetooth), consisting of 6 pairs of hexadecimal digits separated by colons. An example MAC is 02:FF:A5:F2:55:12. Write a generate_mac() function that generates pseudo-random MAC addresses using the random module.
9. Write a function that adds 10 user-entered colors to a list. The program must then ask the user to enter a letter and output only the colors in the list that begin with that letter.
10. Write a function that takes a series of inputs from the user using a while loop and prints them with the print function without wrapping. The while loop should stop when the user presses ENTER without typing anything.
11. Write a function to which you will pass a string as a parameter, and which will print an inverse (in reverse) version of the same string. For example "abcd" will become "dcba".
12. Write a simple rhyme function, which is passed a list of words as a parameter and receives a user-entered word via function input. The rhyme function will have to compare the word entered by the user with those present in the past list, looking for rhymes, understood as words whose last 3 letters are the same as the word entered by the user.
13. Write a function sell_books(), which helps in managing the sale of books in a bookstore: (i) Check if the requested book is present on the shelves of the bookshop (ii) If the book is present, decrease the number of copies (possibly removing the title) and inform us that the sale was successful (iii) If the book is not available, it is placed on a list of books to order and we are notified that the sale was unsuccessful
14. Write a "finder" function that scans a given system path for pdf-type files via the os module. The function must have the following characteristics: (i) The path provided must first be validated, as it must lead to an existing folder (ii) The function should provide a list of pdf files (with/relative/path) as they are found (iii) Finally, the function must output the total number of .pdf files found during the scan.
15. Write a function to which a word is passed and recognizes whether it is a palindrome (words that are read the same even backwards) or not.
16. The Caesar Cipher is an encryption algorithm that consists of moving each letter a certain amount of places in the alphabet. To use it, you choose a key that represents the number of places each letter of the alphabet will be moved: for example, if you choose a key of 3, the letter A will become D, the letter B will become E, and so on. To decipher a message encrypted with the Caesar cipher you need to know the key used and move each letter backwards by a number of places corresponding to the key. Write a function that takes a string and a number as arguments and applies the Caesar Cipher to the string by moving as many positions in the alphabet as the number says.
17. Write a program that, given the two lists of numbers below, create the matrix of their products and print the result:
    v1: 1,2,3,4,5
    v2: 6,7,8,9,10
    mat:

1*6 1*7 1*8 ...
2*6 2*7 2*8 ...
...

18. Write a program that, given two input matrices, prints a new matrix containing the sum of the i-th elements only if they are even, otherwise the cube if they are odd
19. Implements the diag function, which given an n x n matrix as a list of lists, RETURNS a list containing the elements of the diagonal (from the top left to the bottom right corner)
20. Write a program that calculates the Euclidean distance between two vectors of floating point numbers entered by the user

Python language: Advanced Concepts (to be explored further)

- **Errors and Exceptions**: https://docs.python.org/3/tutorial/errors.html (https://docs.python.org/3/tutorial/errors.html)
- **Classes**: (https://docs.python.org/3/tutorial/classes.html)
- **Brief Tour of the Standard Library**: https://docs.python.org/3/tutorial/stdlib.html (https://docs.python.org/3/tutorial/stdlib.html)
- **Brief Tour of the Standard Library — Part II**: https://docs.python.org/3/tutorial/stdlib2.html (https://docs.python.org/3/tutorial/stdlib2.html)
- **Numpy library**: https://www.w3schools.com/python/numpy/numpy_intro.asp

References

https://docs.python.org/3/tutorial/index.html (https://docs.python.org/3/tutorial/index.html)