Lesson Overview

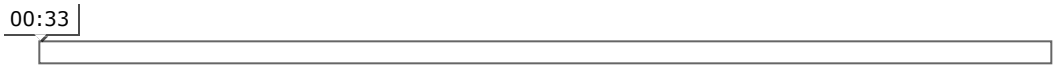
Bloody difficult stuff, these functions, huh? But we know, you have studied like a hell. Good for you.

We can focus on better typing of functions. For example, we can create **function with various amount of arguments**.

Don´t forget the homeworks. Functions are very important concept and you need to understand it properly.

To give you a more detailed overview, we'll go through:

- **function scopes**, where we'll talk about local, enclosing, global, built-in scopes and more,
- **function inputs**, where we'll talk about arguments vs paramters, keyword vs position argumetns, etc.



Sum

Create a function that will calculate a sum of all numbers in a given sequence.

Example of function in use:

```
>>> sequence = [32,43,54,54,76,21,62,83,52,58]
>>> my_sum(sequence)
535
```

Code Solution

Use dropdown feature below if you want to see, how we wrote the code.

Click to see our solution

Count

PYTHON ACADEMY / 7. FUNCTION SCOPES & INPUTS / REVIEW EXERCISES / COUNT

Create a function that will count the number of ocurrences of a given item in a given sequence.

Example of the function in use:

```
>>> data = [35, 14, 26, 48, 49, 26, 18, 25, 16, 16, 39, 17, 10, 29, 30]
>>> count(16, data)
2
>>> count(72, data)
0
```

Code Solution

Use dropdown feature below if you want to see, how we wrote the code.

Click to see our solution

Mean

Create a function that will calculate the average value for a given sequence of numeric values.

Example of function in use:

```
>>> sequence = [32,43,54,54,76,21,62,83,52,58]
>>> mean(sequence)
53.5
```

Code Solution

Use dropdown feature below if you want to see, how we wrote the code.

Click to see our solution

Modus

The modus(mode) of a set of data values is the value that appears most often. Your task is to create a function, that will determine mode for a given sequence of values:

Example of function in use:

```
>>> data = [35, 14, 26, 48, 49, 26, 18, 25, 16, 16, 39, 17, 10, 29, 30]
>>> modus(data)
26
```

Code Solution

Use dropdown feature below if you want to see, how we wrote the code.

Click to see our solution

Median

For a data set, median may be thought of as the "middle" value. For example, in the data set {1, 3, 3, 6, 7, 8, 9}, the median is 6, the fourth largest, and also the fourth smallest, number in the sample.

In case of sequences containing even number of items, we will need to calculate media as the middle point between the two middle values. For example sequence [21, 32, 43, 52, 54, 55, 58, 62, 76, 83] contains 10 items. Therefore the median has to be calculated as the middle point between 54 and 55 what is 54.5

Create a function, that will determine the value, that correspond to median in a given sequence.

Example of function in use:

```
>>> median( [21, 32, 43, 52, 54, 55, 58, 62, 76, 83])
54.5
```

Code Solution

Use dropdown feature below if you want to see, how we wrote the code.

Click to see our solution

Introduction

In order we can work with functions properly, we have to begin to ask ourselves questions like:

- How Python searches for variables?
- If there are different places, where Python can look into, in what order are they searched?
- Can there be multiple separate variables of the same name?

Why these questions?

- Because there are isolated places in Python code / program.
- Because not all variables are visible from all places in Python code.

These isolated places are called **scopes** in Python.

Namespace

Namespace is an important concept in understanding, how Python represents connections between variable names and actual objects. When we assign a value to a variable, Python is creating a connection between that variable object and the assigned value:

```
>>> a = 5
>>> name = 'John'
```

What is actually happening above is that Python creates a **namespace mapping**. We can imagine this mapping as a dictionary, where the variable name is stored as a key and the actual object is the stored as the value.

```
namespace = {'a' : 5, 'name': 'John'}
```

It is important to note that **variable is created (declared) by assigning it a value**.

The important part is that there can be multiple such namespaces in one Python program. For example, Python creates a new namespace, when a function is called. Once the function returns, the namespace is destroyed and the variables and their values no longer exist in the program.

Namespaces have different levels of hierarchy - we call these different levels **scopes**.

What is a scope?

Scopes is a general name for separate namespaces, where the content of some of namespaces is **better accessible than content of other namespaces**. That means there is a kind of a hierarchy between scopes.

Better accesible means for example, that a variable that was created outside a function is accessible from inside the function.

```
>>> name = 'John'
```

```
>>> def func():
...     print(name)
```

Running the above function would look like this:

```
>>> func()
John
```

However it is **not possible to access variables that were created inside a function by expressions outside the function**:

```
>>> def func():
...     name = 'John'
...     print(name)
full_name = name + 'Smith'
```

The result would be a `NameError` :

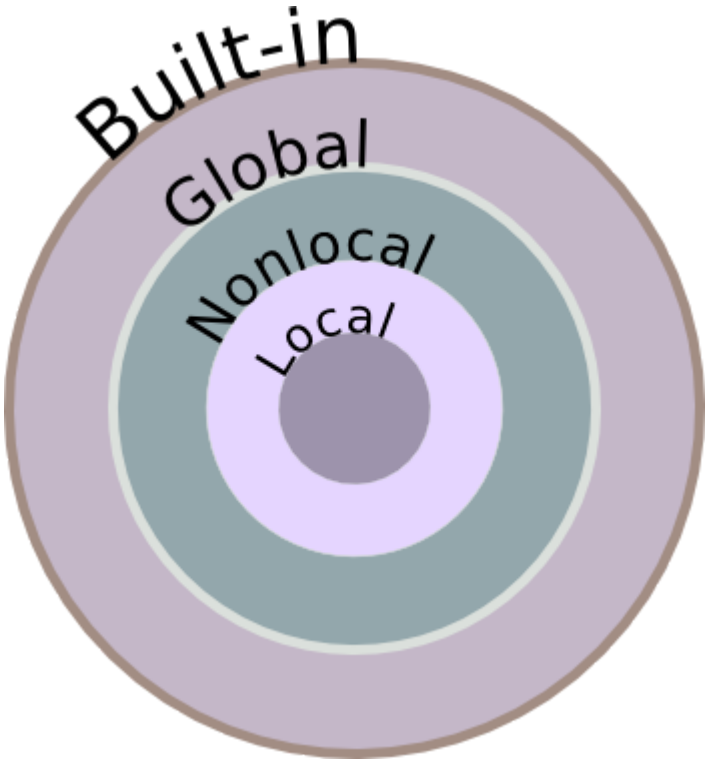
```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'name' is not defined
```

There can be up to four different scopes in a Python program:

- **L**ocal
- **E**nclosing
- **G**lobal
- **B**uilt-in

We will refer to the hierarchy of scopes using the acronym **LEGB**.

We can imagine scopes as spheres, where those inside one sphere have access to items above, but not below:



Built-in & Global Scope

Before we have learned about functions, variables, that existed in our programs, lived in **global** and **built-in** scope. Names that we had already available after initiating Python interpreter belong to **built-in** scope. And example is function `print()`, `sum()`, `all()` etc.

We did not have to define these functions before using them. They were basically **built into Python interpreter**. Therefore (variable) names that come already with Python being turned on, are called **built-ins** and they are stored in **built-in** scope (namespace).

If we wanted to see, what built-in variable are there in running Python program, we can go to our command line (terminal) run Python interpreter and write the following commands:

```
>>> from pprint import pprint as pp
>>> pp(__builtins__.__dict__)
```

What the code above means is that we are importing from the module pprint the function pprint and we'll be using it under the name **pp**. More on importing later.

After running the above commands, you should see pretty printed dictionary of all variables that live in built-in scope. Something like this:

```
{'ArithmeticError': <class 'ArithmeticError'>,
 'AssertionError': <class 'AssertionError'>,
 'AttributeError': <class 'AttributeError'>,
...
 'str': <class 'str'>,
 'sum': <built-in function sum>,
 'super': <class 'super'>,
 'tuple': <class 'tuple'>,
 'type': <class 'type'>,
 'vars': <built-in function vars>,
 'zip': <class 'zip'>}
```

Variables that we have created during our program, were all situated in the **global** scope.

Creating Local Scope

Now, when we begin to learn to define new functions, we have to learn about **local** scope. **Local scope** (a mapping between variable name and the value) is created every time we call a function. This is valid also for those functions we have not defined. Local scope contains mapping between all variables declared inside the function and their values.

For example, if the following function, which purpose is to count, how many times an `item` is present in a `sequence` :

```
1 def count(sequence,item):
2     result = 0
3     for i in sequence:
4         if i == item:
5             result += 1
6     return result
```

... was run:

```
count('Hello', 'l')
```

Python would create a local scope (namespace mapping) at the instant when the function call would be launched. We could imagine this namespace as something like the following dictionary (this is just an illustration for better understanding):

```
count_namespace = {'sequence':'Hello', 'item': 'l', 'result':0}
```

As you can see, also the inputs (function parameters), belong to local function's scope and the variable result has initial value 0.

Global vs Local Scope

PYTHON ACADEMY / 7. FUNCTION SCOPES & INPUTS / FUNCTION SCOPES / GLOBAL VS LOCAL SCOPE

Once we have created local scope in our function, we can start asking ourselves the following questions:

- how can code **access** or even **change** built-in and global variables **from inside of the function**?
- how can code **from global scope access** or even **change** variables that exist in local scope?

Distinguishing Scopes

Right now we distinguish between local, global and built-in variables. In the example code below, we have the following variables

- **local scope** for function **order_sequence**: `my_list` , `i` , `pos`
- **local scope** for function **generate_random_list**: `lst` , `size` , `i`
- **global scope variables**: `random` , `order_sequence` , `generate_random_list` , `l` , `random.randint`

- **built-in variables**: these come into live with turning on Python interpreter - we do not create them and therefore we cannot see them in the code below (but we are using variable name `print` to access the print function)

```
import random
def order_sequence(my_list):
    for i in range(1,len(my_list)):
        pos = i
        while pos > 0 and my_list[pos-1] > my_list[pos]:
            my_list[pos],my_list[pos-1] =  my_list[pos-1], my_list[pos]
            pos -= 1


def generate_random_list(size):

    lst = []
    for i in range(size):
        lst.append(random.randint(1,100))
    return lst

L = generate_random_list(10)
print('Before sorting:',L)
order_sequence(L)
print('After sorting:', L)
```

Global vs Local - Variable Access

Let's again consider the following code. Now, however, we will not pass the variable `l` , that is created inside the **global** scope, into the function call. Also note, that inside the function `order_sequence` we have exchanged variable `my_list` for the global scope variable `l` :

```
import random
def order_sequence():
    for i in range(1,len(l)):
```

```
    for i in range(1,len(l)):
        pos = i
        while pos > 0 and l[pos-1] > l[pos]:
            l[pos],l[pos-1] =  l[pos-1], l[pos]
            pos -= 1
    return l

def generate_random_list(size):
    lst = []
    for i in range(size):
        lst.append(random.randint(1,100))
    return lst

l = generate_random_list(10)
print('Before sorting:',l)
print('After sorting:', order_sequence())
```

Despite the removal of local variable `my_list`, the function still works and now directly accesses variable `l` in the global scope

```
~/PythonBeginner/Lesson5 $ python my_sort.py
Before sorting: [28, 36, 47, 18, 76, 30, 32, 38, 43, 70]
After sorting: [18, 28, 30, 32, 36, 38, 43, 47, 70, 76]
```

This is possible, because the global scope is created first and then, when the function `order_sequence` is called, new local scope for this function is created. That means, that both scopes exist side by side. The only code that is being executed during the existence of local scope is the function's code. Therefore, code from within the function **can access** global variable names.

However, this is **not true in the other direction**. Once the function finishes its execution, its local scope (all its variables) is destroyed. Then the program execution continues in the global scope, from where we cannot access non-existent variables from the `order_sequence` scope. In global scope, we should **not** try to use variable names defined only inside functions, because they do not exist in the global scope.

Access between two function scopes?

This is again not possible. Objects that exist inside `order_sequence` function scope cannot see (access) variables inside `generate_random_list` and vice versa.

Multiple variables with the same name

We can have two variables of the same name, if they were created inside separate namespaces (scopes):

```
>>> name = 'John'
>>> def func(n):
...     name = n
...     print(name)
...
>>> func('Bob')
Bob
>>> name
'John'
```

If we create a variable of the same name in our function as that inside the global or built-in scope, we have to learn, how Python decides, which variable to choose. The choice has to be made, because from inside a function, we can access also global and built-in variables.

Variable Search

This choice is made based on the variable **search through individual namespaces**. What we are interested in is where Python begins its search for the variable name. The search begins in the namespace, where the request for the variable value has been made. If the call for the variable was done **from inside the lowest scope** - function local scope, then **this is the order** in which Python searches through namespaces:

1. **Local** - looks first inside the function namespace - if the variable name has been found, the associated value is returned back and the search stops. If the name has not been found, the search continues to the next level - Global

2. **Global** - again, the search stops and returns the value if the name has been found otherwise the search continues to the built-in scope
3. **Built-in** - if the name has not been found neither in the last scope - built-in scope - Python raises `NameError`

That means that the search stops, where Python finds the requested variable name first and associated value is returned

Code Task

Before you run the following code, try to **guess the output:**

```
1  name = 'John'
2  surname = 'Smith'
3
4  def func():
5      name = 'Bob'
6      fullname = ' '.join((name,surname))
7      print(fullname)
8      print(age)
9
10 func()
```

spustit kód

We can see that when running the function:

1. Variable `name` has been find already inside the function scope (do not forget that a variable is created using assignment statement)
2. Variable `surname` does not exist inside the function scope, but it was found in the global scope and we could use its value
3. Function name `print` has been found in the built-in scope
4. Variable name `age` has not been found in none of the scopes, therefore the NameError is raised

Global vs Local - Change Variable Value

We know, that we can retrieve values from the scopes above our currently executed scope. We have not yet stressed what will happen if we try to assign to the variable from the outter scope a new value.

If we look closer at the code snippet below, we can notice that by assigning a new value to a variable of the same name as the global variable's name, we actually create a new local variable.

```
1  name = 'John'
2  surname = 'Smith'
3
4  def func():
5      name = 'Bob'
6      fullname = ' '.join((name,surname))
7      print(fullname)
8      print(age)
9
10 func()
```

So how can we actually change the value of global variable in this case? We need to use a Python keyword `global` :

```
1 name = 'John'
2 def func():
3     global name
4     name = 'Bob'
5     print(name)
6
7 func()
8 print(name)
```

```
Bob
Bob
```

- This way we declare the specific variable to be global. We have to perform this declaration on **the first line of the function**, which uses it. Global variables can be therefore **changed** from inside of a function if they are declared as `global`
- Changing global variables is **not considered a good practice**. It can cause confusion (code reader does not have to realize that our goal is to change the variable value) and unexpected results in the program. Reasonable use of global state is recommneded rather for advanced programmers for algorithm optimization or reduced complexity.
- Local variables cannot be changed from outside of a function.

Changing built-in variables

PYTHON ACADEMY / 7. FUNCTION SCOPES & INPUTS / FUNCTION SCOPES / CHANGING BUILT-IN VARIABLES

Built-in variable names can be assigned another value from anywhere in the program without having to perform any special declarations. An example would be:

```
>>> sum([1,2,3])
6
>>> sum = 5
>>> sum([1,2,3])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
```

```
TypeError: 'int' object is not callable
>>> sum
5
```

In the example above, we have changed the point of reference for the variable sum. Before it referred to the built-in function, that performs summation of numbers in a collection. Later it referred to integer value 5 and we could not perform call operation with `sum()` anymore.

To regain back access to the original function object, we need to use `del` keyword, used for referrence deletion:

```
>>> del sum
>>> sum([1,2,3])
6
>>> sum
<built-in function sum>
```

Defining Function in a Function

PYTHON ACADEMY / 7. FUNCTION SCOPES & INPUTS / FUNCTION SCOPES / DEFINING FUNCTION IN A FUNCTION

To finish the scope story we still need to explain, what **enclosing scope** is. To understand enclosing scope, we need to know, that functions, can be defined inside functions. Beginner programmers do not use this feature very much, but we will include it here for completeness.

Function inside Function

To build a function inside a function has sense in the two following scenarios:

1. We do not want the inner function to be accessible from outside (we want to isolate it)
2. We want to return the created function

It is hard to find a use case for the first scenario. Why would we write code, that cannot be reused somewhere else? The second case applies to concept of factory functions.

So here is an example of function defining and returning another function:

```
import random
def wrapper():
    start = random.randint(1,10)
    end = random.randint(10,100)
    def inner():
        return range(start,end)
    return inner
```

Now we could store the function object returned by the `wrapper` function inside another variable - e.g. `func`. We can then perform function calls with `func` variable:

```
>>> func = wrapper()
>>> func()
range(4, 55)
>>> func()
range(4, 55)
>>> func = wrapper()
>>> func()
range(9, 34)
>>>
```

From the `inner()` function's point of view, variables `start` and `end` now exist in **enclosing scope**. Code from inside the `inner()` function can access the variables.

With each new call to the `wrapper()` function a new function object, with new enclosing scope variables `start` and `stop` is created. We can see that the first function object we have generated above always returns the same `range(4,55)`. Once we assign the variable `func` a new function object, calls to `func()` generate different `range()`.

Enclosing Scope

PYTHON ACADEMY / 7. FUNCTION SCOPES & INPUTS / FUNCTION SCOPES / ENCLOSING SCOPE

Variables in enclosing scope can be accessed from inside the inner function as we have already demonstrated on the below code.

```
import random
def wrapper():
    start = random.randint(1,10)
    end = random.randint(10,100)
    def inner():
        return range(start,end)
    return inner
```

However, if we wanted to modify those variables inside the local scope of `inner()` function, we would have to declare them as `nonlocal` :

```
import random
def wrapper():

    start = random.randint(1,10)
    end = random.randint(10,100)
    print('Original variable values: start:',start,'end:',end)

    def inner():
        nonlocal start, end
        start += 5
        end += 5
        return range(start,end)

    inner()
    print('Function inner has changed my variable values:
start:',start,'end:',end)

    return inner
```

In the modified code above, we had to call the `inner()` function inside the `wrapper()` in order the values of `start` and `end` change.

Here we can see the code in action:

```
>>> func=wrapper()
Original variable values: start: 10 end: 67
Function inner has changed my variable values: start: 15 end: 72
```

Functions globals() & locals()

In order to find out what variables are accessible from the current scope, we can use `globals()` or `locals()` functions.

Both functions return a dictionary listing all the variable names as dictionary keys and objects themselves as dictionary values. `globals()` function lists only variables created in global scope and `locals()` function lists variables created in current local scope.

If we run these two functions from the global scope, we get the same dictionaries:

```
>>> globals() == locals()
True
```

Let's create a function `func()` that will tell us, what variables are locals in each function's scope:

```
1  glob = 'glob is global variable'
2
3  def func():
4      print('FUNC SCOPE: ', locals())
5
6      def inner_func():
7          print('INNER_FUNC SCOPE: ', locals())
8
9          def basement():
10             print('BASEMENT SCOPE: ', locals())
11
12         basement()
13
14     inner_func()
```

Running the function:

```
>>> func()
FUNC SCOPE:  {}
INNER_FUNC SCOPE:  {}
```

```
BASEMENT SCOPE:  {}
```

There were no assignments performed inside the functions, therefore, the local scope is empty. Now if we create variables inside local scopes, the result will be different:

```
1  glob = 'glob is global variable'
2
3  def func():
4      func_var = 'hello from func'
5      print('FUNC SCOPE: ', locals())
6
7      def inner_func():
8          inner_func_var = 'hello from inner_func'
9          print('INNER_FUNC SCOPE: ', locals())
10
11         def basement():
12             basement_var = 'Hello from the basement'
13             print('BASEMENT SCOPE: ', locals())
14
15         basement()
16
17     inner_func()
```

Running the function:

```
>>> func()
FUNC SCOPE:  {'func_var': 'hello from func'}
INNER_FUNC SCOPE:  {'inner_func_var': 'hello from inner_func'}
BASEMENT SCOPE:  {'basement_var': 'Hello from the basement'}
```

Scopes Summary

PYTHON ACADEMY / 7. FUNCTION SCOPES & INPUTS / FUNCTION SCOPES / SCOPES SUMMARY

In Python, we distinguish among 4 levels of scopes in upwards hierarchy:

1. **Local**
2. **Enclosing**
3. **Global**
4. **Built-in**

Scopes are somehow separate virtual spaces that contain mappings between variable names and values. There cannot be two variables of the same name in one scope. However, we can have variables of the same names in separate scopes (variable name in global scope and then again in local scope).

The topic of scopes is all about what variables can be access resp. change from which point of the code.

Variables

Variables are created using assignment statement. At the same moment, the variable - value mapping is added to the scope, where the assignment has been performed.

Hierarchically lower scopes can access variable in scopes above them, but not vice versa. In order to change reference of global variable inside a running function, we need to declare that variable global using `global` keyword. In order to change reference of variable in enclosing scope, we need to declare that variable `nonlocal` (see additional information section for more).

Variables, that live in built-in scope can be accessed and changed from anywhere in the program. To change built-in variables, we do not have to use any special keyword.

Introduction

Function receives its inputs through the parentheses that follow the function name:

```
>>> sum([1,2,3])
```

Multiple inputs have to be **separated by commas**.

Function inputs have two names in Python:

1. **Parameters**
2. **Arguments**

Parameter

This term is used for function inputs declared inside the function definition:

```
1 def func(parameter1, parameter2):
2     some code
```

Argument

Word argument is used for values that are sent to the function during the function call:

```
>>> argument1 = 5
>>> argument2 = 4
>>> func(argument1, argument2)
```

What actually happens is that variable `argument1` hands over object it refers to the parameter `parameter1` and `argument2` hands over object it refers to the parameter `parameter2`. Once both parameters have collected their values from arguments, we can work with them inside a function.

Ways of passing inputs to a function

PYTHON ACADEMY / 7. FUNCTION SCOPES & INPUTS / FUNCTION INPUTS / WAYS OF PASSING INPUTS TO A FUNCTION

There are two ways, arguments can be passed into a function call:

- **By position**
- **By keyword**

This tells us, how will Python match arguments to function parameters when a function is called.

Passing inputs by position

By default, Python matches arguments to parameters by the order, in which we list the arguments in function call and function definition. So f.e. have a function definition as follows:

```
1 def divisible_in_range(start,stop,divisor):
2     nums = []
3     for n in range(start,stop):
4         if n % divisor == 0:
5             nums.append(n)
6     return nums
```

When we call the above function, we pass it 3 arguments:

```
>>> divisible_in_range(1,11,2)
[2, 4, 6, 8, 10]
```

Python matches arguments to parameters according to the order they are listed:

Parameter	Argument	Matching key
`start`	1	both listed first
`stop`	11	both listed second
`divisor`	2	both listed third

Passing inputs by keyword

Besides matching arguments to parameters by the order, in which they are listed in function call parentheses, we can tell Python explicitly, which argument should be assigned to a given parameter. It can be done by assignment to parameter name:

```
>>> divisible_in_range(divisor=2,stop=11,start=1)
[2, 4, 6, 8, 10]
```

We can see, that now, the arguments do not have to be listed in the order, in which the parameters are defined. Python is given the exact matching from us.

Distinction among passing arguments by position vs. by keyword is analogous to the difference among sequences and dictionaries. When using dictionaries, we do not have to care about the order, because what matters is the key.

Ways of passing inputs determine, which parameter will be matched with which argument value. Normally we pass arguments by position. Sometimes it is however handy to use keyword argument passing. We will learn about that in the following section.

Number of inputs

The number of arguments a function receive depends on the specific function definition. So we have

1. function, that **do not accept any inputs**

2. functions that accept **fixed number of inputs** (one and more)
3. functions that accept **variable number of inputs**

No inputs

```
1 def dummy_func():
2     return 'Hello'
```

```
>>> dummy_func()
'Hello'
```

Fixed number of inputs

```
>>> abs(-6)
6
>>> range(1,100,2)
range(1,100,2)
```

Function `abs()` is defined with only one parameter. If we pass it more than one value inside function call parentheses, we will get an error:

```
>>> abs(1,3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: abs() takes exactly one argument (2 given)
```

To find how many, inputs a built-in function accepts, you can check the Python documentation

Variable number of inputs

```
>>> print('Hello','Mr.', 'This', 'and', 'That')
Hello Mr. This and That
```

We can pass as many objects separated by comma into `print()` function, as we wish. You can try it yourself.

Fixed number of arguments

We determine, whether a function will take fixed or variable number of inputs already **inside function definition** - inside the parentheses.

We know that a function takes a fixed number of inputs when there is **no * or ** operator in front of any parameter**. That means all the function definitions we have seen so far, required fixed number of inputs.

```
def divisible_in_range(start,stop,divisor):
    nums = []
    for n in range(start,stop):
        if n % 2 == 0:
            nums.append(n)
    return nums
```

Functions that require fixed number of inputs, will raise errors if less or more arguments will be passed into that function's call:

```
>>> divisible_in_range()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: divisible_in_range() missing 3 required positional arguments:
'start', 'stop', and 'divisor'
```

Functions defined with empty parentheses do not accept any inputs, and if we pass any argument into the function call, we will get the same error as the one above.

```
def dummy_func():
    return 'Hello'
```

```
>>> dummy_func(12)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: dummy_func() takes 0 positional arguments but 1 was given
```

Default parameter value

Some function parameters are in majority of function calls passed the same value. In that case, it would be much more comfortable to specify this value as **default** in order avoid passing it all the time. Function allows such default parameters. Default parameter is a parameter that is assigned a value in the function definition. Such value will be used, if not enough arguments is sent to a function call:

```
1 def multiply(num, multiplier=2):
2     return num * multiplier
```

If we do not pass value for the second positional parameter, Python will use the default value:

```
>>> multiply(4)
8
>>> multiply(6)
12
```

On the other hand, we cannot pass more arguments, then there are parameters, even though we are using default parameter values:

```
>>> multiply(6,7,9)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: multiply() takes from 1 to 2 positional arguments but 3 were
given
```

Variable number of arguments

To have a function that accepts variable number of arguments means, that the **function can be passed 0 or more inputs**. An example is the `print()` function:

```
>>> print()
>>> print('Hello', 'today', 'is', 'a', 'nice', 'day')
Hello today is a nice day
```

Operator *

If we looked at the Python documentation on print(), we would see that the function can be called with the following inputs:

```
print(*objects, sep=' ', end='\n', file=sys.stdout, flush=False)
```

What tells Python, that `print()` can be passed **any number of positional inputs**? It is the `*` operator in front of the first parameter - `objects` .

Operator **

Besides the single star operator, there can be also double star `**` operator. That tells Python, that **any number of keyword arguments** can be passed to such a function. An example of such a built-in function is the `dict()` function, that creates a dictionary. Check the Python documentation for details:

```
>>> dict(name = 'John', surname = 'Smith', salary= 4323)
{'surname': 'Smith', 'salary': 4323, 'name': 'John'}
```

In Python documentation we can see that the `dict()` function is defined with double-starred parameter `**` :

```
dict(**kwargs)
```

Variable number of positional arguments

PYTHON ACADE… / 7. FUNCTION SCOPES & INP… / FUNCTION INPU… / VARIABLE NUMBER OF POSITIONAL ARGUME…

Variable number of positional arguments has to be signaled with single star `*` in front of the function parameter.

```
def func(*args):
    print(args)
```

If we ran the above function passing it multiple inputs, we would find out, that the parameter `args` collects all the arguments into a single tuple:

```
>>> func(1,'Hello', [1,2,3])
(1, 'Hello', [1, 2, 3])
```

In Python community, parameter, that collects all the resting positional arguments is called `args` .

How do we actually work with the starred argument?

As the content of the starred parameter will be **always a tuple**, we can apply all the tuple operations to it or even convert it into a list if needed. Below, we have created our own `sum` function that takes any number of inputs. If the inputs are all numbers, we get the result

```
def my_sum(*args):
    result=0
    for n in args:
        result += n
    return result
```

```
>>> my_sum(1,2,3,4,5,6)
21
```

Position of starred parameter

Such a starred parameter has to be listed as the **last one among all the arguments** we want to be treated as positional:

```
def func(prefix, suffix, *args):
    for arg in args:
        print(prefix + arg + suffix)
```

```
>>> func('in','ly','competent', 'formal', 'credib')
incompetently
informally
```

```
incredibly
```

What actually happens in the example above is that the parameter `args` collects all the redundant positional inputs into a tuple. Fixed position parameters (`prefix` and `suffix`) have taken arguments that correspond to their position and the rest has been stored as a tuple into `args` parameter.

We can see here a link to extended collection assignment:

```
>>> prefix, suffix, *args = 'in','ly','competent', 'formal', 'credib'
>>> args
['competent', 'formal', 'credib']
>>> prefix
'in'
>>> suffix
'ly'
```

The only difference is that extended collection assignment operation creates a list and stores it in starred variable, meanwhile in function calls, a tuple is created.

Variable number of keyword arguments

PYTHON ACADE… / 7. FUNCTION SCOPES & INPU… / FUNCTION INPU… / VARIABLE NUMBER OF KEYWORD ARGUME…

Variable number of keyword arguments has to be signaled with double star `**` in front of the function parameter.

```
1 def func(**kwargs):
2     print(kwargs)
```

We say keyword arguments, because these have to be passed only using `keyword=value` syntax

```
>>> func(name='Bob', city='London')
{'city': 'London', 'name': 'Bob'}
```

Python collects all the **key-value pairs and stores them in a dictionary**. Therefore we can perform all the **dictionary operations** on the contents of the `kwargs` variable.

If we did not specify the names of the parameters we want to assign arguments to, we get an error:

```
>>> func('Bob', 'London')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: func() takes 0 positional arguments but 2 were given
```

Again, keys, that do not match any parameter are collected into `kwargs` parameter. In Python community, parameter, that collects all the resting keyword arguments is called kwargs (**k**ey**w**ord **arg**ument**s**).

Position of double-starred parameter

Double starred parameter has to be listed as the **last one among all the parameters (positional or keyword)**.

```
1 def func(prefix, suffix, *args, **kwargs):
2     for arg in args:
3         if 'capital' in kwargs and kwargs['capital'] == True:
4             arg = arg.upper()
5         print(prefix + arg + suffix)
```

The function above is testing, whether the dictionary `kwargs` contains a key `capital` and whether its value is equal to `True` . If so, then all the middle strings are capitalized:

```
>>> func('in','ly','competent', 'formal', 'credib', capital=True)
inCOMPETENTly
inFORMALly
inCREDIBly
```

Forcing to pass arguments by keyword

The order of placing function parameters as we know so far is:

1. First we pass fixed number of required positional arguments
2. After that comes single-starred parameter collecting the rest of the arguments passed by the position (without keyword)
3. After that comes double-starred parameter, that collects all the arguments passed by keyword, if no parameter of such name is defined

There can be **only one single-starred and only one double-starred parameter in a function definition.**

What will happen we put another non-starred parameter after the `*args` ?

```
def func(prefix, *args, suffix):
    for arg in args:
        print(prefix + arg + suffix)
```

To make the story complete, we have to say, that between the `*args` and `**kwargs` arguments, we can still put **parameter names**. We could use this knowledge to improve the design of the function from the previous section:

```
def func(prefix, suffix, *args, **kwargs):
    for arg in args:
        if 'capital' in kwargs and kwargs['capital'] == True:
            arg = arg.upper()
        print(prefix + arg + suffix)
```

We do not want to look for parameter `capital` in `kwargs` , but we want to have it explicitly listed among function parameters

```
def func(prefix, suffix, *args, capital):
    for arg in args:
        if capital:
            arg = arg.upper()
        print(prefix + arg + suffix)
```

Now the parameter that is following the single-starred parameter `args` has to be passed to function call **only** using `keyword=value` form.

```
>>> func('in','ly','competent', 'formal', 'credib', capital=True)
inCOMPETENTly
inFORMALly
inCREDIBly
```

We therefore can **force some required parameters to be passed by keyword** if we put those parameters after the single star in a function definition. If we did not use keyword=value form to pass the argument to the function, we would get an error:

```
>>> func('in','ly','competent', 'formal', 'credib', True)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: func() missing 1 required keyword-only argument: 'capital'
```

Required keyword parameters are used to improve the readability of function calls. We can see that the function call that contains only value `True` instead of `capital=True` is not making clear, what parameter is set to the value True. This is the specific scenario, when we should use required keyword arguments.

We can summarize now that function parameters can be passed in the following order:

- **required** positional that we have to pass (we have to pass at least as many values as there are positional arguments)
- container that collects the rest of **optional** values **passed by position**
- **required** keyword-only arguments that have to be passed (all the parameters listed after the single star parameter)
- container that collects the rest of **optional** the **keyword=value pairs** into a dictionary

Unpacking inputs

PYTHON ACADEMY / 7. FUNCTION SCOPES & INPUTS / FUNCTION INPUTS / UNPACKING INPUTS

Under the term unpacking we can imagine a distribution of values among function parameters, during the function call.

Let's say we have a function that requires 3 inputs:

```
def list_divisibles(start,stop,divisor):
    divisibles=[]
    for num in range(start,stop):
        if num % divisor == 0:
            divisibles.append(num)
    return divisibles
```

We could pass it inputs using 3 separate variables or values:

```
>>> list_divisibles(3,10,3)
[3, 6, 9]
```

Or we could collect the values inside a sequence and then unpack during the function call:

```
>>> args = [3,10,3]
>>> list_divisibles(*args)
[3, 6, 9]
```

The first value in `args` has been matched to the first parameter of the function, second value to the second parameter etc. Usually the variable `args` that is later unpacked is built in some other place of code.

Unpacking is used during function call, not function definition. It also uses single-starred (works only with sequences) or double-starred (unpacks only dictionaries) inputs. We can also demonstrate double-starred unpacking on the example above:

```
>>> args = {'start':3, 'stop':10, 'divisor':3}
>>> list_divisibles(**args)
[3, 6, 9]
```

Unpacking could be used on optional parameters too - all unpacked values will be packed again into the starred parameter

```
def my_sum(*args):
    result=0
    for n in args:
        result += n
    return result
```

```
>>> my_sum(*range(50))
1225
```

Passing a function as input

Functions are objects as any other in Python. Therefore we can take a variable name, that references a function object, and send it into a function call:

```
1 def wrapper(func, argument):
2     return func(argument)
```

The above wrapper function is able to call any function that takes exactly one argument:

```
>>> wrapper(abs,-45)
45
```

This is important, because passing optional (starred) arguments happens alongside to passing function objects to functions calls. And that is the use case we discuss in the following section.

Optional Parameters - Use case

Starred parameters in a function definition are in fact optional parameters. Single and double-starred parameters are usually used in contexts, where they are passed alongside a function object into a function call. Why would we pass a function into a function call?

Maybe we want to **measure the time, it takes to a function to execute**. We pass the starred parameters into a function inside timer call. We are using `time.time()` function to determine the time at two points. We then make the difference between those two points in time to calculate time needed to run a function `func()` :

```
import time
def timer(func, *args, **kwargs):

    start_time = time.time()
    func(*args, **kwargs)
    total_time = time.time() -start_time

    return total_time
```

Or we may want have a functionality, that **generates html tags including variable number of tag attributes**:

```
def make_tag(tag, content, **attributes):
    attrs= []
    for k,v in attributes.items():
        attrs.append(str(k) + '=' + str(v))
    attrs_str = ' '.join(attrs)
    return '<' + tag + ' ' + attrs_str + '>' + content + '</' + tag + '>'
```

```
>>> make_tag('a','Python Documentation', href =
'https://docs.python.org/3/', target = '_blank')
'<a href=https://docs.python.org/3/ target=_blank>Python
Documentation</a>'
```

In general, we use optional arguments in cases, when there is no limit or it is unknow, how many inputs a function will process.

Starred arguments - function call = unpacking

The above example with `make_tag` function can be improved by using unpacking inside the function call. First we could build a dictionary of attribute:value pairs and unpack them directly to the `make_tag()` function. Actually unpacking is often combined with functions that accept variable number of arguments (as make_tag does):

```
attributes = {'href':'https://docs.python.org/3/', 'target':'_blank'}
>>> make_tag('a','Python Documentation',**attributes)
'<a href=https://docs.python.org/3/ target=_blank>Python
Documentation</a>'
```

Another case, where unpacking is used a lot is string formatting. We will learn about string formatting in upcoming lessons but at the moment we should know, that formatting allows us to save space in our code by not having to write so many concatenation operations. Here we often unpack dictionaries, lists or tuples in order to distribute the value they contain among formatting function parameters:

```
>>> employee = {'name':'Bob','surname':'Smith','salary':2000}
>>> 'Employee: {surname}, {name}, Earns: {salary}'.format(**employee)
'Employee: Smith, Bob, Earns: 2000'
```

Contents of the dictionary `employee` has been split into individual parameter=arguments pairs for the function `format()` as if we would passed keyword arguments inside- `format(name = 'Bob', surname = 'Smith', salary =2000)`

Function Scopes

PYTHON ACADEMY / 7. FUNCTION SCOPES & INPUTS / QUIZ / FUNCTION SCOPES

1/11

What does the term **namespace** mean?

A. List in which all the global variable names are stored during the program execution

B. It is an abstract term representing mapping between variable names and objects created during the program execution

C. Amount of memory left for our program, during its execution

D. Tuple of tuples, where each tuple gathers related variable names

Function Inputs

PYTHON ACADEMY / 7. FUNCTION SCOPES & INPUTS / QUIZ / FUNCTION INPUTS

1/7

What will happen if the following function is called in the way below?:

```
1 def func(a,b):
2     return a + b
```

```
>>> func()
```

A. Value 0 is returned

B. NameError: name 'a' and 'b' are not defined

C. TypeError: func() missing 2 required positional arguments: 'a' and 'b'

D. Sum of values a and b

Luhn test

Your task is to implement a function, that will perform credit card number validation based on so called Luhn test. The Luhn test is used by some credit card companies to distinguish valid credit card numbers from what could be a random selection of digits.

Validation by Luhn test

1. Reverse the order of the digits in the number,
2. take the first, third and every other odd digit in the reversed digits and sum them to form the partial sum `s1`,
3. take the second, fourth and every other even digit in the reversed digits and:
   a. multiply each digit by two,
   b. if the result of digit multiplication is greater than nine (more than 1 digit number) then sum the digits to form the partial sums for each even digit multiplication,
   c. and sum the partial sums of the even digits to form `s2`,
4. if `s1 + s2` ends in zero then the original number is in the form of a valid credit card number as verified by the Luhn test.

The output of the program should be `True` (valid card number) or `False` (invalid card number).

Example

Complicated, huh? Let's try to take a trial number **49927398716**:

Step	Description	Results
1.	Reverse the digits	61789372994
2.	Sum the odd digits	6 + 7 + 9 + 7 + 9 + 4 = 42 = `s1`
3.	The even digits	1, 8, 3, 2, 9
3a.	Each even digit x 2	2, 16, 6, 4, 18

Step	Description	Results
3b.	Sum the digits of each multiplication	2, 7, 6, 4, 9
3c.	Sum the the partial sums	2 + 7 + 6 + 4 + 9 = 28 = `s2`
4.	The sum s1+s2 ends with digit 0	`s1 + s2 = 70`
Output	The function returns value	`True`

Example of using the function:

```
>>> luhn(61789372994)
True
```

Online Python Editor

```
1
```

spustit kód

Code Solution

Use dropdown feature below if you want to see, how we wrote the code.

Click to see our solution

Anagram

PYTHON ACADEMY / 7. FUNCTION SCOPES & INPUTS / HOME EXERCISES / ANAGRAM

When two or more words are composed of the same characters, but in a different order, they are called **anagrams**. An anagram is direct word switch or word play, the result of rearranging the letters of a word or phrase to produce a new word or phrase, using all the original letters exactly once.

For example the word 'eat' has the following anagrams in english:

1. ate
2. tea

There is also a page, that generates anagrams.

The goal in this assignment is to create a function that takes a list of 2 or more strings as input and returns boolean value telling us, whether all the strings inside the list are are anagrams or not. If the input is an empty string, the output should be `False`. If the input list contains one word, then the result should be `True`.

Example of function in use:

```
>>> all_anagrams(['ship', 'hips'])
```

```
True
>>> all_anagrams(['ship', 'hips', 'name'])
False
>>> all_anagrams(['ship'])
True
>>> all_anagrams([])
False
```

Online Python Editor

```
1
```

spustit kód

Code Solution

Use dropdown feature below if you want to see, how we wrote the code.

Click to see our solution

Collecting emails

Your task is to create multiple functions that will work together in one program and will produce a result in form of a dictionary representing the following mapping:

- emails that contain numeric characters found in the string `my_str`
- all the email domains (part after the '@' symbol) found in the string `my_str`

What the multiple functions could do?

1. extract all the emails from the string `my_str`
2. collect only email containing numeric characters
3. extract all email domains

And here we have the variable `my_str` containing the target string:

```
my_str='''Lorem ipsum dolor sit amet, consectetur adipiscing
        elit. Mauris vulputate lacus id eros consequat tempus.
        Nam viverra velit sit amet lorem lobortis, at tincidunt
        nunc ultricies. Duis facilisis ultrices lacus, id
        tiger123@email.cz auctor massa molestie at. Nunc tristique
        fringilla congue. Donec ante diam cnn@info.com, dapibus
        lacinia vulputate vitae, ullamcorper in justo. Maecenas
        massa purus, ultricies a dictum ut, dapibus vitae massa.
        Cras abc@gmail.com vel libero felis. In augue elit, porttitor
        nec molestie quis, auctor a quam. Quisque b2b@money.fr
        pretium dolor et tempor feugiat. Morbi libero lectus,
```

```
        porttitor eu mi sed, luctus lacinia risus. Maecenas posuere
        leo sit amet spam@info.cz. elit tincidunt maximus. Aliquam
        erat volutpat. Donec eleifend felis at leo ullamcorper
    cursus.
        Pellentesque id dui viverra, auctor enim ut, fringilla est.
        Maecenas gravida turpis nec ultrices aliquet.'''
```

Example of running the program:

```
~/PythonBeginner/Lesson5 $ python collect_email.py
{'domains': ['email.cz', 'info.com,', 'gmail.com', 'money.fr',
'info.cz.'],
'emails_with_nums': ['tiger123@email.cz', 'b2b@money.fr']}
```

Online Python Editor

Our online editor is unsuitable for this kind of task. Considering the scope, it'll be better to do it in you own local code editor. In the future, if there is no Online Python Editor available, it's a sign that you should do the task locally.

Of course, if you've been doing all the task on your computer, that is fine too and you can just keep doing that :)

Code Solution

Use dropdown feature below if you want to see, how we wrote the code.

Click to see our solution

Prime number

PYTHON ACADEMY / 7. FUNCTION SCOPES & INPUTS / HOME EXERCISES / PRIME NUMBER

Your goal in this task is to create two functions:

1. function that will list all the prime numbers up to specified limit
2. function that will tell, whether a number is prime

Algorithm to list prime numbers

To generate a list of prime numbers, you will probably want to follow algorithm designed by Eratosthenes:

1. Create a list of consecutive integers from 2 through n: (2, 3, 4, ..., n).
2. Initially, let p equal 2, the smallest prime number.
3. Enumerate the multiples of p by counting to n from 2p in increments of p, and erase them from the list of generated numbers (these will be 2p, 3p, 4p, ...; the p itself should not be erased).
4. Find the first number greater than p in the list that is not marked. If there was no such number, stop. Otherwise, let p now equal this new number (which is the next prime), and repeat from step 3.
5. When the algorithm terminates, the numbers remaining not erased in the list are all the primes below n.

Example of using `is_prime()` function:

```
>>> is_prime(54)
False
>>> is_prime(53)
True
```

Online Python Editor

```
1
```

spustit kód

Code Solution

Use dropdown feature below if you want to see, how we wrote the code.

Click to see our solution

Caesar cipher

PYTHON ACADEMY / 7. FUNCTION SCOPES & INPUTS / HOME EXERCISES / CAESAR CIPHER

The following function should provide functionality to encode message to Caesar cipher. The method is named after Julius Caesar, who used it in his private correspondence.

We're talking about a type of substitution cipher in which each letter in the plaintext is replaced by a letter some fixed number of positions **up or down the alphabet**.

For example:

- with a LEFT shift of 3 (offset -3), D would be replaced by A, E would become B, and so on.
- with a RIGHT shift of 3 (offset 3), D would be replaced by G, E would become H, and so on.

Example of using the function:

```
>>> message = 'abc def ghi jkl mno pqr stu vwx Yz'
>>> caesar(message,2)
'cde fgh ijk lmn opq rst uvw xyz Ab'
>>> caesar(message,-2)
yza bcd efg hij klm nop qrs tuv Wx
```

Online Python Editor

```
1
```

spustit kód

Code Solution

Use dropdown feature below if you want to see, how we wrote the code.

Click to see our solution

Coins

PYTHON ACADEMY / 7. FUNCTION SCOPES & INPUTS / HOME EXERCISES / COINS

Create a function that will act as a ticket machine that returns **the least possible amount of coins**.

Our machine should work with coins of the following denominations: `1, 2, 5, 10, 20, 50`

If the amount to be returned by the machine is 124, the returned coins should be: **two 50, one 20, two 2**

Example of function in use:

```
>>> coins(124)
{50:2, 20: 1, 2:2}
```

Online Python Editor

```
1
```

spustit kód

Code Solution

Use dropdown feature below if you want to see, how we wrote the code.

Click to see our solution

Hangman

PYTHON ACADEMY / 7. FUNCTION SCOPES & INPUTS / HOME EXERCISES / HANGMAN

The goal of this exercise is to implement the hangman game. The game is played by 2 players, in this case by the computer and the human. The computer selects a secret word and the human tries to guess it by suggesting letters or numbers, within a certain number of guesses.

The word to guess is represented by a row of dashes, representing each letter of the word.

Each time the human player suggests a letter that is not present in the guessed word, the counter of incorrect guesses is increased by one.

Computer wins and the game ends if the number of incorrect guesses reaches specified amount. The human wins if the whole word is guessed before reaching the limit of incorrect guesses.

You shall decide what the limit of incorrect guesses should be.

Example of running the program:

```
~/PythonBeginner/Lesson5 $ python hangman.py
I am thinking of a word. What word is it?:

_ _ _ _ _ _ _
Guess a letter (9 guesses available): 'C'
No, the letter 'C' is not in my word

_ _ _ _ _ _ _
Guess a letter (8 guesses available): 'B'
No, the letter 'B' is not in my word

_ _ _ _ _ _ _
Guess a letter (7 guesses available): 'H'
Yes, there is 1 letter 'H'
H _ _ _ _ _ _
Guess a letter (7 guesses available): 'a'
Yes, there are 2 letters 'a'
H a _ _ _ a _
```

Code Solution

Use dropdown feature below if you want to see, how we wrote the code.

Click to see our solution

Statistics

Our goal is to implement a program, that will be asking us, what statistical measure we want to be calculated and then print the result to the terminal. You can also do the advanced version of this exercise. This version should include all the functionality bellow:

- **sum** - sum of values in a list or tuple
- **count** - count of a given item in a list or tuple,
- **mean** - average value in a given list or tuple
- **modus** - item, that occurs most frequently in a list or tuple
- **median** - middle point in an ordered sequence of values

The program could look something like this:

```
~/PythonBeginner/Lesson5 $ python statistics_easy.py
What you want to calculate (select a number or "q" to quit)?
| SUM | COUNT | MEAN | MODUS| MEDIAN |
sum
RESULT of SUM: 398
Press ENTER to continue
What you want to calculate (select a number or "q" to quit)?
| SUM | COUNT | MEAN | MODUS| MEDIAN |
count
Please enter a number you want to find:
26
RESULT of COUNT: 2
Press ENTER to continue
What you want to calculate (select a number or "q" to quit)?
| SUM | COUNT | MEAN | MODUS| MEDIAN |
mean
RESULT of MEAN: 26.533333333333335
Press ENTER to continue
```

Code Solution

Use dropdown feature below if you want to see, how we wrote the code.

Click to see our solution

Statistics [H]

Create a program that will calculate descriptive statistics for a given dataset. The program should calculate the result from a dataset that should be generated using the code from previous lesson - home exercise Generate data. We recommended you save the in `generate_data.py` module.

In the basic version of this exercise, we have already created functions, that calculate the following descriptive measures:

- **count** - count of a given item in dataset,
- **sum** - sum of values of related to a specific item
- **avg** - average value for a given item
- **modus** - item, that occurs most frequently in a column
- **median** - middle point in an ordered sequence of values

Our main goal now, is to integrate all the functionality into one program. The program could look something like this:

```
$ python statistics.py
Dear user, to use this app correctly follow these rules:
 1. First select a calculation that is to be performed:
    |1-sum|2-count|3-mean|4-modus|5-median|
 2. Then select column that will serve as criterion for
    which the stats is to be computed
    |1-Name|2-Item|
 3. Select a column that will serve as input on which the
```

```
    calculation will be performed.
    The following calculations can be applied to following columns:
    1-Name|2-Item|3-Amount|4-Unit_Price|5-Total_Price|
 4. The result table could look like this:
|            Name            |  Total_Price-SUM   |
=======================================================
|Pate, Ashley                :       1713.15       |
|Conyard, Phil               :       8654.83       |
|Woodison, Annie             :        4235.4       |
|Bettison, Elnora            :       17943.78      |
|Idalia, Craig               :       5713.95       |
|McShee, Glenn               :       6398.76       |
|Skupinski, Wilbert          :       10342.91      |
|Doro, Jeffrey               :       7556.12       |
=======================================================
What you want to calculate - select a number or press "q" to quit?
|1-sum|2-count|3-mean|4-modus|5-median|
1
Select a criterion:
|1-Name|2-Item|

1
What column number?
|1-Name|2-Item|3-Amount|4-Unit_Price|5-Total_Price|
5
|            Name            |  Total_Price-SUM   |
=======================================================
|Idalia, Craig               :       5713.95       |
|Woodison, Annie             :        4235.4       |
|Doro, Jeffrey               :       7556.12       |
|McShee, Glenn               :       6398.76       |
|Conyard, Phil               :       8654.83       |
|Bettison, Elnora            :       17943.78      |
|Pate, Ashley                :       1713.15       |
|Skupinski, Wilbert          :       10342.91      |
What you want to calculate - select a number or press "q" to quit?
|1-sum|2-count|3-mean|4-modus|5-median|
```

Code Solution

Use dropdown feature below if you want to see, how we wrote the code.

Click to see our solution

DALŠÍ LEKCE