

## Лабораторная работа №4

**Тема:** «Работа с функциями. Хранений функций. Пространство имен. Создание, поиск и использование модулей. Описание основных встроенных модулей».

### Функции

Функции представляют блок кода, который выполняет определенную задачу и который можно повторно использовать в других частях программы. Формальное определение функции:

```
def имя_функции ([параметры]):  
    инструкции
```

Определение функции начинается с выражения *def*, которое состоит из имени функции, набора параметров в скобках и двоеточия. Параметры в скобках необязательны. Со следующей строки идет блок инструкций, которые выполняет функция. Все инструкции функции имеют отступы от начала строки.

Например, определение простейшей функции:

```
def say_hello():  
    print("Hello")
```

Функция называется *say\_hello*. Она не имеет параметров и содержит одну единственную инструкцию, которая выводит на консоль строку *"Hello"*.

Для вызова функции указывается имя функции, после которого в скобках идет передача значений для всех ее параметров. Например:

```
def say_hello():  
    print("Hello")
```

```
say_hello()  
say_hello()  
say_hello()
```

Здесь три раза подряд вызывается функция `say_hello`. В итоге мы получим следующий консольный вывод:

```
Hello  
Hello  
Hello
```

Теперь определим и используем функцию с параметрами:

```
def say_hello(name):  
    print("Hello,", name)  
  
say_hello("Tom")  
say_hello("Bob")  
say_hello("Alice")
```

Функция принимает параметр `name`, и при вызове функции мы можем передать вместо параметра какое-либо значение:

```
Hello, Tom  
Hello, Bob  
Hello, Alice
```

### **Значения по умолчанию**

Некоторые параметры функции мы можем сделать необязательными, указав для них значения по умолчанию при определении функции. Например:

```
def say_hello(name="Tom"):  
    print("Hello,", name)  
  
say_hello()  
say_hello("Bob")
```

Здесь параметр `name` является необязательным. И если мы не передаем при вызове функции для него значение, то применяется значение по умолчанию, то есть строка `"Tom"`.

## Именованные параметры

При передаче значений функция сопоставляет их с параметрами в том порядке, в котором они передаются. Например, пусть есть следующая функция:

```
def display_info(name, age):  
    print("Name:", name, "\t", "Age:", age)  
  
display_info("Tom", 22)
```

При вызове функции первое значение "Tom" передается первому параметру – параметру *name*, второе значение – число 22 передается второму параметру – *age*. И так далее по порядку. Использование именованных параметров позволяет переопределить порядок передачи:

```
def display_info(name, age):  
    print("Name:", name, "\t", "Age:", age)  
  
display_info(age=22, name="Tom")
```

Именованные параметры предполагают указание имени параметра с присвоением ему значения при вызове функции.

## Неопределенное количество параметров

С помощью символа звездочки можно определить неопределенное количество параметров:

```
def custom_sum(*params):  
    result = 0  
    for n in params:  
        result += n  
    return result  
  
sumOfNumbers1 = custom_sum(1, 2, 3, 4, 5)      # 15  
sumOfNumbers2 = custom_sum(3, 4, 5, 6)        # 18  
print(sumOfNumbers1)  
print(sumOfNumbers2)
```

В данном случае функция *custom\_sum* принимает один параметр – *\*params*, но звездочка перед названием параметра указывает, что фактически на место этого параметра мы можем передать неопределенное количество значений или набор значений. В самой функции с помощью цикла *for* можно пройтись по этому набору и произвести с переданными значениями различные действия.

Например, в данном случае возвращается сумма чисел.

### Возвращение результата

Функция может возвращать результат. Для этого в функции используется оператор *return*, после которого указывается возвращаемое значение:

```
def exchange(usd_rate, money):  
    result = round(money/usd_rate, 2)  
    return result  
  
result1 = exchange(60, 30000)  
print(result1)  
result2 = exchange(56, 30000)  
print(result2)  
result3 = exchange(65, 30000)  
print(result3)
```

Поскольку функция возвращает значение, то мы можем присвоить это значение какой-либо переменной и затем использовать ее: `result2 = exchange(56, 30000)`.

В Python функция может возвращать сразу несколько значений:

```
def create_default_user():  
    name = "Tom"  
    age = 33  
    return name, age  
  
user_name, user_age = create_default_user()  
print("Name:", user_name, "\t Age:", user_age)
```

Здесь функция *create\_default\_user* возвращает два значения: *name* и *age*. При вызове функции эти значения по порядку присваиваются переменным *user\_name* и *user\_age*, и мы их можем использовать.

### Функция main

В программе может быть определено множество функций. И чтобы всех их упорядочить, хорошей практикой считается добавление специальной функции *main*, в которой потом уже вызываются другие функции:

```
def main():
    say_hello("Tom")
    usd_rate = 56
    money = 30000
    result = exchange(usd_rate, money)
    print("К выдаче", result, "долларов")

def say_hello(name):
    print("Hello,", name)

def exchange(usd_rate, money):
    result = round(money/usd_rate, 2)
    return result

# Вызов функции main
main()
```

## Анонимные функции

Анонимные функции могут содержать лишь одно выражение, но и выполняются они быстрее. Анонимные функции создаются с помощью инструкции *lambda*. Кроме этого, их не обязательно присваивать переменной, как делали мы инструкцией *def func()*:

```
func = lambda x, y: x + y
func(1, 2)           # 3
func('a', 'b')      # 'ab'
(lambda x, y: x + y)(1, 2)    # 3
(lambda x, y: x + y)('a', 'b') # 'ab'
```

## Функциональное программирование

Функциональным называется такой подход к процессу программирования, в котором программа рассматривается как вычисление математических функций, при этом не используются состояния и изменяемые объекты. Как правило, когда говорят о элементах функционального программирования в Python, то подразумеваются следующие функции: *lambda*, *map*, *filter*, *reduce*, *zip*.

### Функция map

В Python функция *map* принимает два аргумента: функцию и аргумент составного типа данных, например, список. *map* применяет к каждому элементу списка переданную функцию. Например, вы прочитали из файла список чисел, изначально все эти числа

имеют строковый тип данных, чтобы работать с ними, нужно превратить их в целое число:

```
old_list = ['1', '2', '3', '4', '5', '6', '7']

new_list = []
for item in old_list:
    new_list.append(int(item))

print(new_list) # [1, 2, 3, 4, 5, 6, 7]
```

Тот же эффект мы можем получить, применив функцию *map*:

```
old_list = ['1', '2', '3', '4', '5', '6', '7']
new_list = list(map(int, old_list))
print(new_list) # [1, 2, 3, 4, 5, 6, 7]
```

Как видите такой способ занимает меньше строк, более читабелен и выполняется быстрее. *map* также работает и с функциями, созданными пользователем:

```
def miles_to_kilometers(num_miles):
    """ Converts miles to the kilometers """
    return num_miles * 1.6

mile_distances = [1.0, 6.5, 17.4, 2.4, 9]
kilometer_distances = list(map(miles_to_kilometers, mile_distances))
print(kilometer_distances) # [1.6, 10.4, 27.84, 3.84, 14.4]
```

А теперь то же самое, только используя *lambda* выражение:

```
mile_distances = [1.0, 6.5, 17.4, 2.4, 9]
kilometer_distances = list(map(lambda x: x * 1.6, mile_distances))
print(kilometer_distances) # [1.6, 10.4, 27.84, 3.84, 14.4]
```

Функция *map* может быть так же применена для нескольких списков, в таком случае функция-аргумент должна принимать количество аргументов, соответствующее количеству списков:

```
l1 = [1,2,3]
l2 = [4,5,6]

new_list = list(map(lambda x,y: x + y, l1, l2)) # [5, 7, 9]
```

Если же количество элементов в списках совпадать не будет, то выполнение закончится на минимальном списке:

```
l1 = [1,2,3]
l2 = [4,5]

new_list = list(map(lambda x,y: x + y, l1, l2)) # [5,7]
```

### Функция *filter*

Функция *filter* предлагает элегантный вариант фильтрации элементов последовательности. Принимает в качестве аргументов функцию и последовательность, которую необходимо отфильтровать:

```
mixed = [1, 2, 3, -2, -3, 20, 5]
positive = list(filter(lambda x: x > 0, mixed))
print(positive) # [1, 2, 3, 20, 5]
```

Обратите внимание, что функция, передаваемая в *filter* должна возвращать значение *True* / *False*, чтобы элементы корректно отфильтровались.

### Функция *reduce*

Функция *reduce* принимает 2 аргумента: функцию и последовательность. *reduce* последовательно применяет функцию-аргумент к элементам списка, возвращает единичное значение. Обратите внимание: в Python 2.x функция *reduce* доступна как встроенная, в то время как в Python 3 она была перемещена в модуль *functools*.

```
from functools import reduce
items = [1,2,3,4,5]
sum_all = reduce(lambda x,y: x + y, items) # 15
```

Вычисление наибольшего элемента в списке при помощи *reduce*:

```
from functools import reduce
items = [1, 24, 17, 14, 9, 32, 2]
all_max = reduce(lambda a,b: a if (a > b) else b, items) # 32
```

### Функция *zip*

Функция *zip* объединяет в кортежи элементы из последовательностей, переданных в качестве аргументов.

```
a = [1, 2, 3]
b = "xyz"
c = (None, True)
res = list(zip(a, b, c)) # [(1, 'x', None), (2, 'y', True)]
```

Обратите внимание, что **zip** прекращает выполнение, как только достигнут конец самого короткого списка.

## Область видимости переменных

Область видимости или *scope* определяет контекст переменной, в рамках которого ее можно использовать. Другими словами, область видимости определяет, когда и где вы можете использовать свои переменные, функции, и т.д. Если вы попытаетесь использовать что-либо, что не является в вашей области видимости, вы получите ошибку **NameError**.

Python содержит три разных типа области видимости:

1. Локальная область видимости [*local*]
2. Глобальная область видимости [*global*]
3. Нелокальная область видимости [*enclosing*] (*была добавлена в Python 3*)

### Локальная область видимости

Локальная переменная определяется внутри функции и доступна только из этой функции, то есть имеет локальную область видимости:

```
def say_hi():
    name = "Sam"
    surname = "Johnson"
    print("Hello", name, surname)

def say_bye():
    name = "Tom"
    print("Good bye", name)

say_hi()
say_bye()
```

В данном случае в каждой из двух функций определяется локальная переменная **name**. И хотя эти переменные называются одинаково, но тем не менее это две разные переменные, каждая из которых доступна только в рамках своей функции. Также в функции **say\_hi** определена переменная **surname**, которая также является локальной, поэтому в функции **say\_bye** мы ее использовать не сможем.

### Глобальная область видимости

Глобальный контекст подразумевает, что переменная является глобальной, она определена вне любой из функций и доступна любой функции в программе. Например:



```

name = "Tom"

def say_hi():
    print("Hello", name)

def say_bye():
    print("Good bye", name)

say_hi()
say_bye()

```

Здесь переменная *name* является глобальной и имеет глобальную область видимости. И обе определенные здесь функции могут свободно ее использовать.

Есть еще один вариант определения переменной, когда локальная переменная скрывают глобальную с тем же именем:

```

name = "Tom"

def say_hi():
    print("Hello", name)

def say_bye():
    name = "Bob"
    print("Good bye", name)

say_hi() # Hello Tom
say_bye() # Good bye Bob

```

Здесь определена глобальная переменная *name*. Однако в функции *say\_bye* определена локальная переменная с тем же именем *name*. И если функция *say\_hi* использует глобальную переменную, то функция *say\_bye* использует локальную переменную, которая скрывает глобальную.

Если же мы хотим изменить в локальной функции глобальную переменную, а не определить локальную, то необходимо использовать ключевое слово *global*:

```

def say_bye():
    global name
    name = "Bob"
    print("Good bye", name)

```

## Область *enclosing*

В Python 3 было добавлено новое ключевое слово под названием *nonlocal*. С его помощью мы можем добавлять переопределение области во внутреннюю область. Вы можете ознакомиться со всей необходимой на данный счет информацией в [PEP 3104](https://www.python.org/dev/peps/pep-3104/). Это наглядно демонстрируется в нескольких примерах. Один из самых простых – это создание функции, которая может увеличиваться:

```
def counter():  
    num = 0  
    def incrementer():  
        num += 1  
        return num  
    return incrementer
```

Если вы попытаетесь запустить этот код, вы получите ошибку *UnboundLocalError*, так как переменная *num* ссылается прежде, чем она будет назначена в самой внутренней функции. Добавим *nonlocal*:

```
def counter():  
    num = 0  
    def incrementer():  
        nonlocal num  
        num += 1  
        return num  
    return incrementer
```

Результат работы:

```
c = counter()  
  
c() # 1  
c() # 2  
c() # 3
```

*nonlocal* указывает на то, что эта переменная не является локальной, следовательно, ее значение будет взято из ближайшей области видимости, в которой существует переменная с таким же именем.

Суть данной области видимости в том, что внутри функции могут быть вложенные функции и локальные переменные, так вот локальная переменная функции для ее вложенной функции находится в *enclosing* области видимости.

Тип такой функции (*counter*) называется замыкание.

Замыкание (*closure*) – это функция, в теле которой присутствуют ссылки на переменные, объявленные вне тела этой функции в окружающем коде и не являющиеся ее параметрами.

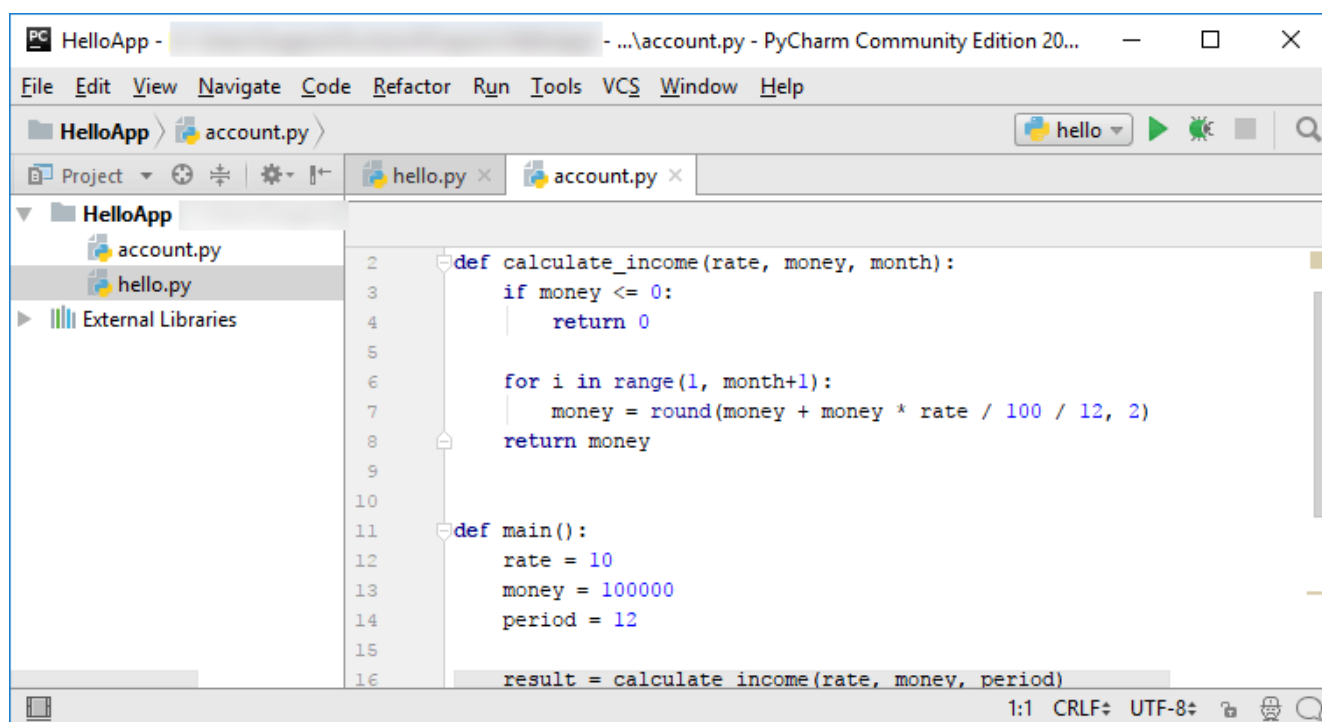
## Модули

Модуль в языке Python представляет отдельный файл с кодом, который можно повторно использовать в других программах.

Для создания модуля необходимо создать собственно файл с расширением *\*.py*, который будет представлять модуль. Название файла будет представлять название модуля. Затем в этом файле надо определить одну или несколько функций.

Пусть основной файл программы будет называться *hello.py*. И мы хотим подключить к нему внешние модули.

Для этого сначала определим новый модуль: создадим новый файл, который назовем *account.py*, в той же папке, где находится *hello.py*. Если используется *PyCharm* или другая IDE, то оба файла просто помещаются в один проект.



Соответственно модуль будет называться *account*. И определим в нем следующий код:

```
def calculate_income(rate, money, month):
    if money <= 0:
        return 0

    for i in range(1, month+1):
        money = round(money + money * rate / 100 / 12, 2)
    return money
```

Здесь определена функция *calculate\_income*, которая в качестве параметров получает процентную ставку вклада, сумму вклада и период, на который делается вклад, и высчитывает сумму, которая получится в конце данного периода.

В файле *hello.py* используем данный модуль:

```
#!/ Программа Банковский счет
import account

rate = int(input("Введите процентную ставку: "))
money = int(input("Введите сумму: "))
period = int(input("Введите период ведения счета в месяцах: "))

result = account.calculate_income(rate, money, period)
print("Параметры счета:\n", "Сумма: ", money, "\n", "Ставка: ", rate,
"\n",
      "Период: ", period, "\n", "Сумма на счете в конце периода: ",
result)
```

Для использования модуля его надо импортировать с помощью оператора *import*, после которого указывается имя модуля: *import account*.

Чтобы обращаться к функциональности модуля, нам нужно получить его пространство имен. По умолчанию оно будет совпадать с именем модуля, то есть в нашем случае также будет называться *account*.

Получив пространство имен модуля, мы сможем обратиться к его функциям по схеме *пространство\_имен.функция*:

```
account.calculate_income(rate, money, period)
```

И после этого мы можем запустить главный скрипт *hello.py*, и он задействует модуль *account.py*. В частности, консольный вывод мог бы быть следующим:

```
Введите процентную ставку: 10
Введите сумму: 300000
Введите период ведения счета в месяцах: 6
Параметры счета:
Сумма: 300000
Ставка: 10
Период: 6
Сумма на счете в конце периода: 315315.99
```

## Настройка пространства имен

По умолчанию при импорте модуля он доступен через одноименное пространство имен. Однако мы можем переопределить это поведение. Так, ключевое слово **as** позволяет сопоставить модуль с другим пространством имен.

Например:

```
import account as acc

#.....

result = acc.calculate_income(rate, money, period)
```

В данном случае пространство имен будет называться *acc*.

Другой вариант настройки предполагает импорт функциональности модуля в глобальное пространство имен текущего модуля с помощью ключевого слова *from*:

```
from account import calculate_income

#.....

result = calculate_income(rate, money, period)
```

В данном случае мы импортируем из модуля *account* в глобальное пространство имен функцию *calculate\_income*. Поэтому мы сможем ее использовать без указания пространства имен модуля, как если бы она была определена в этом же файле.

Если бы в модуле *account* было бы несколько функций, то могли бы их импортировать в глобальное пространство имен одним выражением:

```
from account import *

#.....

result = calculate_income(rate, money, period)
```

Но стоит отметить, что импорт в глобальное пространство имен чреват коллизиями имен функций. Например, если у нас том же файле определена функция с тем же именем, то при вызове функции мы можем получить ошибку. Поэтому лучше избегать использования импорта в глобальное пространство имен.

## Имя модуля

В примере выше модуль *hello.py*, который является главным, использует модуль *account.py*. При запуске модуля *hello.py* программа выполнит всю необходимую работу.

Однако, если мы запустим отдельно модуль `account.py` сам по себе, то ничего на консоли не увидим. Ведь модуль просто определяет функцию и не выполняет никаких других действий. Но мы можем сделать так, чтобы модуль `account.py` мог использоваться как сам по себе, так и подключаться в другие модули.

При выполнении модуля среда определяет его имя и присваивает его глобальной переменной `__name__` (с обеих сторон два подчеркивания). Если модуль является запускаемым, то его имя равно `__main__` (также по два подчеркивания с каждой стороны). Если модуль используется в другом модуле, то в момент выполнения его имя аналогично названию файла без расширения `.py`. И мы можем это использовать. Так, изменим содержимое файла **`account.py`**:

```
def calculate_income(rate, money, month):
    if money <= 0:
        return 0

    for i in range(1, month+1):
        money = round(money + money * rate / 100 / 12, 2)
    return money

def main():
    rate = 10
    money = 100000
    period = 12

    result = calculate_income(rate, money, period)
    print("Параметры счета:\n", "Сумма: ", money, "\n", "Ставка: ",
rate, "\n",
        "Период: ", period, "\n", "Сумма на счете в конце периода:
", result)

if __name__=="__main__":
    main()
```

Кроме того, для тестирования функции определена главная функция `main`. И мы можем сразу запустить файл `account.py` отдельно от всех и протестировать код.

Следует обратить внимание на вызов функции *`main`*:

Переменная `__name__` указывает на имя модуля. Для главного модуля, который непосредственно запускается, эта переменная всегда будет иметь значение `__main__` вне зависимости от имени файла.

Поэтому, если мы будем запускать скрипт `account.py` отдельно, сам по себе, то Python присвоит переменной `__name__` значение `__main__`, далее в выражении `if` вызовет функцию `main` из этого же файла.

Однако если мы будем запускать другой скрипт, а этот – `account.py` – будем подключать в качестве вспомогательного, для `account.py` переменная `__name__` будет иметь значение `account`. И соответственно метод `main` в файле `account.py` не будет работать.

Данный подход с проверкой имени модуля является более рекомендуемым подходом, чем просто вызов метода `main`.

В файле `hello.py` также можно сделать проверку на то, является ли модуль главным (хотя в принципе это необязательно):

```
#!/ Програма Банковски смет
import account

def main():
    rate = int(input("Введете процентную ставку: "))
    money = int(input("Введете сумму: "))
    period = int(input("Введете период ведения счета в месяцах: "))

    result = account.calculate_income(rate, money, period)
    print("Параметры счета:\n", "Сумма: ", money, "\n", "Ставка: ",
rate, "\n",
        "Период: ", period, "\n", "Сумма на счете в конце периода:
", result)

if __name__ == "__main__":
    main()
```

## Основные встроенные модули

### Модуль `random`

Модуль `random` управляет генерацией случайных чисел. Его основные функции:

- `random()`: генерирует случайное число от 0.0 до 1.0
- `randint()`: возвращает случайное число из определенного диапазона
- `randrange()`: возвращает случайное число из определенного набора чисел
- `shuffle()`: перемешивает список
- `choice()`: возвращает случайный элемент списка

Функция ***random()*** возвращает случайное число с плавающей точкой в промежутке от 0.0 до 1.0. Если же нам необходимо число из большего диапазона, скажем от 0 до 100, то мы можем соответственно умножить результат функции ***random*** на 100.

```
import random

number = random.random() # значение от 0.0 до 1.0
print(number)
number = random.random() * 100 # значение от 0.0 до 100.0
print(number)
```

Функция ***randint(min, max)*** возвращает случайное целое число в промежутке между двумя значениями ***min*** и ***max***.

```
import random

number = random.randint(20, 35) # значение от 20 до 35
print(number)
```

Функция ***randrange()*** возвращает случайное целое число из определенного набора чисел. Она имеет три формы:

- ***randrange(stop)***: в качестве набора чисел, из которых происходит извлечение случайного значения, будет использоваться диапазон от 0 до числа ***stop***
- ***randrange(start, stop)***: набор чисел представляет диапазон от числа ***start*** до числа ***stop***
- ***randrange(start, stop, step)***: набор чисел представляет диапазон от числа ***start*** до числа ***stop***, при этом каждое число в диапазоне отличается от предыдущего на шаг ***step***

```
import random

number = random.randrange(10) # значение от 0 до 10
print(number)
number = random.randrange(2, 10) # значение в диапазоне 2, 3, 4, 5,
6, 7, 8, 9, 10
print(number)
number = random.randrange(2, 10, 2) # значение в диапазоне 2, 4, 6,
8, 10
print(number)
```



Для работы со списками в модуле ***random*** определены две функции: функция ***shuffle()*** перемешивает список случайным образом, а функция ***choice()*** возвращает один случайный элемент из списка:

```
numbers = [1, 2, 3, 4, 5, 6, 7, 8]
random.shuffle(numbers)
print(numbers)
random_number = random.choice(numbers)
print(random_number)
```

## Модуль **math**

Встроенный модуль **math** в Python предоставляет набор функций для выполнения математических, тригонометрических и логарифмических операций. Некоторые из основных функций модуля:

- ***pow(num, power)***: возведение числа *num* в степень *power*
- ***sqrt(num)***: квадратный корень числа *num*
- ***ceil(num)***: округление числа до ближайшего наибольшего целого
- ***floor(num)***: округление числа до ближайшего наименьшего целого
- ***factorial(num)***: факториал числа
- ***degrees(rad)***: перевод из радиан в градусы
- ***radians(grad)***: перевод из градусов в радианы
- ***cos(rad)***: косинус угла в радианах
- ***sin(rad)***: синус угла в радианах
- ***tan(rad)***: тангенс угла в радианах
- ***acos(rad)***: арккосинус угла в радианах
- ***asin(rad)***: арксинус угла в радианах
- ***atan(rad)***: арктангенс угла в радианах
- ***log(n, base)***: логарифм числа *n* по основанию *base*
- ***log10(n)***: десятичный логарифм числа *n*

Пример применения некоторых функций:

```
import math

# возведение числа 2 в степень 3
n1 = math.pow(2, 3)
print(n1) # 8

# ту же самую операцию можно выполнить так
n2 = 2**3
print(n2)

# возведение в квадрат
print(math.sqrt(9)) # 3

# ближайшее наибольшее целое число
print(math.ceil(4.56)) # 5

# ближайшее наименьшее целое число
print(math.floor(4.56)) # 4

# перевод из радиан в градусы
print(math.degrees(3.14159)) # 180

# перевод из градусов в радианы
print(math.radians(180)) # 3.1415.....
# косинус
print(math.cos(math.radians(60))) # 0.5
# синус
print(math.sin(math.radians(90))) # 1.0
# тангенс
print(math.tan(math.radians(0))) # 0.0

print(math.log(8,2)) # 3.0
print(math.log10(100)) # 2.0
```

Также модуль *math* предоставляет ряд встроенных констант, такие как **PI** и **E**:

```
import math
radius = 30
# площадь круга с радиусом 30
area = math.pi * math.pow(radius, 2)
print(area)

# натуральный логарифм числа 10
number = math.log(10, math.e)
print(number)
```

## Модуль *locale*

При форматировании чисел Python по умолчанию использует англосаксонскую систему, при которой разряды целого числа отделяются друг от друга запятыми, а дробная часть от целой отделяется точкой. В континентальной Европе, например, используется другая система, при которой разряды разделяются точкой, а дробная и целая часть – запятой:

```
# англосаксонская система
1,234.567
# европейская система
1.234,567
```

И для решения проблемы форматирования под определенную культуру в Python имеется встроенный модуль *locale*.

Для установки локальной культуры в модуле *locale* определена функция *setlocale()*. Она принимает два параметра:

```
setlocale(category, locale)
```

Первый параметр указывает на категорию, к которой применяется функция - к числам, валютам или и числам, и валютам. В качестве значения для параметра мы можем передавать одну из следующих констант:

- **LC\_ALL**: применяет локализацию ко всем категориям – к форматированию чисел, валют, дат и т.д.
- **LC\_NUMERIC**: применяет локализацию к числам
- **LC\_MONETARY**: применяет локализацию к валютам
- **LC\_TIME**: применяет локализацию к датам и времени
- **LC\_CTYPE**: применяет локализацию при переводе символов в верхний или нижний регистр
- **LC\_COLLATE**: применяет локаль при сравнении строк

Второй параметр функции *setlocale* указывает на локальную культуру, которую надо использовать. На **ОС Windows** можно использовать код страны по **ISO** из двух символов, например, для США – "us", для Германии – "de", для России – "ru". Но на **MacOS** необходимо указывать код языка и код страны, например, для английского в США – "en\_US", для немецкого в Германии – "de\_DE", для русского в России – "ru\_RU". По умолчанию фактически используется культура "en\_US".

Непосредственно для форматирования чисел и валют модуль *locale* предоставляет две функции:

- *currency(num)*: форматирует валюту
- *format(str, num)*: подставляет число *num* вместо *плейсхолдера* в строку *str*

Применяются следующие *плейсхолдеры*:

- d: для целых чисел
- f: для чисел с плавающей точкой
- e: для экспоненциальной записи чисел

Перед каждым *плейсхолдером* ставится знак процента %, например: "%d"

При выводе дробных чисел перед *плейсхолдером* после точки можно указать, сколько знаков в дробной части должно отображаться:

```
%.2f          # два знака в дробной части
```

## Модуль datetime

Основной функционал для работы с датами и временем сосредоточен в модуле *datetime* в виде следующих классов:

- *date*
- *time*
- *datetime*

### Класс date

Для работы с датами воспользуемся классом *date*, который определен в модуле *datetime*. Для создания объекта *date* мы можем использовать конструктор *date*, который последовательно принимает три параметра: год, месяц и день.

```
date(year, month, day)
```

Например, создадим какую-либо дату:

```
import datetime

yesterday = datetime.date(2019, 9, 1)
print(yesterday)      # 2019-09-01
```

Если необходимо получить текущую дату, то можно воспользоваться методом *today()*:

```
from datetime import date

t = date.today()
print(t)           # 2019-09-02
print("{}.{}.{}".format(t.day, t.month, t.year)) # 2.9.2019
```

С помощью свойств *day*, *month*, *year* можно получить соответственно день, месяц и год.

### Класс *time*

За работу с временем отвечает класс *time*. Используя, его конструктор, можно создать объект времени:

```
time([hour] [, min] [, sec] [, microsec])
```

Конструктор последовательно принимает часы, минуты, секунды и микросекунды. Все параметры необязательные, и если мы какой-то параметр не передадим, то соответствующее значение будет инициализироваться нулем.

```
from datetime import time

current_time = time()
print(current_time)    # 00:00:00

current_time = time(16, 25)
print(current_time)    # 16:25:00

current_time = time(16, 25, 45)
print(current_time)    # 16:25:45
```

### Класс *datetime*

Класс *datetime* из одноименного модуля объединяет возможности работы с датой и временем. Для создания объекта *datetime* можно использовать следующий конструктор:

```
datetime(year, month, day [, hour] [, min] [, sec] [, microsec])
```

Первые три параметра, представляющие год, месяц и день, являются обязательными. Остальные необязательные, и если мы не укажем для них значения, то по умолчанию они инициализируются нулем.

```
from datetime import datetime

deadline = datetime(2019, 9, 01)
print(deadline)      # 2019-09-01 00:00:00
```

```
deadline = datetime(2019, 9, 01, 4, 30)
print(deadline)      # 2019-09-01 04:30:00
```

Для получения текущих даты и времени можно вызвать метод *now()*:

```
from datetime import datetime

now = datetime.now()
print(now)      # 2019-09-02 12:50:56.239443

print("{}.{}.{}  {}:{}".format(now.day, now.month, now.year, now.hour,
now.minute))  # 2.9.2019  12:50

print(now.date())
print(now.time())
```

С помощью свойств *day*, *month*, *year*, *hour*, *minute*, *second* можно получить отдельные значения даты и времени. А через методы *date()* и *time()* можно получить отдельно дату и время соответственно.

### Преобразование из строки в дату

Из функциональности класса *datetime* следует отметить метод *strptime()*, который позволяет распарсить строку и преобразовать ее в дату. Этот метод принимает два параметра:

```
strptime(str, format)
```

Первый параметр *str* представляет строковое определение даты и времени, а второй параметр – формат, который определяет, как различные части даты и времени расположены в этой строке.

Для определения формата мы можем использовать следующие коды:

- *%d*: день месяца в виде числа
- *%m*: порядковый номер месяца
- *%y*: год в виде 2-х чисел
- *%Y*: год в виде 4-х чисел
- *%H*: час в 24-х часовом формате
- *%M*: минута
- *%S*: секунда

Применим различные форматы:

```
from datetime import datetime
deadline = datetime.strptime("22/05/2019", "%d/%m/%Y")
print(deadline)      # 2019-05-22 00:00:00

deadline = datetime.strptime("22/05/2019 12:30", "%d/%m/%Y %H:%M")
print(deadline)      # 2019-05-22 12:30:00

deadline = datetime.strptime("05-22-2019 12:30", "%m-%d-%Y %H:%M")
print(deadline)      # 2019-05-22 12:30:00
```

## Задания

### Общее задание

1. Изучите теорию и ответьте на вопросы по теме лабораторной работы.
2. Во всех заданиях необходимо проверять корректность вводимых данных и выводить соответствующие сообщения об ошибках.
3. Напишите функцию, которая выводит первые  $n$  строк [треугольника Паскаля](#).
4. С помощью анонимной функции извлеките из списка числа, делимые на 17.

$\text{№ Варианта} = (\text{№ПК} + 1) \% 2 + 1$
---

### Вариант №1

**Задание №5.** Напишите функцию, которая принимает в качестве аргумента список и возвращает *True*, если все значения внутри данного списка уникальны, иначе возвращает *False*.

**Задание №6.** Напишите функцию, которая принимает произвольное число аргументов (каждый аргумент – список) и возвращает *True* если пересечение списков пустое, иначе возвращает *False*.

## Вариант №2

**Задание №5.** Напишите функцию, которая принимает в качестве аргумента список целых чисел и возвращает *True*, если последовательность является монотонной, иначе возвращает *False*.

**Задание №6.** Напишите функцию, которая принимает произвольное число аргументов (каждый аргумент – список) и возвращает пересечение множеств.

## Дополнительное задание

Напишите функцию, которая возвращает количество корректных комбинаций пар круглых скобок (*правильно открытые и закрытые скобки*). В качестве аргумента функция должна принимать целое число  $N$  – количество пар скобок.