Programming with Python

Stefan Güttel, guettel.com (http://guettel.com)

Contents:

Data structures

So far, we have seen how to work with integers, floating point numbers, and Boolean values. However, there is often a need to work with more complex values like lists, sets, and various other collections of data. The joint name for these is *data structures* or *compound data types*.

The most common data structure is a list.

Lists

As the name sugests, a list is a list of values. Each value has its index (a zero-based position in the list) and can appear in the list as many times as we want.

A list is defined as a comma-separated list of values, enclosed in square brackets:

In [1]:

```
x = [17, 19, 23, 29, 31]
```

The list of values may end with a comma (for lists as well as the other structures in this lecture). For example, this is equivalent to x above:

In [2]:

```
x = [17, 19, 23, 29, 31,]
```

This is convenient when a list contains items with longer definitions, so each is put in its own line for better readability, as well as ease of cut/copy/paste. For example:

In [3]:

```
marx_bros = [
    "Chico (Leonard)",
    "Harpo (Adolph/Arthur)",
    "Groucho (Julius Henry)",
    "Gummo (Milton)",
    "Zeppo (Herbert Manfred)",
]
```

An empty list can be given by

In [4]:

```
x = []
```

but it is preferred to do it like this, for better code readability:

In [5]:

```
x = list()
```

A list in Python can be printed using a simple call to `print`:

In [6]:

```
x = [17, 19, 23, 29, 31, 37, 41, 43, 47, 53]
print(x)
```

```
[17, 19, 23, 29, 31, 37, 41, 43, 47, 53]
```

To get the list elements, we use indices. The first element has index 0 (zero), the second one has index 1, etc. Negative indices count from the end of the list, with `-1` denoting the last element, `-2` denoting the one before it, and so on.

Indexing a nonexistent list element will result in an error.

In [7]:

```
print("The list:", x)
print("The first element of the list x:", x[0])
print("The second element of the list x:", x[1])
print("The last element of the list x:", x[-1])
print("The next to last element of the list x:", x[-2])
print("The nonexistent element of the list x:", x[1719])
```

```
The list: [17, 19, 23, 29, 31, 37, 41, 43, 47, 53]
The first element of the list x: 17
The second element of the list x: 19
The last element of the list x: 53
The next to last element of the list x: 47

---------------------------------------------------------------------
-------
IndexError                                Traceback (most recent cal
l last)
<ipython-input-7-804389ab6b93> in <module>()
      4 print("The last element of the list x:", x[-1])
      5 print("The next to last element of the list x:", x[-2])
----> 6 print("The nonexistent element of the list x:", x[1719])

IndexError: list index out of range
```

We can also slice a list:

In [ ]:

```
print("The list:", x)
print("All elements except the first two (the one with index 2 and those after i
t):", x[2:])
print("Elements with indices 2, 3, and 4 (indices 2 through 5, but excluding 5):"
, x[2:5])
print("The first 3 elements (elements from start up to the one indexed by 3, but
 without it):", x[:3])
print("A copy of the list (all of its elements):", x[:])
print("x[3], x[5], x[7]:", x[3:8:2])
print("All elements with even indices:", x[::2])
print("All elements with odd indices:", x[1::2])
```

Changing the elements of a list

The same indexing that we have used to read the elements of a list can be used to change them:

In [ ]:

```
x = [17, 19, 23, 29, 31]
print("The list:      ", x)
x[3] = 13
print("The list:      ", x, "      (29 -> 13)")
x[-1] = 11
print("The list:      ", x, "      (31 -> 11)")
x[1:3] = [2, 3, 5, 7]
print("The list:      ", x, "  ([19, 23] -> [2, 3, 5, 7])")
x[:] = []
print("A cleared list:", x, "                        (all the elements replaced w
ith an empty list)")
```

Warning: Be very careful with list assignments:

- `y = x` $\Longrightarrow$ y becomes **the same list** as x;
- `z = x[:]` $\Longrightarrow$ z becomes **a copy** of x.

Observe the difference:

In [ ]:

```
x = [17, 19, 23]
y = x
z = x[:]

print("Before the change:")
print("  x =", x)
print("  y =", y)
print("  z =", z)

y[1] = 13
z[2] = 11

print("After the change:")
print("  x =", x)
print("  y =", y)
print("  z =", z)
```

The `x[:]` makes a so-called *shallow copy* (equivalent to `x.copy()`), which is not exactly what one would consider a "copy". For a "real" copy, one needs to use a *deep copy*. To learn more about this, read the on-line documentation on copy module (https://docs.python.org/3/library/copy.html).

For the same reason that the above change of `y[1]` also changed `x[1]`, changing a list parameter inside a function will result in the change of the argument:

In [ ]:

```
def f(a, b, c):
    a = 19
    b = [2, 3, 5]
    c[0] = 11

x = 17
y = [17, 19, 23]
z = [29, 31, 37]

print("Initial values:")
print("  x =", x)
print("  y =", y)
print("  z =", z)

f(x, y, z)

print("After a call to the function `f`:")
print("  x =", x)
print("  y =", y)
print("  z =", z)
```

The length of a list

Getting the length of a list is easy:

In [ ]:

```
x = [17, 19, 23, 29, 31]

print("Before change:")
print("  x =", x)
print("  Length of x:", len(x))

x[1:3] = [2]

print("After change:")
print("  x =", x)
print("  Length of x:", len(x))
```

Appending a list

We can add a single element to a list:

In [ ]:

```
x = [17, 19, 23]
print("Before appending:", x)
x.append(29)
print("After appending: ", x)
```

Warning: `append` accepts only one argument. If it gets more, an error occurs:

In [ ]:

```
x.append(31, 37)
```

Also, if we `append` a list, it will be appended as a single element:

In [ ]:

```
x.append([31, 37])
print("After appending: ", x)
```

To do the latter, i.e., to append more than one element to the list, we need to *extend* it:

In [ ]:

```
x = [17, 19, 23]
print("Before extending:", x)
x.extend([29,31])
print("After extending: ", x)
```

One can also create a new extended list using a + operator:

In [ ]:

```
x = [17, 19, 23]
print("x =", x)
y = x + [29,31]
print("y =", y)
```

We can also insert elements to the beginning or somewhere in the middle of a list:

In [ ]:

```
x = [17, 19, 23]
print("Before inserting:                              ", x)
x.insert(0, 13)
print("After inserting to the beginning:              ", x)
x.insert(2, 11)
print("After inserting to the position with the index 2:", x)
```

Like `append`, `insert` takes a single element that it inserts in the list.

However, inserting a list in the middle of an existing list is not a problem: we put the new list in place of the sublist between, for example, indices 2 and 2, excluding the last one (so, a sublist of a zero legth, starting at index 2):

In [ ]:

```
x = [17, 19, 23]
print("Before inserting:          ", x)
x[2:2] = [2, 3, 5, 7]
print("After inserting at index 2:", x)
```

Or, maybe a bit easier to read, we can take the elements up to (but excluding) index 2, append the new list, and then append that with the elements of the original list starting at index 2:

In [ ]:

```
x = [17, 19, 23]
print("Before inserting:          ", x)
x = x[:2] + [2, 3, 5, 7] + x[2:]
print("After inserting at index 2:", x)
```

This method is practical for creating a new list, i.e.,

```
new_list = old_list[:2] + [2, 3, 5, 7] + old_list[2:]
```

Like using the + operator to concatenate a list, we can use * to create a list made of concatenated copies of a list:

In [ ]:

```
print([ 17, 19, 23 ] * 3)
```

However, be careful when using this with list of lists:

In [ ]:

```
x = [ [17, 19] ] * 5
print(x)
x[0][1] = 23
print(x)
```

You probably wanted this:

In [ ]:

```
x = [ [17, 19] for _ in range(5) ]
print(x)
x[0][1] = 23
print(x)
```

Why did

```
x = [ [17, 19] ] * 5
```

not work as expected?

Types of the elements

In strongly typed languages the elements of a list have to be of a same type. However, in Python, this is not the case:

In [ ]:

```
x = [17, 1.7, [23, 29]]
for i in range(len(x)):
    print("x[" + str(i) + "] =", x[i])
    print("  type(x[" + str(i) + "]) =", type(x[i]))
print("x[2][0] =", x[2][0])
print("  type(x[2][0]) =", type(x[2][0]))
```

Notice the indexing of the elements in a list that is an element of another list:

- `x` is a list.
- `x[2]` is the third element of the list `x`; that element is, in our example, another list.
- `x[2][0]` is the first element of the list `x[2]`.

Of course, we cannot do that for elements of the original list that are not indexable. For example, we cannot access `x[1][1]` because `x[1]` is a real number that is not *subscriptable* (i.e., it cannot be indexed):

In [ ]:

```
print(x[1][1])
```

Reversing a list

How can we reverse the order of the list's elements?

The Python way:

In [ ]:

```
x = [17, 19, 23]
y = list(reversed(x))
print(y)
```

The function `reversed` returns its argument in a reverse order. However, the return value is not a new list, but something called *iterator*, which is the same object that `range` returns. For that reason, we need to print `list(reversed(x))`, but we don't need the `list(...)` conversion if we just want to use it in a `for`-loop. For example:

In [8]:

```
print("x =", x)
print("Reversed, one by one element:")
for i in reversed(x):
    print(i)
```

```
x = [17, 19, 23, 29, 31, 37, 41, 43, 47, 53]
Reversed, one by one element:
53
47
43
41
37
31
29
23
19
17
```

We can also reverse the existing list without creating a new one (so called *in-place reversing*):

In [9]:

```
x = [17, 19, 23]
print("Before reversing:", x)
x.reverse()
print("After reversing: ", x)
```

```
Before reversing: [17, 19, 23]
After reversing:  [23, 19, 17]
```

This is faster and uses less memory than creating a new list and then assigning it to the same variable (`x = reversed(x)`).

Sorting

Sorting is the process of rearranging a list so that its elements come in a certain order. For example, a list `[17, 13, 19, 11, 23]` can be sorted in many ways (i.e., by many criteria):

- ascending by the value: `[11, 13, 17, 19, 23]`,
- descending by the value: `[23, 19, 17, 13, 11]`,
- ascending by the sum of the digits: `[11, 13, 23, 17, 19]` (because the sums of the digits are `[2, 4, 5, 8, 10]`,
  etc.

When we talk about sorting without mentioning a criterion, a sorting by value is implied. If we omit the direction (ascending or descending), ascending is assumed. So, "sort a list" would mean "sort a list ascendingly by the value".

A question: Why do we sort the data?

Basic sorting in Python is simple. First, we define the function `digit_sum` from the previous lecture, in order to be able to sort our list by the sum of the digits:

In [10]:

```
def digit_sum(n, d=1, r=0):
    """
    ...
    """

    # Initializations
    res = 0 # Result (do not use "sum", because there is a built-in function with
that name)
    n = abs(n) # Lose the sign

    while n > 0:
        digit = n % 10
        if digit % d == r:
            res += digit
        n //= 10

    return res
```

Now, we can sort by all three criteria:

In [11]:

```
x = [17, 13, 19, 11, 23]
print("The original list:                ", x)
print("Ascending by the value:           ", sorted(x))
print("Descending by the value:          ", list(reversed(sorted(x))))
print("or:                               ", sorted(x, key=lambda t: -t))
print("or (the best approach):           ", sorted(x, reverse=True))
print("Ascending by the sum of the digits:", sorted(x, key=lambda t: digit_sum(t
)))
print("or:                               ", sorted(x, key=digit_sum))
```

```
The original list:                [17, 13, 19, 11, 23]
Ascending by the value:           [11, 13, 17, 19, 23]
Descending by the value:          [23, 19, 17, 13, 11]
or:                               [23, 19, 17, 13, 11]
or (the best approach):           [23, 19, 17, 13, 11]
Ascending by the sum of the digits: [11, 13, 23, 17, 19]
or:                               [11, 13, 23, 17, 19]
```

The function `sorted` creates a new list with the same elements as the function's argument, but in a sorted order. There is also an in-line version for sorting a list without creating a new one:

In [12]:

```
x = [17, 13, 19, 11, 23]
print("The original list:                ", x)
x.sort()
print("Ascending by the value:           ", x)
x.sort()
x.reverse()
print("Descending by the value:          ", x)
x.sort(reverse=True)
print("or:                               ", x)
x.sort(key=lambda t: digit_sum(t))
print("Ascending by the sum of the digits:", x)
x.sort(key=digit_sum)
print("or:                               ", x)
```

```
The original list:                [17, 13, 19, 11, 23]
Ascending by the value:           [11, 13, 17, 19, 23]
Descending by the value:          [23, 19, 17, 13, 11]
or:                               [23, 19, 17, 13, 11]
Ascending by the sum of the digits: [11, 13, 23, 17, 19]
or:                               [11, 13, 23, 17, 19]
```

Membership

Does some element exist in a list?

In [13]:

```
lst = [17, 13, 23, 17, 19]
print("Is 17 in lst?", 17 in lst)
print("Is 17 not in lst?", 17 not in lst)
print("Is 19 in lst?", 19 in lst)
print("Is 19 not in lst?", 19 not in lst)
print("Is 11 in lst?", 11 in lst)
print("Is 11 not in lst?", 11 not in lst)
```

```
Is 17 in lst? True
Is 17 not in lst? False
Is 19 in lst? True
Is 19 not in lst? False
Is 11 in lst? False
Is 11 not in lst? True
```

So, we can check if some element is in the list, but this does not tell us **where** it is, nor **how many** of the list elements are equal to it.

Finding a list element

One of the more useful list operations is finding an element (any one or all of them) with a given value or some characteristic. Python has a simple way to find one element with a given value:

In [14]:

```
x = [17, 13, 11, 19, 7, 23, 17, 2, 31, 29, 3, 5]
print("Index of the first 17 in the list:", x.index(17))
print("Index of the first 11 in the list:", x.index(11))
print("Index of the first 53 in the list:", x.index(53))
```

```
Index of the first 17 in the list: 0
Index of the first 11 in the list: 2

---------------------------------------------------------------------
-------
ValueError                                Traceback (most recent cal
l last)
<ipython-input-14-ac0e925f2f71> in <module>()
      2 print("Index of the first 17 in the list:", x.index(17))
      3 print("Index of the first 11 in the list:", x.index(11))
----> 4 print("Index of the first 53 in the list:", x.index(53))

ValueError: 53 is not in list
```

As you can see, if an element is not found, an error occurs. While this may seem strange, it is a very natural thing in modern languages, as there is a simple way to "catch" and "handle" errors like that. We will do this on the lectures in two weeks time.

Python also makes it easy to find how many times a certain element appears in a list:

In [ ]:

```
x = [17, 13, 11, 19, 7, 23, 17, 2, 31, 29, 3, 5]
print("Number 17 appears", x.count(17), "times in the list.")
print("Number 11 appears", x.count(11), "times in the list.")
print("Number 53 appears", x.count(53), "times in the list.")
```

As you can see, one can easily use `count` function to find out if an element exists in a list:

In [ ]:

```
x = [17, 13, 11, 19, 7, 23, 17, 2, 31, 29, 3, 5]

if x.count(17):
    print("Index of the first 17 in the list:", x.index(17))
else:
    print("Number 17 is not in the list.")

if x.count(11):
    print("Index of the first 11 in the list:", x.index(11))
else:
    print("Number 11 is not in the list.")

if x.count(53):
    print("Index of the first 53 in the list:", x.index(53))
else:
    print("Number 53 is not in the list.")
```

A somewhat better approach is to just check if an element exists in the list (instead of counting how many times it appears in it):

In [ ]:

```
x = [17, 13, 11, 19, 7, 23, 17, 2, 31, 29, 3, 5]

if 17 in x:
    print("Index of the first 17 in the list:", x.index(17))
else:
    print("Number 17 is not in the list.")

if 11 in x:
    print("Index of the first 11 in the list:", x.index(11))
else:
    print("Number 11 is not in the list.")

if 53 in x:
    print("Index of the first 53 in the list:", x.index(53))
else:
    print("Number 53 is not in the list.")
```

Without using Python specific functions `count` and `index`, we can search for a list element by testing each of them one by one:

In [ ]:

```
x = [17, 13, 11, 19, 7, 23, 17, 2, 31, 29, 3, 5]

def find(L, x):
    """
    Returns index of `x` in `L`, or `None` if `x` doesn't exist in `L`.
    """
    i = 0
    for el in L:
        if el == x:
            return i
        i += 1
    return None

print("Index of the first 17 in the list:", find(x, 17))
print("Index of the first 11 in the list:", find(x, 11))
print("Index of the first 53 in the list:", find(x, 53))
```

This is called **sequential** or **linear** search (http://en.wikipedia.org/wiki/Linear_search) and it can be easily adapted to different criteria (for example "in the list `L` find the first/last number with the sum of digits equal to 17") and it can be also adapted to find **all** the instances of an element.

Example: Print **all** the prime numbers in a given list.

Let us first define a function that will check if a given number is a prime:

In [ ]:

```
def is_prime(n):
    """
    Returns `True` if `n` is prime, and `False` otherwise.
    """
    if n < 2:
        return False
    for i in range(2, n):
        if n % i == 0:
            return False
    return True
```

A more Pythonic way:

In [ ]:

```
def is_prime(n):
    """
    Returns `True` if `n` is prime, and `False` otherwise.
    """
    return (n >= 2) and all(n % i for i in range(2, n))
```

If we were given a list of numbers written on a piece of paper and asked to somehow mark the primes, how would we do it?

Probably like this:

> Look at the first one. If it is prime, mark it.
> Look at the second one. If it is prime, mark it.
> ...

We can do the same here. For simplicity, we will just provide a list of numbers for now (usually, we would have loaded the numbers).

In [ ]:

```
lst = [-17, 13, 26, 17, -1, 0, 1, 2, 165351462, 24631669, 19]
for x in lst:
    if is_prime(x):
        print(x)
```

There are faster ways to check if a number is prime. We shall see some of those in the later lectures.

Example: Print **indices** of **all** the prime numbers in a given list.

In [ ]:

```
lst = [-17, 13, 26, 17, -1, 0, 1, 2, 165351462, 24631669, 19]
for i in range(len(lst)):
    if is_prime(lst[i]):
        print(i)
```

or, a more Pythonic way:

In [ ]:

```
lst = [-17, 13, 26, 17, -1, 0, 1, 2, 165351462, 24631669, 19]
for i, x in enumerate(lst):
    if is_prime(x):
        print(i)
```

Example: Print **indices and values** of **all** the prime numbers in a given list.

In [ ]:

```
lst = [-17, 13, 26, 17, -1, 0, 1, 2, 165351462, 24631669, 19]
for i in range(len(lst)):
    if is_prime(lst[i]):
        print(str(lst[i]) + ", at index", i)
```

or, a more Pythonic way:

In [ ]:

```
lst = [-17, 13, 26, 17, -1, 0, 1, 2, 165351462, 24631669, 19]
for i, x in enumerate(lst):
    if is_prime(x):
        print(str(x) + ", at index", i)
```

Exercises

Try to write a program that prints value/index/both of:

1. only the first prime element; for the above list `lst`, that would be 13;
2. only the second prime element; for the above list `lst`, that would be 17;
3. prime elements with odd indices; for the above list `lst`, those would be all except 19 (which is at an even index 10);
4. all elements with prime indices; for the above list `lst`, those would be 26 (index 2), 17 (index 3), 0 (index 5), and 2 (index 7);
5. every second prime in the list, starting from the first one; for the above list `lst`, those would be 13 (the first one), 2 (the third one), and 19 (the fifth one).

Inputing a list

It is convenient to define the value of a list in program as we are writing it, so that it doesn't have to be loaded with each test. However, a program with the built-in input values is not very useful. Like integers, reals, and strings, lists also have to be loaded from the user.

While there are various Pythonic ways to input a list, depending on what kind of input is requested, we shall show more general ways that should work with all kinds of inputs (and in most of the modern languages that suport data structures similar to Python lists).

Example: Load a natural number `n`, then load a list of `n` numbers, sort it, and print it.

In [ ]:

```
lst = []  # Create an empty list
n = int(input("How many numbers do you want to sort? "))
while n <= 0:
    n = int(input("Please type a positive integer: "))
for _ in range(n):
    lst.append(float(input("Input a real number: ")))
print("The sorted list:", sorted(lst))
```

Example: Input integers until -17 is loaded, then print them all (**except** -17) sorted descendingly.

In [ ]:

```
lst = []  # Create an empty list
not_done = True
while not_done:
    x = int(input("Please type an integer: "))
    if x == -17:
        not_done = False
    else:
        lst.append(x)
print("The sorted list:", sorted(lst, reverse=True))
```

Example: Input integers until -17 is loaded, then print them all (**including** -17) sorted descendingly.

In [ ]:

```
lst = []  # Create an empty list
not_done = True
while not_done:
    x = int(input("Please type an integer: "))
    lst.append(x)
    if x == -17:
        not_done = False
print("The sorted list:", sorted(lst, reverse=True))
```

Note: There is a slightly better way to do this, which we shall cover in two weeks.

Deleting elements from a list

We have already seen one way to remove some element(s) from the list: replacing them with an empty list. For example:

In [ ]:

```
x = [17, 19, 23, 29, 31, 37]
print("The list:                                    ", x)
x[-2:] = []
print("The list without its last two elements:      ", x)
x[2:3] = []
print("The list without its third element:          ", x)
x[1] = []
print('The list with wrongly "deleted" second element:', x)
x[1] = None
print('The list with wrongly "deleted" second element:', x)
```

However, it is much better to use the dedicated `del` (https://docs.python.org/3/reference/simple_stmts.html#the-del-statement) statement, as it makes the code easier to read and the intention more obvious:

In [ ]:

```
x = [17, 19, 23, 29, 31, 37]
print("The list:                                    ", x)
del x[-2:]
print("The list without its last two elements:      ", x)
del x[2]
print("The list without its third element:          ", x)
del x[:]
print("An empty list:                               ", x)
del x
print("Completely removed variable:                 ", x)
```

Apart from using somewhat confusing `del x[:]`, we can also empty the list like this:

In [ ]:

```
x = [17, 19, 23, 29, 31, 37]
print("The list:                                    ", x)
x.clear()
print("An empty list:                               ", x)
```

Note that this is different from `x = []` which creates a new empty list and assigns it to `x` instead of clearing the elements of `x`. While the result will usually be the same, there is a difference:

In [ ]:

```
def clear_through_assignment(x):
    """Clear the list `x` using the assignment `x = []`."""
    x = []

def clear_through_clear(x):
    """Clear the list `x` using `x.clear()`."""
    x.clear()

def clear_through_del(x):
    """Clear the list `x` using `del x[:]`."""
    del x[:]

x = [17, 19, 23, 29, 31, 37]
print("The list:                ", x)
clear_through_assignment(x)
print("Whoops! Nothing happened:", x)
clear_through_clear(x)
print("No more elements:        ", x)

x = [17, 19, 23, 29, 31, 37]
print("The list:                ", x)
clear_through_del(x)
print("No more elements:        ", x)
```

Wrap it up

Example: Write a program that loads integers until it loads a zero (which is not included in the subsequent computations), and then writes:

1. the arithmetic mean of all the loaded numbers,
2. the median (http://en.wikipedia.org/wiki/Median) of all the loaded numbers,
3. all the loaded numbers bigger than or equal to the median, in their original order.

Arithmetic mean should be easy enough to compute. It is simply the sum of the loaded numbers divided by how many of them there are.

Notice that if we load no numbers, the arithmetic mean is $0/0$, which is undefined. We have to make sure that the program behaves properly in this case.

The median of a list is the numerical value separating the higher half of data from the lower half. For the finite data, this simply means sorting the list and taking the middle element (if the list has an odd length) or an arithmetic mean of the two middle elements (if the list has an even length).

Printing the numbers bigger than some given number is a simple matter of running through the list and printing those that satisfy an appropriate condition. However, there are two things to consider:

1. The original order of the list elements is required.
   This means that we need to preserve the original ordering. In other words, we must not sort the existing list to find the mean, but we need to create a new sorted list for that purpose, preserving the original one.
2. Printing one number in each line is not always desirable.
   We shall create a new list of numbers that are bigger than the mean, and then print it.

In [15]:

```
import sys

# Input:
lst = []  # Create an empty list
not_done = True
while not_done:
    x = int(input("Please type an integer: "))
    if x == 0:
        not_done = False
    else:
        lst.append(x)

# Check that the list has a non-zero length
if len(lst) == 0:
    print("Cowardly refusing to work with means and medians of an empty list!")
    sys.exit(1)

# We'll need this several times, so better put it in a variable
list_length = len(lst)

# Find the arithmetic mean
mean = 0
for x in lst:
    mean += x
mean /= list_length

# Find the median
sorted_list = sorted(lst)
median_index = list_length // 2
if list_length % 2 == 0:
    # The length of the list is even
    median = (sorted_list[median_index - 1] + sorted_list[median_index]) / 2
else:
    # The length of the list is odd
    median = sorted_list[median_index]

# Create the list of elements bigger than or equal to the median
list_bigger = []
for x in lst:
    if x >= median:
        list_bigger.append(x)

# Print the results
print("Arithmetic mean:", mean)
print("Median:", median)
print("Elements >= median:", list_bigger)
```

```
Please type an integer: 17
Please type an integer: -19
Please type an integer: 23
Please type an integer: 0
Arithmetic mean: 7.0
Median: 17
Elements >= median: [17, 23]
```

Note: `sys.exit(1)` is used to stop a program's execution (it doesn't work in IPython). The number in the parenthesis can be any integer and it should be zero if the program successfully did what it had to do, or a non-zero value unique for each of the errors for which the program was stopped. Here, we used `1` to describe "the list is empty, hence we cannot work with it".

In order for `sys.exit(...)` to work, we need `import sys` somewhere before it (almost always at the beginning of the program).

A more Pythonic solution:

In [16]:

```
import sys

# Input:
lst = []  # Create an empty list
while True:
    x = int(input("Please type an integer: "))
    if x == 0:
        break
    lst.append(x)

# Check that the list has a non-zero length
if len(lst) == 0:
    print("Cowardly refusing to work with means and medians of an empty list!")
    sys.exit(1)

# We'll need this several times, so better put it in a variable
list_length = len(lst)

# Find the median
sorted_list = sorted(lst)
median_index = list_length // 2
median = sorted_list[median_index] if list_length % 2 else \
    sum(sorted_list[median_index-1:median_index+1]) / 2

# Print the results
print("Arithmetic mean:", sum(lst) / list_length)
print("Median:", median)
print("Elements >= median:", [x for x in lst if x >= median])
```

```
Please type an integer: 17
Please type an integer: -19
Please type an integer: 23
Please type an integer: 0
Arithmetic mean: 7.0
Median: 17
Elements >= median: [17, 23]
```

or, using NumPy (a widely used numerical library for Python):

In [17]:

```
import numpy as np

# Input:
lst = []  # Create an empty list
while True:
    x = int(input("Please type an integer: "))
    if x == 0:
        break
    lst.append(x)

if len(lst) == 0:
    print("Cowardly refusing to work with means and medians of an empty list!")
else:
    median = np.median(lst)
    print("Arithmetic mean:", sum(lst) / len(lst))
    print("Median:", median)
    print("Elements >= median:", [x for x in lst if x >= median])
```

```
Please type an integer: 17
Please type an integer: -19
Please type an integer: 23
Please type an integer: 0
Arithmetic mean: 7.0
Median: 17.0
Elements >= median: [17, 23]
```

or, using `statistics` (a Python library of statistical functions, available since Python 3.4):

In [ ]:

```
import statistics

# Input:
lst = []  # Create an empty list
while True:
    x = int(input("Please type an integer: "))
    if x == 0:
        break
    lst.append(x)

try:
    median = statistics.median(lst)
except StatisticsError:
    print("Cowardly refusing to work with means and medians of an empty list!")
else:
    print("Arithmetic mean:", statistics.mean(lst))
    print("Median:", median)
    print("Elements >= median:", [x for x in lst if x >= median])
```

Iterables

Lists are one type of Python's *iterables* -- objects that contain some members that can be accessed one at a time. In other words, any variable `var` contains an iterable if you can iterate through it:

```
for some_other_var in var:
    ...
```

While each iterable has its own specific characteristics and/or operations, they are generally used in the same manner as lists. Their functionality can be, with more or less effort, emulated by lists (and is often done like that in some of the other languages).

We now bring a short preview of some other iterables that exist in Python.

Other built-in iterables

Python has several built-in data structures other than lists.

Tuples

A tuple (https://docs.python.org/3/library/stdtypes.html#sequence-types-list-tuple-range) is basically an immutable (unchangeable) list. It can be assigned a new value (as any variable), but its elements cannot be changed or deleted, nor can the new ones be added.

A tuple is defined simlarly to a list, but the values are enclosed in parentheses.

In [18]:

```
an_empty_tuple = tuple()
a_less_preferred_empty_tuple = ()
a_tuple = (17, 13, 19, 23)
print("A tuple:                                                ", a_tuple)
print("Its first element:                                      ", a_tuple[0])
print("Its last element:                                       ", a_tuple[-1])
print("Without the first and the last element:                 ", a_tuple[1:-1])
print("A sorted tuple:                                         ", sorted(a_tuple))
another_tuple = (1, 2, 3)
print("Another tuple appended to the first one:                ", a_tuple + anothe
r_tuple)
print("Another tuple inserted in the middle of the first one: ",
      a_tuple[:2] + another_tuple + a_tuple[2:])
print("All the membership stuff works, i.e., is 17 in a tuple:", 17 in a_tuple)
```

```
A tuple:                                                (17, 13, 19,
23)
Its first element:                                      17
Its last element:                                       23
Without the first and the last element:                 (13, 19)
A sorted tuple:                                         [13, 17, 19,
23]
Another tuple appended to the first one:                (17, 13, 19,
23, 1, 2, 3)
Another tuple inserted in the middle of the first one:  (17, 13, 1,
2, 3, 19, 23)
All the membership stuff works, i.e., is 17 in a tuple: True
```

Notice that

```
a_tuple.extend(another_tuple)
```

and

```
a_tuple.insert(position, another_tuple)
```

would not work because `a_tuple` is a tuple and cannot be changed. For the same reason, `reverse` and `sort` wouldn't work either, but `sorted` and `reversed` would. Furthermore, `sorted(a_tuple)` returns a (new) **list** with the same elements that `a_tuple` had, ordered as required.

However, we can insert a tuple's elements into a list:

In [19]:

```
x = [17, 19, 23]  # a list
y = (13, 11, 7)  # a tuple
x.extend(y)  # append the tuple y to the list x
print(x)
```

```
[17, 19, 23, 13, 11, 7]
```

Since y is a tuple (i.e., immutable), the opposite won't work:

In [20]:

```
x = [17, 19, 23]  # a list
y = (13, 11, 7)  # a tuple
y.extend(x)  # append the list x to the tuple y
print(y)
```

```
---------------------------------------------------------------------
-------
AttributeError                            Traceback (most recent cal
l last)
<ipython-input-20-9e6a4ae41591> in <module>()
      1 x = [17, 19, 23]  # a list
      2 y = (13, 11, 7)  # a tuple
----> 3 y.extend(x)  # append the list x to the tuple y
      4 print(y)

AttributeError: 'tuple' object has no attribute 'extend'
```

Note: Be careful when creating single-element tuples:

In [21]:

```
x = (1)
y = (1,)
print("An integer:              {:4} (type: {})".format(x, type(x)))
print("A single-element tuple: {:4} (type: {})".format(y, type(y)))
```

```
An integer:                1 (type: <class 'int'>)
A single-element tuple: (1,) (type: <class 'tuple'>)
```

Tuples are often used for multiple assignments. For example, a traditional way to swap two variables is:

In [22]:

```
x = input("x = ")
y = input("y = ")

temp = x
x = y
y = temp

print("x =", x)
print("y =", y)
```

```
x = 17
y = 19
x = 19
y = 17
```

However, using tuples, it boils down to:

In [23]:

```
x = input("x = ")
y = input("y = ")

(x, y) = (y, x)

print("x =", x)
print("y =", y)
```

```
x = 17
y = 19
x = 19
y = 17
```

The parentheses are not needed:

In [24]:

```
x = input("x = ")
y = input("y = ")

x, y = y, x

print("x =", x)
print("y =", y)
```

```
x = 17
y = 19
x = 19
y = 17
```

Another common use for them is returning more than one value from a function. For example:

In [25]:

```
def modiv(x, y):
    """Returns mod and div together, i.e., x//y and x%y."""
    return x // y, x % y

a, b = modiv(19, 17)
print("19 // 17 =", a)
print("19 % 17 = ", b)
```

```
19 // 17 = 1
19 % 17 =  2
```

To ignore one or more of the return values, one can use a "throwaway variable" _:

In [26]:

```
a, _ = modiv(19, 17)
print("19 // 17 =", a)
```

```
19 // 17 = 1
```

Note that this is **not** the same as

In [27]:

```
a = modiv(19, 17)
print("19 // 17 =", a)
```

```
19 // 17 = (1, 2)
```

The above can also be done with lists (although it almost never is).

So, why would anyone want a list with reduced functionality (i.e., without the ability to change)?

Traversing both the indices and the elements of an iterator

Sometimes, using "`for x in L`" or "`for i in range(len(L))`" is not enough because we need both the index and the value of each element. Of course, we can always do

```
for i in range(len(L)):
    print("index:", i)
    print("value:", L[i])
```

but there is a more Pythonic (and easier way): the function `enumerate` (https://docs.python.org/3/library/functions.html#enumerate) which returns an iterator of two-member tuples, each containing the index and the value of the given iterator.

For example:

In [28]:

```
L = [ 17, 19, 23 ]
print(list(enumerate(L)))
```

```
[(0, 17), (1, 19), (2, 23)]
```

A more common way to use this is:

In [29]:

```
L = [ 17, 19, 23 ]
for idx, el in enumerate(L):
    print("L[{}] = {}".format(idx, el))
```

```
L[0] = 17
L[1] = 19
L[2] = 23
```

We can also have the index counter start from a value other than zero. For example:

In [30]:

```
L = [ 17, 19, 23 ]
for idx, el in enumerate(L, start=1):
    print("Position #{} in L has the value {}.".format(idx, el))
```

```
Position #1 in L has the value 17.
Position #2 in L has the value 19.
Position #3 in L has the value 23.
```

The argument's name "`start`" can be ommited, but it's better to leave it for clarity.

Sets

A set is like a list in which all elements are unique and they have no ordering defined (hence, we cannot index its elements). Sets are changed by using the standard sets operations, and the membership is checked in the same manner that it was check with lists and tuples.

In [31]:

```
an_empty_set = set()
this_is_NOT_a_set = {}
a = {17, 13, 19, 23}
b = {11, 17, 19, 31}
c = {17, 19, 13}
print("a:                          ", a)
print("b:                          ", b)
print("The union of a and b:       ", a.union(b))
print("  or:                       ", a | b)
print("The intersection of a and b:", a.intersection(b))
print("  or:                       ", a & b)
print("Difference a \\ b:           ", a.difference(b))
print("  or:                       ", a - b)
print("Symmetric difference a △ b: ", a.symmetric_difference(b))
print("  or:                       ", a ^ b)
print("Is 17 in the set a union b? ", 17 in a|b)
print("Is c a subset of a?         ", c.issubset(a))
print("  or:                       ", c <= a)
print("Is c a strict subset of a?  ", c < a)
print("Is c a superset of a?       ", c.issuperset(a))
print("  or:                       ", c >= a)
print("Is c a strict superset of a?", c > a)
```

```
a:                          {17, 19, 13, 23}
b:                          {11, 17, 19, 31}
The union of a and b:       {11, 13, 17, 19, 23, 31}
  or:                       {11, 13, 17, 19, 23, 31}
The intersection of a and b: {17, 19}
  or:                       {17, 19}
Difference a \ b:           {13, 23}
  or:                       {13, 23}
Symmetric difference a △ b:  {23, 11, 13, 31}
  or:                       {23, 11, 13, 31}
Is 17 in the set a union b?  True
Is c a subset of a?          True
  or:                       True
Is c a strict subset of a?   True
Is c a superset of a?        False
  or:                       False
Is c a strict superset of a? False
```

Sets are muttable because they can be changed using their functions `update`, `intersection_update`, `difference_update`, and `symmetric_difference_update`. Immutable version of a set is called `frozenset`. You can read more about both in the official Python documentation for `set` and `frozenset` (https://docs.python.org/3/library/stdtypes.html#set).

Dictionaries

A dictionary (https://docs.python.org/3/library/stdtypes.html#mapping-types-dict) is like a list that uses indices of various types, not just zero-based integers. These "indices" are called *keys*.

There are many ways to define a dictionary. This is from the official Python documentation:

In [32]:

```
a = dict(one=1, two=2, three=3)
b = {'one': 1, 'two': 2, 'three': 3}
c = dict(zip(['one', 'two', 'three'], [1, 2, 3]))
d = dict([('two', 2), ('one', 1), ('three', 3)])
e = dict({'three': 3, 'one': 1, 'two': 2})
a == b == c == d == e
```

Out[32]:

```
True
```

An empty dictionary should be defined by

In [33]:

```
an_empty_dictionary = dict()
```

even though `an_empty_dictionary = {}` would also work. The latter should be avoided due to ambiguity (it can easily be confused for an empty set).

Accessing all the elements of a dictionary, one by one

We can use `for` to access the elements of a dictionary one by one (in an order that is not predetermined):

In [34]:

```
for key in a:
    print(key, "->", d[key])
```

```
three -> 3
one -> 1
two -> 2
```

or, through keys and values together, using a function `a.items` that returns a dictionary as a list of `(key, value)` tuples:

In [35]:

```
for key, value in a.items():
    print(key, "->", value)
```

```
three -> 3
one -> 1
two -> 2
```

Since the default sort of a list of tuple will first compare the first elements of the tuples, then the second ones, etc, we can easily access the items sorted by their key values (incidentally revealing the original quote by

In [36]:

```
for key, value in sorted(a.items()):
    print(key, "->", value)
```

```
one -> 1
three -> 3
two -> 2
```

Of course, Python does not interpret "one", "two", and "three", so the keys are sorted alphabetically (often called *lexically*).

Printing a dictionary

A dictionary can be printed using `print`:

In [37]:

```
print(a)
```

```
{'three': 3, 'one': 1, 'two': 2}
```

However, this is usually hard to read:

In [38]:

```
d = {
    "xyz": "something",
    "2": 1719,
    1719: [1, 2, [3, "python"], ("a", 13)],
    "xyz": "something else",
    (1,0): "the key is a tuple",
}
print(d)
```

```
{'2': 1719, (1, 0): 'the key is a tuple', 'xyz': 'something else', 1
719: [1, 2, [3, 'python'], ('a', 13)]}
```

There is a nicer way to print a dictionary:

In [39]:

```
from pprint import pprint
pprint(d, indent=4)
```

```
{   1719: [1, 2, [3, 'python'], ('a', 13)],
    '2': 1719,
    'xyz': 'something else',
    (1, 0): 'the key is a tuple'}
```

We can also write our own function:

In [40]:

```
def dict_print(name, d, indent=4):
    """
    Very crude pretty printer for dictionaries

    Parameters:
    `name`:
        The name to be printed on top of the data.
    `d`:
        The dictionary to be printed.
    `indent`:
        Number of spaces to prepend to each of the printed lines.
    """
    print(name + ":")
    for key, value in d.items():
        print(" " * (indent-1), "*", key, "->", value)

dict_print("Dictionary d", d)
```

```
Dictionary d:
    * 2 -> 1719
    * (1, 0) -> the key is a tuple
    * xyz -> something else
    * 1719 -> [1, 2, [3, 'python'], ('a', 13)]
```

Note that the multiple values assigned to the same key will override each other: "`something`" was overriden by "`something else`" because we tried to assign both of them to the key "`xyz`".

As we can see above, the values in a dictonary can be whatever we want. However, the keys cannot be exactly anything. Mutables, like lists, cannot be used as keys:

In [41]:

```
a = {
    "1719": "A string is OK",
    (17, 19): "A tuple is also OK",
    [17, 19]: "A list is not",
}
```

```
---------------------------------------------------------------------
-------
TypeError                                 Traceback (most recent cal
l last)
<ipython-input-41-386b9604e97c> in <module>()
      2     "1719": "A string is OK",
      3     (17, 19): "A tuple is also OK",
----> 4     [17, 19]: "A list is not",
      5 }

TypeError: unhashable type: 'list'
```

Before using the values of a certain type as keys, it is always best to try out if it will actually work, or the code needs to be done differently. If the desired type cannot be used for keys, we can usually fix it by converting what we want to use as keys to a more appropriate data type. For example, lists are trivially converted to tuples.

Of course, when keys or values hold the values of uncomparable types, we cannot sort them (an error will occur):

In [42]:

```
for key, value in sorted(d.items()):
    print(" " * (indent-1), "*", key, "->", value)
```

```
---------------------------------------------------------------------
-------
TypeError                                 Traceback (most recent cal
l last)
<ipython-input-42-18f72b1686aa> in <module>()
----> 1 for key, value in sorted(d.items()):
      2     print(" " * (indent-1), "*", key, "->", value)

TypeError: unorderable types: tuple() < str()
```

Adding, changing, and deleting the data

Much like sets, dictionaries don't have the "first", the "second", etc. element.

Adding new data or changing the existing one is done by using its key:

In [43]:

```
d = {
    "xyz": "something",
    "2": 1719,
    1719: [1, 2, [3, "python"], ("a", 13)],
    "xyz": "something else",
    (1,0): "the key is a tuple",
}

dict_print("Dictionary d", d, indent=4)
d["this is also a key"] = "something added"
d["xyz"] = "something changed"
dict_print("Changed dictionary d", d, indent=4)
```

```
Dictionary d:
    * 2 -> 1719
    * (1, 0) -> the key is a tuple
    * xyz -> something else
    * 1719 -> [1, 2, [3, 'python'], ('a', 13)]
Changed dictionary d:
    * 2 -> 1719
    * (1, 0) -> the key is a tuple
    * this is also a key -> something added
    * xyz -> something changed
    * 1719 -> [1, 2, [3, 'python'], ('a', 13)]
```

To delete the data, we can simply use `del`, like with lists:

In [44]:

```
del d[(1,0)], d["xyz"]
dict_print("Dictionary d with some keys removed", d, indent=4)
```

```
Dictionary d with some keys removed:
    * 2 -> 1719
    * this is also a key -> something added
    * 1719 -> [1, 2, [3, 'python'], ('a', 13)]
```

Using `for` with a dictionary

When `for` is directly used with a dictionary, it gives us the items' **keys** in some order:

In [45]:

```
d = {
    "first": "going once",
    "second": "going twice",
    "third": "sold"
}
for x in d:
    print(x)
```

```
first
second
third
```

This is enough to get the values as well:

In [46]:

```
d = {
    "first": "going once",
    "second": "going twice",
    "third": "sold"
}
for x in d:
    print('d["{}"] = "{}"'.format(x, d[x]))
```

```
d["first"] = "going once"
d["second"] = "going twice"
d["third"] = "sold"
```

However, there is a better way, very much alike `enumerated` call that we have shown above:

In [47]:

```
d = {
    "first": "going once",
    "second": "going twice",
    "third": "sold"
}
for key, value in d.items():
    print('d["{}"] = "{}"'.format(key, value))
```

```
d["first"] = "going once"
d["second"] = "going twice"
d["third"] = "sold"
```

Converting between iterables

Converting between lists, sets, and tuples can be easily done with their namesake functions:

In [1]:

```
lst = [17, 19, 11, 23]
print("List:")
print("  - as a list: ", lst)
print("  - as a tuple:", tuple(lst))
print("  - as a set:  ", set(lst))
tup = (17, 19, 11, 23)
print("Tup:")
print("  - as a list: ", list(tup))
print("  - as a tuple:", tup)
print("  - as a set:  ", set(tup))
st = {17, 19, 11, 23}
print("Set:")
print("  - as a list: ", list(st))
print("  - as a tuple:", tuple(st))
print("  - as a set:  ", st)
```

```
List:
  - as a list:  [17, 19, 11, 23]
  - as a tuple: (17, 19, 11, 23)
  - as a set:   {11, 17, 19, 23}
Tup:
  - as a list:  [17, 19, 11, 23]
  - as a tuple: (17, 19, 11, 23)
  - as a set:   {11, 17, 19, 23}
Set:
  - as a list:  [11, 17, 19, 23]
  - as a tuple: (11, 17, 19, 23)
  - as a set:   {11, 17, 19, 23}
```

In lists and tuples, the keys are assigned in order: 0, 1,...; in sets, they are simply disregarded. However, dictionaries are a bit more complex, as Python cannot just make the keys up.

One way to create a dictionary is from a list, tuple, or a set of two-member lists, tuples, or sets:

In [49]:

```
d = dict([ (17, 19), {"a", "b"}, [13, 11]])
dict_print("d", d)
```

```
d:
    * b -> a
    * 17 -> 19
    * 13 -> 11
```

Notice that sets do not define the order of their elements, so keys and values may end up swapped.

More often we want to create a dictionary from two lists, tuples, or sets, one containing keys and another containing their corresponding values. For this purpose, we use the built-in function `zip` (https://docs.python.org/3/library/functions.html#zip). Of course, the lengths of these two lists must be equal.

In [50]:

```
keys = ["a", "b", (1,0)]
values = (1, {17, 19}, "xyz")
print(list(zip(keys, values)))
dict_print("A dictionary created from a list and a tuple", dict(zip(keys, values
)))
```

```
[('a', 1), ('b', {17, 19}), ((1, 0), 'xyz')]
A dictionary created from a list and a tuple:
    * b -> {17, 19}
    * (1, 0) -> xyz
    * a -> 1
```

Again, sets are unreliable for this purpose, due to the lack of their elements' ordering.