Programming in Python

Lecture 2

Numeral systems.

Real numbers.

Strings

Table of contents

Type float

- **Limited precision.** Use them only if integers are completely impossible to use.
- **Type conversion – float().**
- You can use real numbers together with integers in one expression. **The result will be a float.**
- Type float has **the same set of arithmetic operations** as the integer, except integer division.
- Operation **division (X/Y) always has a real result**, even if X and Y are both integers.

Float. Numbers representations

- Digital Computers use **Binary number system** to represent all types of information inside the computers.

- There are two major approaches to store real numbers (i.e., numbers with fractional component) in modern computing:
  - **Fixed Point Notation.** There are a fixed number of digits after the decimal point: 0.000011
  - **Floating Point Notation.** Allows for a varying number of digits after the decimal point: 11E-6

Unsigned integer		Integer		
Signed integer	Sign	Integer		
Unsigned fixed point		Integer	Fraction	
Signed fixed point	Sign	Integer	Fraction	
Floating point	Sign	Exponent	Sign	Mantissa
Variable length	Sign	Size	Digits	
Unsigned rational		Numerator	Denominator	
Signed rational	Sign	Numerator	Denominator	

Fixed-point representation

- Has **fixed number of bits for integer part and for fractional part**.

Example: Assume number is using 32-bit format which reserve **1 bit for the sign, 15 bits for the integer part and 16 bits for the fractional part**. Then, -43.625 is represented as following:

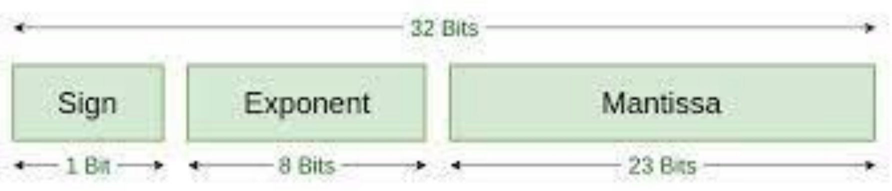
1	000000000101011	1010000000000000
Sign bit	Integer part	Fractional part

Where, in **Sing bit** 0 is used to represent '+' and 1 is used to represent '-'. 000000000101011 is 15-bit binary value for decimal 43 and 1010000000000000 is 16-bit binary value for fractional 0.625.

The advantage of using a fixed-point representation **is performance** and **disadvantage is relatively limited range of values that they can represent.**

Floating point representation

- Does not reserve a specific number of bits for the integer or fractional parts.
- It has two parts: the first - represents **a signed fixed-point number** called **mantissa**. The second part denotes **the position of the decimal point** and is called the **exponent**.
- **Only the Mantissa and the Exponent are physically represented in the memory (including their signs).**



Single Precision
IEEE 754 Floating-Point Standard

Most common for Python – using 64 bits to store type *float*. 1 bit for sign, 52 – mantissa, 11 – exponent.

Floating point representation

Example: Suppose number is using 32-bit format: the 1-bit - sign bit, 8 bits for signed exponent, and 23 bits for the fractional part.

Then −53.5 is normalized as $-53.5=(-110101.1)_2=(-1.101011)*2^5$, which is represented as following below,

Where 00000101 is the 8-bit binary value of exponent value +5.

Note that 8-bit exponent field is used to store integer exponents $-126 \leq n \leq 127$.

1	00000101	10101100000000000000000
Sign bit	Exponent part	Mantissa part

Real numbers

For output a real number in a fixed-point notation use the method ***format():***

```
x= 0.1
print('{0:.25f}'.format(x))
```

0.1000000000000000055511151

Two real numbers are equal if they differ by no more than epsilon:

/X – Y /< Ɛ

```
if 0.1 + 0.2 == 0.3:
   print('All right!')
else:
   print('What?')
```

Real numbers. Accuracy

Let's see at the real number *X* as at the line segment : $[X - \mathcal{E};\ X + \mathcal{E}]$.

Example. Let *X* and *Y* be real numbers with accuracy 6 decimal places,
$\mathcal{E} = 5 * 10^{-7}$ and $|X| < 10^{9}$, $|Y| < 10^{9}$.

In worst case, when *X* and *Y* equal 10^{9} and maximum deviation:
$(X + \mathcal{E}) * (Y + \mathcal{E}) = XY + (X + Y) * \mathcal{E} + \mathcal{E}^{2}$, the value of $\mathcal{E}^{2}$ is negligible,
XY – right answer, $(X + Y) * \mathcal{E}$ – inaccuracy.

$(X + Y) * \mathcal{E} = 2 * 10^{9} * 5 * 10^{-7} = 10^{3}$
Absolute accuracy is 1000 ☹

```python
x = float(input())
y = float(input())
epsilon = 5 * 10**-7
if abs(x - y) < 2 * epsilon:
    print('Equal')
else:
    print('Not Equal')
```

Real numbers rounding

- **int** - discard the fractional part
- **round** - rounds to the nearest integer;

if there are several nearest integers, then to even

- **floor** – rounds down
- **ceil** – rounds up

function	2.5	3.5	-2.5
int	2	3	-2
round	2	4	-2
floor	2	3	-3
ceil	3	4	-2

```
import math
print (math.floor(-2.5))
print (math.ceil(-2.5))
```

```
from math import floor, ceil
print(floor(-2.5))
print(ceil(-2.5))
```

while loop

```
while    <test1>:            # Conditional expression test
        <statements1>        # loop body
        if <test2>: break    # Exit the loop by skipping the else part
        if <test3>: continue # Go to the beginning of the loop, to the
                               expression test1
else:
        <statements2>        # It is executed if the exit from the loop
                             # is performed by other statement (not break)
```

The ***while*** statement allows you to repeatedly execute the ***statements1*** command block while the ***test1*** condition is satisfied. It may also have an optional ***else*** clause.

Example. ***Print in increasing order all numbers from 1 to N :***

```
n = int(input ())
i = 1 # counter
while i <= n:
        print( i )
        i = i + 1
```

Example. ***Search for the minimum sequence of numbers ending by zero:***

```
now = int(input ())
nowMin = now
while now != 0:
        if now < nowMin:
              nowMin = now
        now = int(input ())
print(nowMin)
```

Example. **Guess the number.**

```
number = 23
running = True

while running:
        guess = int(input(Input integer: '))
        if guess == number:
            print(Congratulations, you are right!')
            running = False # stop the loop while
        elif guess < number:
              print(No, the number guessed is bigger than this. ')
        else:
              print(No, the number guessed is smaller than this.')
else:   print(‘the end of while')
# Here you can do anything else you need

print(The end.')
```

Break & continue

*The meaningful use of **break** is possible only if some condition is met, that is, **break** should be called only inside the **if** (inside the loop). Using **break** is a bad tone; it's best to do without it.*

```
i = 1
while True :
        print( i )
        i = i + 1
        if i > 100:
                break
```

The ***continue*** command starts the execution of the loop body again, starting with checking the condition. It should be used if, starting from some place in the body of the loop, under certain conditions, further execution is undesirable.

Example. **Print all the positive numbers in the sequence:**

```
now = −1 # the variable is initialized with a known appropriate value
while now != 0:
        now = int(input ())
        if now <= 0:
                continue ;
        print(now)
```

for loop

```
for <target> in <object>:   # Binds object elements with a loop variable
    <statements>            # loop body uses a loop variable
    if <test>: break        # Exiting the loop bypassing the else block
    if <test>: continue     # Go to the beginning of the loop
else:
    <statements>            # If you do not get on the instruction 'break'
```

When the interpreter executes a ***for*** loop, it alternately, **one by one assigns the elements of the sequence object to the loop variable** and executes the loop body for each of them. To access the current element in a sequence, the loop variable is usually used in the body of the loop, as if it were a cursor that moves from element to element.

After exiting the loop, the variable <target> usually **still refers to the last element of the sequence** if the loop was not finished by the break statement.

Loop often uses function **range** as an iterable object. It has three types of records: with one, two or three integer parameters.

1) **range(a)** generates **iterable** , which contains consecutive numbers from 0 to a-1
2) **range(a, b)** generates **iterable** with integers from a to b-1 inclusive
3) **range(a, b, c)** generates **iterable** with integers from a to b-1 inclusive with step **c**

```
# let’s calculate pythons
for i in range(3):
    print(i, ‘Pythons’)
else:
    print('No more Pythons')
```

```
# Factorial: F(n) = n!
n = int(input(Input a number: '))
f = 1
for i in range(1,n+1): # right border not included
   f=f*i
print(f)
```

Number systems.

1. What is a number system?
2. A bit of history.
3. Positional number systems. Conversion.
4. Mixed-based systems.

1. What is a number system?

Meaning I: **A collection of things** (usually called numbers) together with operations on those numbers and the properties that the operations satisfy.

Example: The counting numbers (1, 2, 3, …) together with the operations of addition, subtraction, multiplication, and division and the properties they satisfy.

Meaning II: **A system for representing numbers of a certain type.**
Example: The usual "base ten" or "decimal" system: 1, 2, 3, … , 10, 11, 12, … 99, 100, ….

- Roman numerals: I, II, III, IV, V, VI, VII, VIII, IX, X, …
- The binary system: 1, 10, 11, 100, 101, …(read as "one", "one, zero", "one, one", "one, zero, zero", etc.)

Meaning III: **A combination of Meanings I and II**. In other words, a collection of numbers together with operations, properties of the operations, and a system of representing these numbers.
Example: If we consider the counting numbers as a number system using **Meaning I, it doesn't matter** whether we expressed 4 as 4 (decimal), IV (Roman numeral), or 100 (binary), but using **Meaning III, it will matter.**

2. A bit of history

- Animals in a flock

  …

- Members in a tribe

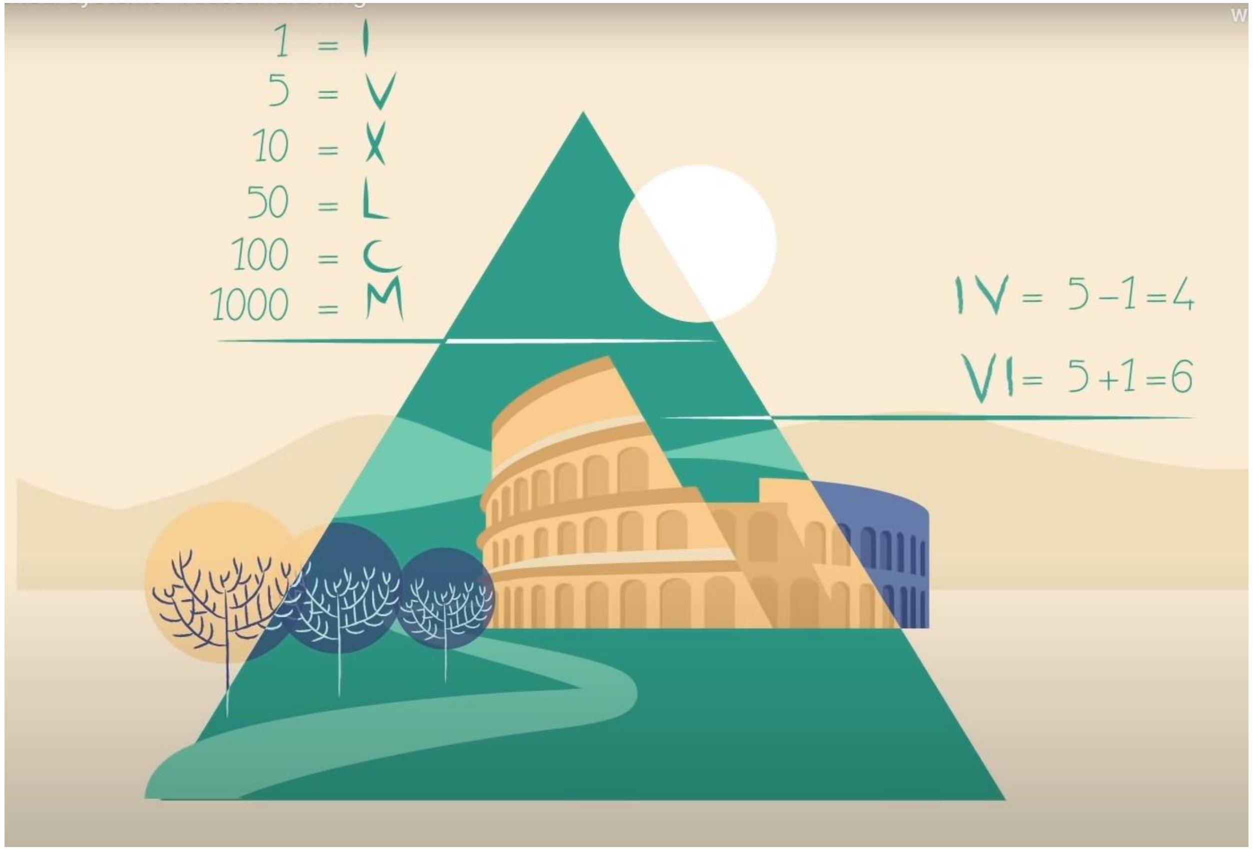
https://ed.ted.com/lessons/a-brief-history-of-numerical-systems-alessandra-king

As the complexity of life increased, along with the number of things to count, these methods were no longer sufficient.

Different civilizations came up with ways of recording higher numbers.

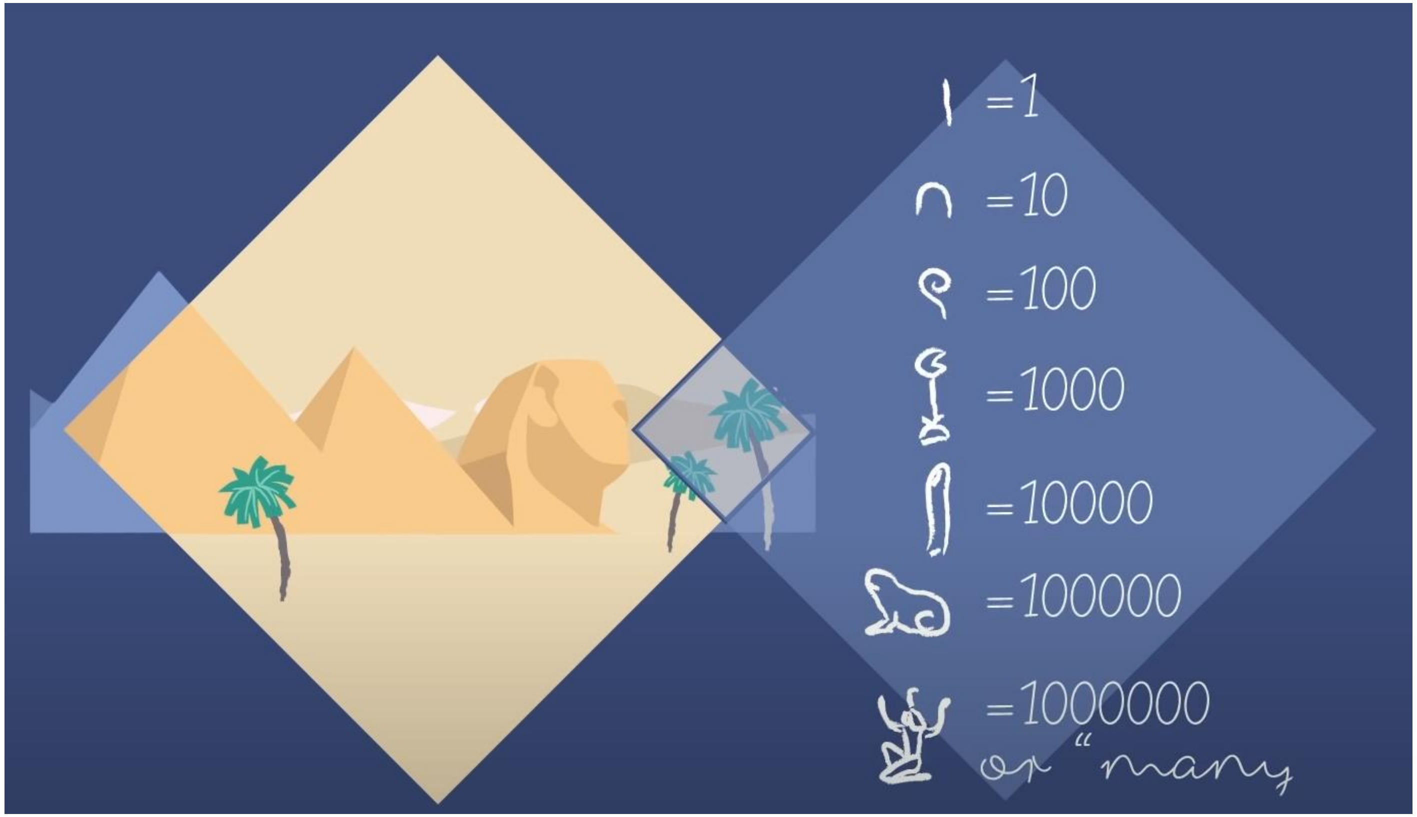
Many of them were **just extensions of tally marks** with new symbols added to represent larger magnitudes of value.

Roman number system



If a numeral appeared before one with a higher value, **it would be subtracted** rather than added.

Egyptian number system



It was needed to draw many symbols repeatedly and invent a new symbol for each larger magnitude.

A positional system could reuse the same symbols, assigning them different values based on their position in the sequence

A key breakthrough of this system, which is also independently developed by the Mayans, was the **number zero**.

3. Positional number systems

b – base of number system: the number of different digits used in this system

$\mathbf{(a_i a_{i-1} \ldots a_2 a_1 a_0 , a_{-1} \ldots)_b = a_i b^i + a_{i-1} b^{i-1} + \ldots. + a_2 b^2 + a_1 b^1 + a_0 b^0 + a_{-1} b^{-1} + \ldots}$

$13_{10} = 15_8 = 23_5 = 1101_2$

$253_7 = 2*7^2 + 5*7^1 + 3*7^0$ Convert to base 10: $2*49 + 5*7 + 3*1 = 136_{10}$

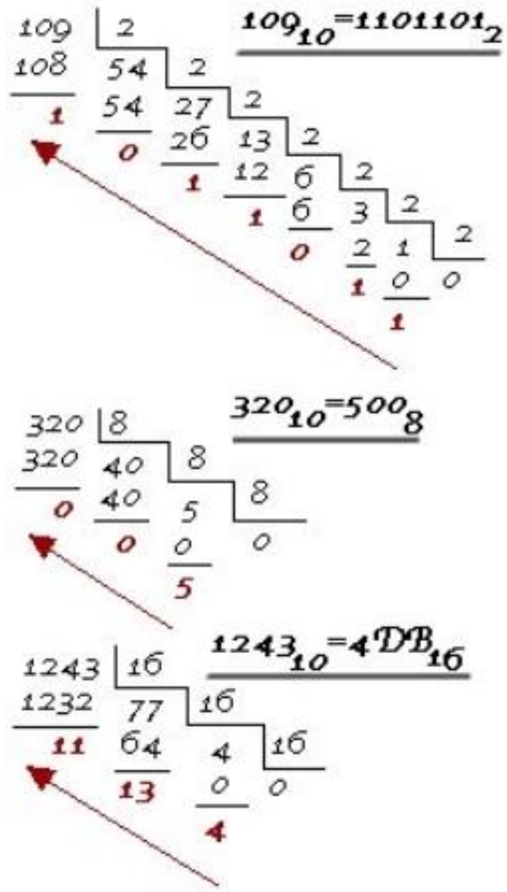
$1302{,}2_4 = 1*4^3 + 3*4^2 + 0*4^1 + 2*4^0 + 2*4^{-1} =$

$=1*64 + 3*16 + 0*4 + 2*1 + 2*0.25 = 64 + 48 + 2 + 0.5 = 114{,}5_{10}$

Conversion from decimal

In general, to converse the integer part of a number from the decimal to a system with some other base:

- Perform **sequential division with the remainder** of the original number and **each resulting quotient** by new base
- Write the calculated residuals starting from the last one (i.e. in reverse order)



Conversion from decimal

To converse the fractional part of a number into other number systems, you need to turn the integer part to zero and start **multiplying** the resulting **number by the base of the new system**.

If, as a result of multiplication, whole parts appear again, they must be turned to zero, having previously memorized their values. **The operation ends when the fractional part becomes zero or the required accuracy is achieved**.

$10{,}625_{10}$ → base 2

0,625*2 = 1,25
0,250*2 = 0,5
0,5*2 = 1,0

$10{,}625_{10}$ = (1010), (101) = $1010{,}101_2$

Conversion from decimal

$$(75)_{10} = (1001011)_2$$

…	0	0	0	1	0	0	1	0	1	1
	2^9	2^8	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
	||	||	||	||	||	||	||	||	||	||
	512	256	128	64	32	16	8	4	2	1

$$1 + 2 + 4 + 8 + 16 + 32 = 63$$

$$1 + 2 + 4 + 8 + \ldots + 2^k = 2^{k+1} - 1$$

Binary ←→ hexadecimal

Hex Digit	4 Bit Binary	Hex Digit	4 Bit Binary
0	0000	8	1000
1	0001	9	1001
2	0010	a	1010
3	0011	b	1011
4	0100	c	1100
5	0101	d	1101
6	0110	e	1110
7	0111	f	1111

$$(3b9)_{16} = (\underbrace{0011}_{3}\underbrace{1011}_{b}\underbrace{1001}_{9})_2$$

$$(\underbrace{0110}_{6}\underbrace{1101}_{d}\underbrace{0011}_{3})_2 = (6d3)_{16}$$

Binary ←→ hexadecimal

$(011011010011)_2$

$$= \underbrace{1\cdot2^0 + 1\cdot2^1 + 0\cdot2^2 + 0\cdot2^3} + \underbrace{1\cdot2^4 + 0\cdot2^5 + 1\cdot2^6 + 1\cdot2^7} + \underbrace{0\cdot2^8 + 1\cdot2^9 + 1\cdot2^{10} + 0\cdot2^{11}}$$

$$= \underbrace{(1\cdot2^0 + 1\cdot2^1 + 0\cdot2^2 + 0\cdot2^3)}\cdot2^0 + \underbrace{(1\cdot2^0 + 0\cdot2^1 + 1\cdot2^2 + 1\cdot2^3)}\cdot2^4 + \underbrace{(0\cdot2^0 + 1\cdot2^1 + 1\cdot2^2 + 0\cdot2^3)}\cdot2^8$$

$$= \quad \underbrace{(3)}\cdot2^0 \qquad \underbrace{(13)}\cdot2^4 \qquad \underbrace{(6)}\cdot2^8$$

$$= (6d3)_{16}$$

Mixed-based systems

In each digit (position) of the number, the set of valid values may differ from the sets of other digits.

The time measurement system:

- seconds and minutes - from "00" to "59"
- hours - from "00" to "23"
- days – 365

The digit a_i in any number $a_n a_{n-1} \ldots a_0$ lies in the range 0 to R_i, where R_i is not the same for every i. The number is then interpreted as

$(\ldots(a_n R_{n-1}) + a_{n-1})R_{n-2} + \ldots + \ldots + a_1)R_0 + a_0$

Example:

122 days 17 hours 35 minutes 22 seconds =

$((((1\times10) + 2)10 + 2)24 + 17)60 + 35)60 + 22$ seconds

Python data types

Python has 4 types categories:

- **Numbers** (integer, real, complex). Supports addition, multiplication, etc.
- **Sequencies** (string, list, tuple). Support indexing, slice extraction, concatenation, etc..
- **Mappings** (Dictionaries). Support key indexing operation, etc.
- **Sets.** They form a separate category of types (they do not map keys to values and are not ordered sequences).

Built-in types

- **Built-in objects simplify the creation of programs.** Since they eliminate the need to implement own data structures.
- **Built-in objects give the possibility to quick extension.** To solve complex problems, the user can create his own objects using the built-in classes of the Python language.
- **Built-in objects are often more efficient than user-created data structures.** They use the already optimized data structures implemented in the C language to achieve high performance.
- **Built-in objects are a standard part of the language.** Python takes a lot from languages that rely on the use of built-in tools (such as LISP) as well as from languages that rely on the skill of a programmer who must implement their own data structures (such as C ++). In Python, you can create your own types of objects, but this is not recommended at the very beginning. Moreover, built-in components are standard components of the Python language, they always remain unchanged, while user's structures can be unexpectedly changed.

Object generation

Object type	Example literals/creation
Numbers	`1234, 3.1415, 3+4j, 0b111, Decimal(), Fraction()`
Strings	`'spam', "Bob's", b'a\x01c', u'sp\xc4m'`
Lists	`[1, [2, 'three'], 4.5], list(range(10))`
Dictionaries	`{'food': 'spam', 'taste': 'yum'}, dict(hours=10)`
Tuples	`(1, 'spam', 4, 'U'), tuple('spam'), namedtuple`
Files	`open('eggs.txt'), open(r'C:\ham.bin', 'wb')`
Sets	`set('abc'), {'a', 'b', 'c'}`

A literal is an expression that generates an object.

Python does not have a type declaration construct**; the syntax of executed expressions itself defines the types of objects** to be created and used.

As soon as an **object** is **created**, it will be **associated with its own set of operations** throughout its existence - only string operations for strings, only list operations for lists, etc.

Strings

Strings are used to write text information, as well as sequences of bytes.

As a sequence a string **support the order of elements**, from left to right: elements are stored and retrieved based on their positions in the sequences.

Strictly speaking, **strings are sequences of characters**.

>>> S = 'Spam' #A variable is created when it is assigned a value.
>>> len(S) # length
4
>>> S[0] # The first element
'S'
>>> S[1] # The second element
'p'

Reverse Indexing:

S[-1] # The last element
S[-2] # The second element from the end
S[len(S)-1] # The last element

Strings. Literals.

- ‘spa”m’
- “spa’m”
- ‘‘‘... spam ...’’’, “““... spam ...”””
- Escaped sequences allow you to insert characters that are difficult to enter from the keyboard: “s\tp\na\0m”:

s p
a m

- R ”C:\new\test.spm” – raw strings
- Byte string: b’sp\x01am’

Escaped sequences

Sequence	Description
\\	The backslash character itself (one character \ remains)
\'	Apostrophe (one character remains ‘)
\"	Quotation mark (one character remains “)
\b	Backspace
\n	Newline (linefeed)
\r	Carriage return
\t	Horizontal tab
\v	Vertical tab
\xhh	Character with hex value *hh* (exactly 2 digits)
\ooo	Character with octal value *ooo* (up to 3 digits)
\0	Null: binary 0 character (doesn’t end string)
\N{id}	Unicode database ID
\uhhhh	Unicode character with 16-bit hex value
\Uhhhhhhhh	Unicode character with 32-bit hex valuea

String Formatting

Often there are situations when you need to **make a string**, substituting some **data obtained during the execution** of the program.

1. % Operator

A format string containing one or more **format specifiers,** whose value should be substituted for the specifiers on the left side of the expression.
for example, % d (digit)) **%** <an object (or objects, in the form of a tuple)>.

```
>>> 'That is %d %s fish!' % (1, 'gold')
That is 1 gold fish!
```

```
>>> "%d %s %d you" % (1, 'spam', 4)
1 spam 4 you
```

```
>>> "%s -- %s -- %s" % (42, 3.14159, [1, 2, 3])
42 -- 3.14159 -- [1, 2, 3]
```

String Formatting

Once you start using several parameters and longer strings, your code will quickly become much less easily readable

```
first_name = "Eric"
last_name = "Idle"
age = 74
profession = "comedian"
affiliation = "Monty Python"
print("Hello, %s %s. You are %s. You are a %s. You were a member of %s." %
(first_name, last_name, age, profession, affiliation))
```

Hello, Eric Idle. You are 74. You are a comedian. You were a member of Monty Python.

String Formatting

2. **The format method**

str.format() is an improvement on %-formatting, it **uses normal function call syntax**.

<Replacement Fields> .**format** <parameters>

The replacement fields are marked by curly braces.

```
>>> '{}, {}, {}'.format('a', 'b', 'c')
'a, b, c’

>>> '{0}, {1}, {2}'.format('a', 'b', 'c')
'a, b, c’

>>> '{2}, {1}, {0}'.format('a', 'b', 'c')
'c, b, a’
```

```
s1='a'; s2='b'; s3='c'
print('{2}, {1}, {0}'.format(s1, s2, s3))
```

c, b, a

```
person = {'name': 'Eric', 'age': 74}
print("Hello, {name}. You are{age}.".format(name=person['name'],age=person['age’]))
```

Hello, Eric. You are 74.

2. **The format method**

Code using ***str.format()*** is much more easily readable than code using ***%-formatting***, but str.format() can still be quite verbose when you are dealing with multiple parameters and longer strings.

```
first_name = "Eric"
last_name = "Idle"
age = 74
profession = "comedian"
affiliation = "Monty Python"
print(("Hello, {first_name} {last_name}. You are {age}. " +
"You are a {profession}. You were a member of
{affiliation}.") \
.format(first_name=first_name, last_name=last_name,
age=age, \
profession=profession, affiliation=affiliation))
```

Hello, Eric Idle. You are 74. You are a comedian. You were a member of Monty Python.

String Formatting

3. **f-Strings**

Also called “formatted string literals”, ***f-strings*** are string literals that have an ***f*** at the beginning and curly braces containing expressions that will be replaced with their values.

```
name = "Eric"
age = 74
print(f"Hello, {name}. You are {age}.")
Hello, Eric. You are 74.

print(F"Hello, {name}. You are {age}.")
Hello, Eric. You are 74.
```

String Formatting

3. **f-Strings**

Because **f-strings are evaluated at runtime**, you can put any valid Python expressions in them.

```
>>> f"{2 *37}"
'74'
```

You could also call functions:

```
def to_lowercase(input):
   return input.lower()
name = "Eric Idle"
print(f"{to_lowercase(name)} is funny.")

eric idle is funny.
```

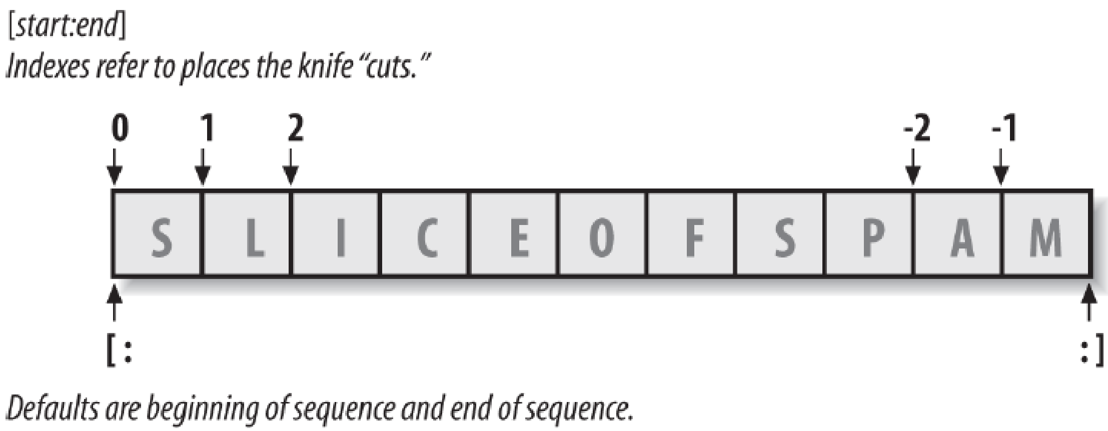
You also have the option of calling a method directly:

```
>>> f"{name.lower()} is funny."
'eric idle is funny.'
```

String operations

base	description
len(‘abc’) = 3	Length: number of elements
‘abc’ + ‘def’ = 'abcdef'	Concatenation
‘Hi!’ * 4 = 'Hi!Hi!Hi!Hi!'	Replication
s[i]	Indexing by position
s[i:j]	Slicing
S.find(‘pa’); S.rfind(‘pa’)	Search for a substring in a string (returns index of 1st or last substring, or -1)
‘spam’ in S	Is ‘spam’ in S?
S.count('pa', i, j)	Count the number of occurrences of a substring in a slice [i :j] of the string S (till j-1)
S.lstrip() и S.rstrip()	Removing whitespace from the beginning (lstrip) and the end (rstrip)
S.replace(‘pa’, ‘xx’)	Replacing one substring with another (all occurrences)
S.split(‘,’)	Separation by a character - separator
S.isdigit()	Checking the string’s content
S.lower(); S.upper()	Character case conversion
S.endswith(‘spam’)	String ending check
for c in mystr: print(c, end=’ ‘)	Loop through string items
s.rjust(n, ‘c’)	Increasing string s to length n by filling left with ’c’

Slices



Positive offsets are counted from the left end (the first element has an offset of 0), and **negative ones are counted from the right end** (the last element has an offset of –1).

S[i:j] - The offset to the left of the colon indicates the left border (inclusive), and to the right the upper border (it is not included in the slice).

If the left and right borders are omitted (:), the default values are 0 and the length of the object, respectively.

Slices

The full form of the substring extraction operation looks like this: **S[i:j:k]**.
It means: **«Extract all elements of the sequence S, starting from offset i, up to offset j-1, in increments of k.»**
By default, k = 1. For k < 0, counting is performed in the opposite direction (from right to left).

```
>>> S='Spam’
>>> S[1:3] # Slice of the string S starting at offset 1 and up to 2
pa
>>> S[1:] # All exept the first element (1:len(S))
pam
>>> S[:3]  # All exept the last element ( S[0:3])
Spa
>>> S[:-1] # All exept the last element
Spa
>>> S[ : : -1]
?
```

String elements numbering

S	t	r	i	n	g
0	1	2	3	4	5
-6	-5	-4	-3	-2	-1

Immutable sequence

All operations on strings as a result create a new string, because strings in Python are immutable - once a string is created, it cannot be changed.

It is impossible to change a string by assigning a value to one of its positions, however, you can create a new string and give it the same name:

>>> S
Spam
>>> S[0] = ‘z’ # Immutable objects cannot be changed
... error message text omitted ...
>>> S = ‘z’ + S[1:] # But using expressions you can create new objects
>>> S
zpam

Numbers, strings and tuples are immutable, but lists and dictionaries are not (they can easily be changed in any part).

Example 1.

```
# Delete all symbols, which indexes are divided by 3.

s = input('Input a string: ')
for i in range(len(s)):
    if i % 3 != 0:
        print(s[i], end='')
print()
print('The end')
```

Examples 2 & 3 & 4.

```
# Double all characters except the 'o'
character.

s = input()
for i in s:
    if i == 'o':
        continue
    print ( i * 2, end=‘’ )
```

```
# Double all characters, to the character 'o'.
Other characters do not display.

s = input()
for i in s:
    if i == 'o':
        break
    print ( i * 2, end=‘’ )
```

```
# Check if there is an 'o' in the string.
s = input()
for i in s:
    if i == 'o’:
        print ( “ There is ‘o’ in the string“ )
        break
else:
    print (“There is not ‘o’ in the string“ )
```

Example 5.

```
# Display the message, and between each character it will
display a '*'.

msg = input('Write a message: ')
for letter in msg:
        print(letter, end='*')
```

Hello
*H*e*l*l*o**

Example 6.

```
# Ask the user to type in their name and then tell them how many vowels are in their name

name = input('Enter your name: ')
count = 0
name = name.lower()
for x in name:
    if x == 'a' or x == 'e' or x == 'i' or x == 'o' or x == 'u':
        count += 1
print("Vowels = ", count)
```

Yandex Contest

Message	Short	Meaning	Possible reason
OK	OK	The solution accepted	The program works correctly on the built set of tests
Compilation error	CE	Compilation error	1. The program made a syntax or semantic error 2. The language is incorrect
Wrong answer	WA	Wrong answer	1.error in the program 2.incorrect algorithm
Presentation error	PE	The testing system cannot verify the output, as its format does not match the one described in the task conditions	1.Incorrect output format 2.The program does not print the result 3.Extra output
Time-limit exceeded	TL	Программа превысила установленный лимит времени	1. ошибка в программе 2. неэффективное решение
Memory limit exceeded	ML	The program has exceeded the memory limit	1.an error in the program (for example, infinite recursion) 2.inefficient solution
Run-time error	RT	The program terminated with a non-zero return code	1. runtime error 2. a program in C or C ++ does not end with the operator return 0 3. a nonzero return code is specified explicitly