BaseFunctionsPy_EN

Introduction

In this guide you can get basic knowledge of Python. The reason for writing the guide was the complexity and confusion that I have noticed in other guides and guides. Of course, I am not an expert in Python, but at the same time I have some experience and I think that I can help others with this knowledge :)

Useful links

Since this guide is quite subjective, I'm not sure that I will cover all the topics and I'm not sure that we won't have questions after reading, so in this block I will attach good links that can be used in such cases.

First of all, you can email me and I will try to help you:

boyarkin.gleb@gmail.com

source.boar@gmail.com

The mail is also duplicated in my profile.

Besides me, you can use:

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

These sources I guarantee that they are verified, I myself studied from them, but as for me, many things are painted there with too complicated words.

Briefly about Python

Python is quite an old but popular language nowadays. It is very easy to write small programs on it, which is why it is mainly used in small projects or for small automations.

Syntax

The syntax in Python is not C-like and differs quite a bit, but it doesn't make it any more difficult.

** If you go through the basic rules of syntax then: **

- there is no line ending character
- `:` - go one level down, open code block
- `snake_case` - naming style for variables and functions
- `PascalCase` - class naming style
- `#` - character for single line comment
- `''' '''` - characters for commenting a block of code

For more information about coding rules, see the official `PEP8` ruling: https://peps.python.org/pep-0008/

Working with the console

`input('''message text''')` – console input

`print('''message text''')` - output to the console

Variables

As I think you know, or will become known now, in fact the entire programming language is based on variables and operations on them.

In Python, a variable is a pointer to an area or location in memory that holds a value. For now, the fact about memory can simply be put into your head, we will come to it later.

The ad does not require keywords.

```
a1 = 1
a2=True
a3="hello"
```

When naming variables, as mentioned above, the snake_case style is used.

What does it include?

snake_case:

- all lowercase letters
- words are separated by `_`

Also, if specifically about Python, then in it the name of a variable can begin with the symbol `_` or a letter of any case, subsequent characters can already be the same plus numbers (0-9). It is also worth clarifying that Python is case sensitive, that is, in it `a` and `A` are different names.

It is not explicitly specified whether a variable is local or global in Python. By default, they are local to the block of code in which they are declared. To use globals or variables of an external block of code, use `global` and `nonlocal`

###global

This keyword is used to indicate that the variable in this block of code will be taken from the global level, the program level.

Example:

```
a1 = 1

def func1():
     print(a1) # a1 is not declared

def func2():
     global a1
     print(a1) #1

def func3():
```

```
    fucn4():
        global a1
        print(a1) #1
```

###nonlocal

This keyword is used to indicate that the variable in this code block will be taken from the code block one level up.

Example:

```
a1 = 1

def func1():
    a1 = 2
    fucn2():
        global a1
        print(a1) #1

    def func3():
        nonlocal a1
        print(a1) #2
```

Data types

Data type - types of what type of value the variable can store.

Basic data types in Python:

- `bool` – boolean data type, stores 2 types of values: `True` or `False`
- `int` - integer data type, stores integers, no limit on value, it depends on the power of the computer
- `float` – fractional data type, stores fractional numbers
- `complex` – complex data type, stores complex numbers
- `str` - string data type, stores a string of text of almost unlimited size

Typing

Python typing is dynamic, strong, implicit.

Now let's try to understand:

- Dynamic - means that the variable is not bound to a single type when declared, the variable can store different types as the program is used
- Strong - means that when doing actions with a variable of one type, actions of another type will cause an error. *For example:* `1+"2"` will throw an error
- Implicit - means that when declaring it is not necessary to "explicitly" indicate the type of the variable. *For example, like this in C++ with explicit typing:* `string a = "a"` , when `a = "a"` is enough in Python, without specifying the type

What gives us this knowledge?

From Python's typing, we can extract the facts that:

- We don't need to keep track of types at the program level
- We do not need to declare a variable by specifying its type
- We can assign values of different types to one variable

Consider specifically the different types

bool

This is a boolean data type that is used in places where you need to store 2 options for a variable: YES (`True`) or NO (`False`)

Declarations of variables of this type may look like this:

```
t=True
f=False
```

They are used mainly in conditional constructions, which we will consider later.

int

This is an integer data type that stores integers.

Declarations of variables of this type may look like this:

```
p=1
n=-2
o = 0
```

To convert other types to `int` you can use the `int()` function

float

The same as `int` only stores fractional numbers

To convert other types to `float` you can use the `float()` function

str

A string data type that stores text.

Declarations of variables of this type may look like this:

```
txt1 = "Hello"
txt2 = 'World'
```

There is no difference between the quotes, the main thing is that they open and close the string at the same time.

To convert other types to `str` you can use the `str()` function

There are also a very large number of methods for strings, a list of which and instructions for them can be found at the link: https://www.w3schools.com/python/python_strings_methods.asp

Operators

Assignment operators

`=` - assignment

`+=` - addition assignment

`-=` - assignment with subtraction

`*=` - multiplication assignment

`/=` - assignment with division

`//=` - integer division assignment

`%=` - division assignment with remainder

`**=` - assignment with exponentiation

`<<=` - left shift assignment

`>>=` - right shift assignment

`&=` - assignment with bitwise `AND`

`|=` - assignment with bitwise `OR`

`^=` - assignment with bitwise exclusive `OR`

Boolean operators

`==` - equality

`!=` - inequality

`>` - more

`>=` - greater than or equal to

`<` - less than

`<=` - less than or equal to

`and` is a logical `AND`, i.e. both conditions in the construction `(condition 1)&&(condition 2)` must be `True`

`or` is a logical `OR`, i.e. at least one condition in the construction `(condition 1)||(condition 2)` must be `True`

Arithmetic operators

`+` - plus

`-` - minus

`*` - multiplication

`/` - division

`//` - integer division

`%` - division with remainder

`**` - exponentiation

Bitwise operators

`&` - bitwise `AND`

`|` - bitwise `OR`

`^` - bitwise exclusive `OR`

`~` - bitwise `NOT`

`<<` - shift left

`>>` - right shift

Small important note

`pass` is a keyword which means that the block is empty. It should be used if you need to leave a block of code empty, because without this word there will be an error.

Conditional constructs

Variables are, of course, good, but you have to work with them somehow, and conditional constructions are one of the ways to write program logic.

if...else

The most famous and commonly used design.

A condition is passed to `if`, if the condition is true, then what is in the `if` block is executed, otherwise what is in the `else` block

Syntax looks something like this:

```
if '''condition: either a logical expression or a variable of type bool''':
    # action if condition == True
else:
    # action if condition == False
```

You can also add `elif` to this construction

Application example:

```
if '''condition 1: either boolean expression or bool''':
    # action if condition 1 == True
elif '''condition 2: either boolean expression or bool''':
    # action if condition 1 == False, but condition 2 == True
...
else:
    # action if condition 1 == False and condition 2 == False
```

Such `elif` constructions can be added any number of times

try...except...finally

Conditional block, in which the execution condition will be the presence or absence of an error in the `try` code block

Syntax looks something like this:

```
try:
    # block of code that will reproduce the search for an error
except:
    # action if error
finally:
    # action after the execution of the block, which will not reproduce on the error search
```

Loops

while

A loop that executes an action while the condition is true

Syntax looks something like this:

```
while '''condition: either a logical expression or a variable of type bool''':
    # action if condition is True
```

for...in

Type of loop with an action that is performed before the start of the bypass, an execution condition and an action that is performed at the end of each pass

Syntax looks something like this:

```
for '''variable''' in '''set of values''':
    # action while the variable is in the value set
```

range

`range()` is a function to create a value object

Syntax looks something like this:

```
range('''from''', '''to''', '''step, default 1''')
```

`break` and `continue` keywords

`break` is used to exit the loop early, i.e. to terminate the loop regardless of the condition

Usage example:

```
for i in range(0, 10):
    if i % 3 == 0: # if the number is evenly divisible by 3
        break
    print(i)
```

That is, the loop was executed until it came across the first number that is divisible by 3

`continue` is used to terminate the loop pass early and move to the next pass, that is, it is used when you need everything after its call not to be executed in this pass, but to be executed immediately making the pass

```
for i in range(0, 10):
    if i % 3 == 0: # if the number is evenly divisible by 3
        continue
    print(i)
```

That is, for numbers that are divisible by 3, `print()` was not executed, but the next loop run was immediately performed

Practical tasks on the given knowledge

For better consolidation, I propose to perform a few simple tasks that will check how much you understand this level of material, all the knowledge that may be needed to complete the tasks is above.

All problem solutions will be in the "Problems" folder and will be numbered, but I do not recommend you use them until you yourself succeed

1. Add such variable declarations or keywords at the beginning of functions to the beginning of the code so that the rest of the code works as written in the comment. Not all places to write code must be filled

```
# write your ads here
print(a) #1
def func1():
    # or here
    print(a) #1

def func2():
    a = 3
    def func3():
        # or here
        print(a) # 3
    func3()
```

```
func1()
func2()
```

(All console output must occur)

2. Write a loop that will print to the console all numbers from 1 to 100 inclusive that are divisible by 5 or 7 (`n % /*number*/ == 0` - the condition for divisibility of `n` by any number)
3. Write a block of code that, if the action `a = 1 / 0` is performed, the phrase "You cannot divide by zero" will be displayed, and after that the phrase "Errors processed" will be displayed

Functions

Functions are another important programming tool. If in simple words, this is the union of any section of code for its further call.

Every function in any programming language has:

- Name
- List of arguments (parameters) – variables whose values are set when calling the function. They are declared in `()` separated by commas. Also, when calling, their values are specified when calling in the same way in `()`. In the code block of our function, they play the role of ordinary block-scoped variables, but they do not need to be declared, since they are declared when the function is called.
- A block of code that will be executed when called. Written after `:`

Syntax looks something like this:

```
def '''function name'''('''argument1''', '''argument2'''...'''argumentN'''):
    # block of executable code
```

`def` is a keyword that means function declaration

To call a function, it is enough to write its name and `()` in which you can list the arguments if they were declared

Function example:

```
def f(a1, a2, a3):
    print(a1)
    print(a2)
    print(a3)

f(1, 2, 3)
```

Functions and procedures

If we deviate from the topic of Python, where in fact there is no syntactic difference between these concepts, then this is a very important question.

What's the difference?

Function is a block of code that will return some value as a result of its execution.

Procedure is a function that does not return a value

The `return` keyword is responsible for returning a value in Python.

Let's see the difference with examples:

```
def func(a1, a2, a3, a4): # function
    print(a1)
    print(a2)
    print(a3)
    return a4

def proc(b1, b2, b3, b4): # procedure
    print(b1)
    print(b2)
    print(b3)

let a = func(1, 2, 3, 4) # == 4(a4)
let b = proc(1, 2, 3, 4) # == None
```

What happened?

When declaring a **function**, we used `return`, so the function call can be assigned to a variable and it will have the value that we specified in `return`

And when declaring **procedure** `return` we didn't use it, which is why our call doesn't matter, i.e. `None`

lambda functions and assigning a function to a variable

In Python, it is possible to assign a function to a variable, or, to be more precise, a reference to a function. Thus it can be "renamed"

*** So it might look like this: ***

```
def func(a1, a2, a3):
    print(a1)
    print(a2)
    print(a3)

proc = func
```

This method is not not particularly correct from a logical point of view, but at the same time it is possible.

If you want to shorten the code and not break the logic, you can use **lamda functions**

Syntax looks something like this:

```
'''function name''' = lambda '''argument1''', '''argument2'''...'''argumentN''': #
executable code block
```

There are several important aspects:

- Everything is written in one line, transfer is impossible
- The block of executable code is automatically returned

Example:

```
sq = lambda x: x ** 2 # squaring function
```

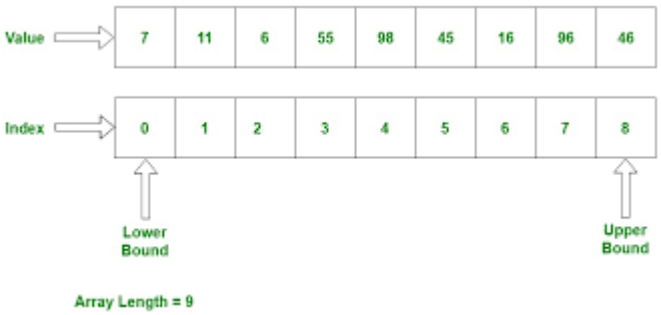
Ordering data types

Lists

In Python, arrays are represented by a **list**. In fact, in other languages: **array** is a data type with a fixed size, and **list** is a resizable data type. But some languages ignore this fact, but not Python.

For declaration, you can simply enumerate the elements in `[]` and assign to a variable

```
arr = [1, 2, 3]
```



For a better understanding, let's use the picture above. As you can see, an array (list) is, roughly speaking, a set of "cells" with values to which indexes are attached in parallel, by which you can get the value to which they correspond.

!!!Indexing in programming languages starts from 0!!!

To display by index, the name of the list in `[]` is enough to indicate the index. To reverse from the end, you can use negative indices (-1 - last, -2 - penultimate ...)

```
arr = [1, 2, 3]
a = arr[0] # 1
arr[-1] = 4 # arr == [1, 2, 4]
```

Accessing a non-existent index in Python will throw an error.

```
arr = [1, 2, 3]
a = arr[4] # Error
```

Basic array methods and functions

- `len()` - returns the length of the list

```
arr = [1, 2, 3]
l = len(arr) # 3
```

- `.append(item)` - adds an `item` element to the end of the list

```
arr = [1, 2, 3]
arr.append(4) # arr == [1, 2, 3, 4]
```

- `.insert(index, item)` - adds the element `item` to the list at index `index`

```
arr = [1, 2, 3]
arr.insert(1, 4) # arr == [1, 4, 2, 3]
```

- `.extend(items)` - adds a set of `items` elements to the end of the list

```
arr = [1, 2, 3]
arr.append([4, 5, 6]) # arr == [1, 2, 3, 4, 5, 6]
```

- `.clear()` - remove all elements from the list

```
arr = [1, 2, 3]
arr.clear() # arr == []
```

- `.index(item)` - returns the index of the `item` element. If the element is not found, throws a `ValueError` exception

```
arr = [1, 2, 3]
i = arr.index(2) #1
j = arr.index(4) # ValueError
```

- `.find(item)` - returns the index of the `item` element. If the element is not found return -1

```
arr = [1, 2, 3]
i = arr.find(2) # 1
j = arr.find(4) # -1
```

- `.pop(index)` - removes and returns the element at index `index`. If no index is passed, then simply removes the last element.

```
arr = [1, 2, 3]
i = arr.pop(1) # arr == [1, 3]
```

- `.count(item)` - returns the number of occurrences of the element `item` in the list

```
arr = [1, 2, 2, 3]
c = arr.count(2) # 2
```

- `.reverse()` - reverses all elements in a list

```
arr = [1, 2, 3]
i = arr.reverse() # arr == [3, 2, 1]
```

- `'''separator'''.join()` - joins the array into a string and inserts between the elements the separator that we specify in the argument

```
arr = [1, 2, 3]
s = "-".join(arr) # "1-2-3"
```

- `min()` - returns the smallest element of the list

```
arr = [1, 2, 3]
a = min(arr) # 1
```

- `max()` - returns the largest element in the list

```
arr = [1, 2, 3]
a = max(arr) # 3
```

For the rest you can find out at the link: https://www.w3schools.com/python/python_lists_methods.asp

Slice

To string a list in Python, use the following construct:

```
arr['''c''':'''to''']
```

If one of the parameters is not specified, it automatically becomes the edge of the list on its side.

Also, the `[::-1]` construct will invert the list.

Array iteration

for...in...range

```
for i in range(0, len(arr)):
    # array element arr[i]
```

for...in

```
for a in arr:
    # array element a
```

Dictionaries

Sometimes there are situations when it is necessary that the indices were not numbers, but something of their own. For this, **dictionaries** were created.

Dictionaries is an array with self-specified indices.

Declaration syntax

```
dic={
    '''index name 1''': '''value 1''',
    '''index name 2''': '''value 2''',
    ...
    '''index name N''': '''value N'''}
```

Reversal and assignment is the same, only indices are not necessarily digits:

```
dic = {"name": "Sam", "age": 17}
d = dic["name"] # "Sam"
dic["age"] = 18
```

Tuples

Tuple - a sequence of elements, similar to a list, except that it cannot be changed.

Declared as an array, except not in `[]` but in `()`. But when accessing elements, so is `[]`.

Usage example:

```
tup = ("Tom", 17)
t = tup[0] # "Tom"
tup[1] = 18 # not possible
```

Working with strings

After we got acquainted with arrays, we can talk about strings.

In fact: **string** is an array of characters. Its elements can be accessed as in an array, but cannot be changed as elements in an array.

```
s="abc"
c = s[0] # "a"
s[0] = "d" # not possible
```

Also in Python, strings can be added and multiplied.

```
s1 = "hello"
s2 = "world"
sp = s1 + " " + s2 # "hello world"
sm = s1 * 3 # "hellohellohello"
```

String methods and functions

- `len()` - returns the length of the string

```
s="abc"
l = len(s) # 3
```

- `.lower()` - returns a lowercase string

```
s1 = "abC"
s2 = s1.lower() # "abc"
```

- `.upper()` - returns an uppercase string

```
s1 = "abC"
s2 = s1.upper() # "ABC"
```

- `.split('''delimiter''')` – splits a string into an array of strings by delimiter

```
s = "a-b-c"
arr = s.split("-") # ["a", "b", "c"]
```

- `ord()` - get element code

```
c = ord("a") # 97
```

- `chr()` - getting an element by code

```
c = chr(97) # "a"
```

Practical tasks on the given knowledge

4. Figure out how to reverse a string
5. Find a way to turn the string `"abcdef"` into `"cdefgh"`
6. Write a function that will return a string created according to the `snake_case` rule (the separator must be passed as the second argument)

   Example:

   `hello world` -> `hello_world`
7. Create a function to populate an array of dictionaries that contain information about a person

   Structure:

```
people = [] # main array
'''dictionary structure
{
```

```
    "firstName": "",
    "secondName": "",
    "age": "",
    "country": "",
    "city": ""
}
'''
```

8. - Think about how you can sort the array

"*" - difficult task

Everything!

It's all about the prince! I gave you everything I wanted. Hope this guide helped you.

I also have a more difficult guide with more complex topics, but it's more just a collection of solutions, but for those who are interested, you can read: https://github.com/s0urcedev/AdditionalFunctions

Thanks to all! Hope to see you again :)