# Python

**Типы данных**

**Операторы**

**Циклы**

**Функции**

# Python

Типы данных

Операторы

Циклы

Функции

# Python

Типы данных

Операторы

Циклы

Функции

# Python

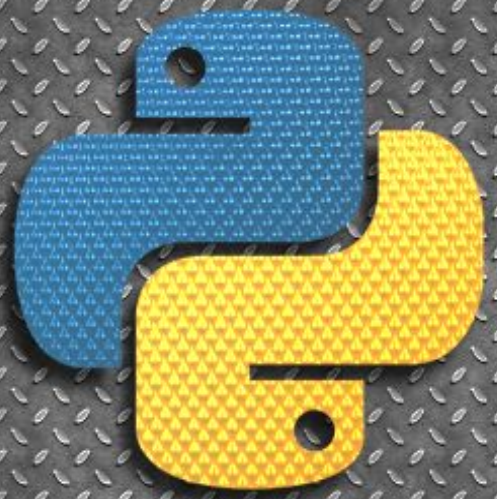
Типы данных

Операторы

Циклы

Функции

# Python

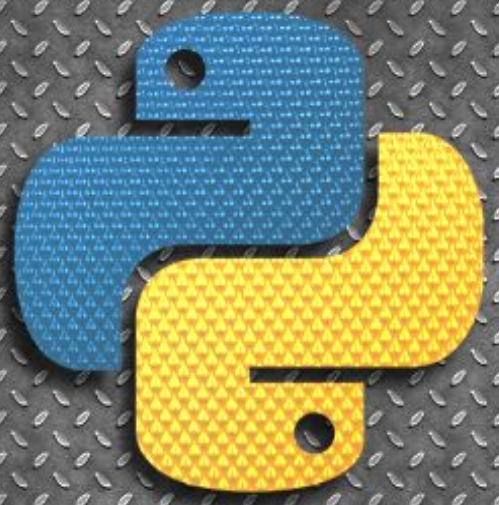


Скриптовый язык программирования. Он универсален, поэтому подходит для решения разнообразных задач и многих платформ, начиная с iOS и Android и заканчивая серверными ОС. Он используется в веб-разработке, создании десктопных и мобильных приложений, программировании игр, а также в аналитике и машинном обучении.

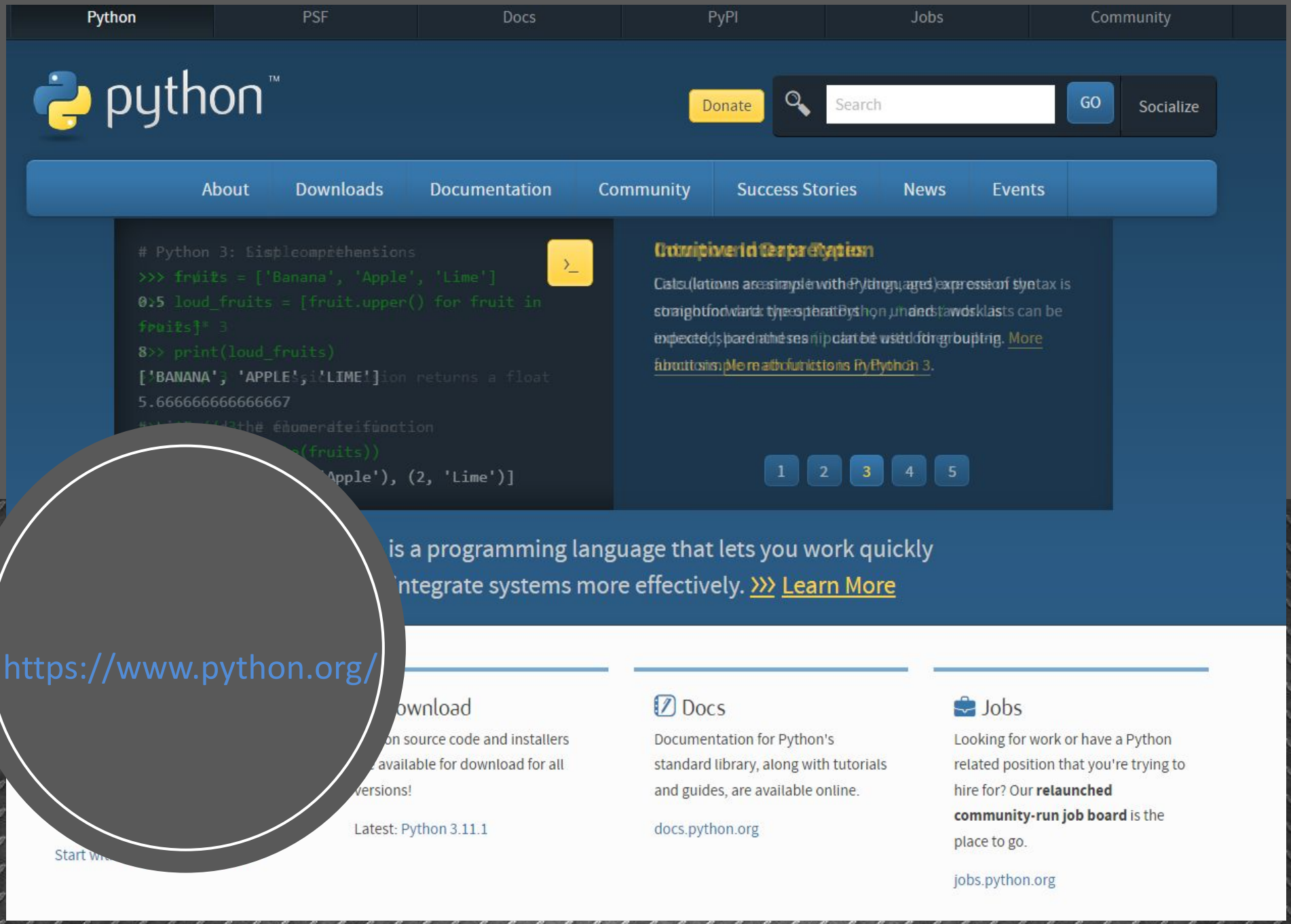
Это интерпретируемый язык — он не компилируется, то есть до запуска представляет из себя обычный текстовый файл. Программировать можно практически на всех платформах, язык хорошо спроектирован и логичен (требуется наличия интерпретатора)

Разработка на нем в разы быстрее, потому что приходится писать меньше кода, чем на Java, C и других языках.

# Python



# Установка интерпретатора

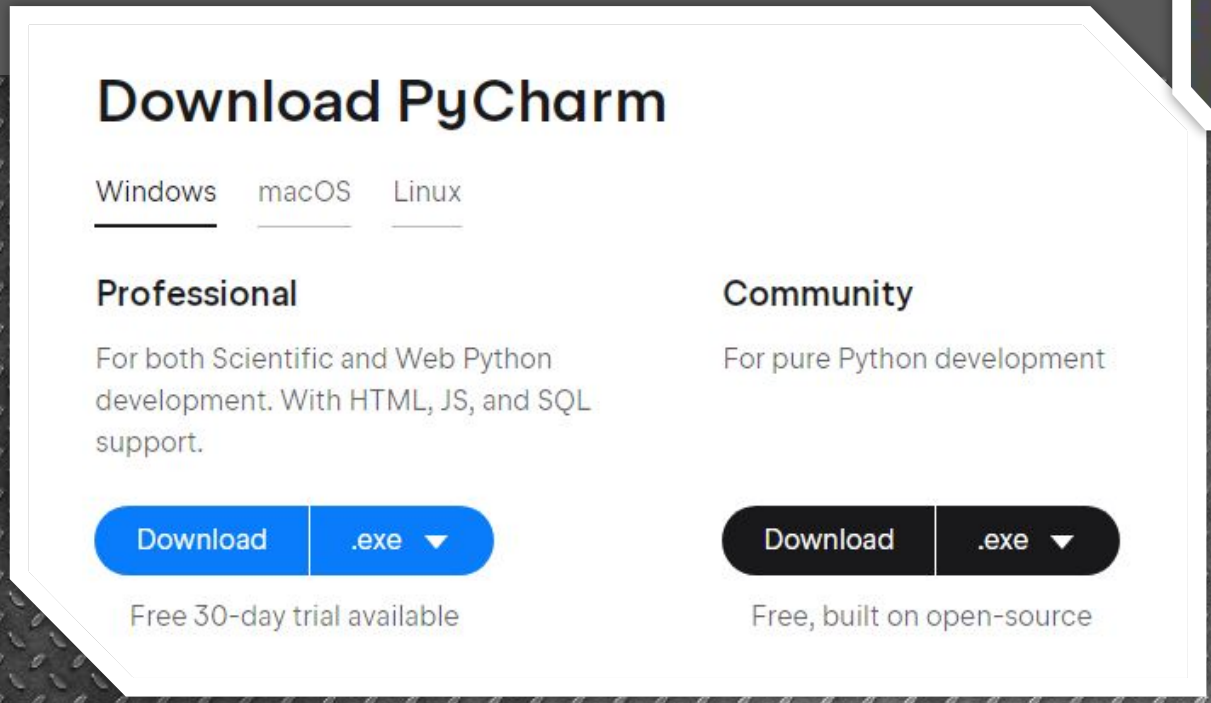



https://www.python.org/

# Установка среды разработки

https://www.jetbrains.com/pycharm/download/#section=windows

PyCharm — популярная среда разработки, заточенная под потребности Python-разработчиков. Она упрощает и ускоряет работу с кодом, помогает избегать багов и писать более чисто.
Community Edition будет достаточно для обучения программированию и небольших личных проектов, написанных на чистом Python.
Professional Edition подойдёт для крупных проектов, научной и веб-разработки.

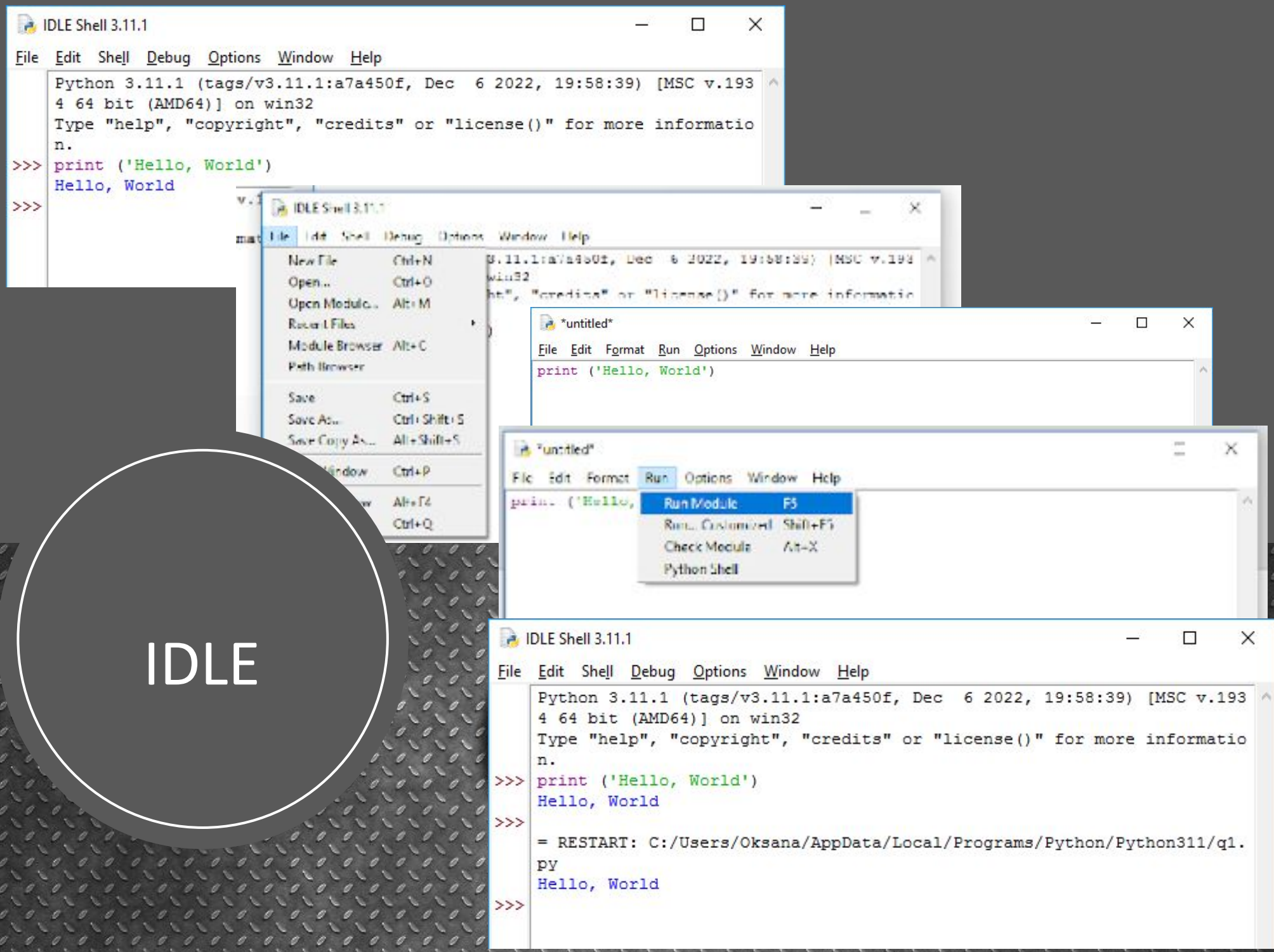



Sublime Text

Thonny
Python IDE for beginners

Visual Studio Code

# IDLE


IDLE Shell 3.11.1
File Edit Shell Debug Options Window Help
Python 3.11.1 (tags/v3.11.1:a7a450f, Dec 6 2022, 19:58:39) [MSC v.193
4 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more informatio
n.
>>> print ('Hello, World')
Hello, World
>>>
New File Ctrl+N
Open... Ctrl+O
Open Module... Alt+M
Recent Files
Module Browser Alt+C
Path Browser
Save Ctrl+S
Save As... Ctrl+Shift+S
Save Copy As... Alt+Shift+S
Ctrl+P
Alt+F4
Ctrl+Q
*untitled*
File Edit Format Run Options Window Help
print ('Hello, World')
Run Module F5
Run... Customized Shift+F5
Check Module Alt+X
Python Shell
IDLE Shell 3.11.1
File Edit Shell Debug Options Window Help
Python 3.11.1 (tags/v3.11.1:a7a450f, Dec 6 2022, 19:58:39) [MSC v.193
4 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more informatio
n.
>>> print ('Hello, World')
Hello, World
>>>
= RESTART: C:/Users/Oksana/AppData/Local/Programs/Python/Python311/q1.
py
Hello, World
>>>

Программы состоят из набора инструкций. Каждая инструкция помещается на новую строку. Например:

```
1 print(2 + 3)
2 print("Hello")
```

Большую роль в python играют отступы. Неправильно поставленный отступ фактически является ошибкой. В этом одно из важных отличий python от других языков программирования, как C# или Java.

```
1 print(2 + 3)
2     print("Hello")
```

# Введение в написание программ

Однако стоит учитывать, что некоторые конструкции языка могут состоять из нескольких строк. Например, условная конструкция if:

```
1 if 1 < 2:
2     print("Hello")
```

# Регистрозависимост

Python - регистрозависимый язык, поэтому выражения print и Print или PRINT представляют разные выражения. Также как и названия переменных Name и name – будут разными

**Строчные** комментарии предваряются знаком решетки #.
Они могут располагаться на отдельной строке:

В **блочных** комментариях до и после текста комментария ставятся три одинарные кавычки: '''текст комментария'''.

## КОММЕНТАРИИ

```
# Вывод на консоль
# сообщения Hello World
print("Hello World")
```

```
'''
    Вывод на консоль
    сообщения Hello World
'''
print("Hello World")
```

Переменная - это именованное местоположение, зарезервированное для хранения значений в памяти. Переменная создается или инициализируется автоматически, когда вы присваиваете ей значение в первый раз.

Каждая переменная должна иметь уникальное имя - идентификатор. Имя допустимого идентификатора должно быть непустой последовательностью символов, должно начинаться с символа подчеркивания (_) или буквы и не может быть ключевым словом python. За первым символом могут следовать подчеркивания, буквы и цифры. Идентификаторы в python чувствительны к регистру.

**Переменные**

| | | | | |
|---|---|---|---|---|
| False | await | else | import | pass |
| None | break | except | in | raise |
| True | class | finally | is | return |
| and | continue | for | lambda | try |
| as | def | from | nonlocal | while |
| assert | del | global | not | with |
| async | elif | if | or | yield |

В python имеется множество встроенных типов данных. Наиболее важные из них:

- **Логический**

- **Числа**: целые, с плавающей точкой, дробные и комплексные

- **Строки** — последовательности символов юникода

- **Списки** — упорядоченные последовательности значений

- **Кортежи** — упорядоченные неизменяемые последовательности значений

- **Множества** — неупорядоченные наборы значений

- **Словари** — неупорядоченные наборы пар вида ключ-значение

**Типы данных**

Есть еще один специальный **литерал**, который используется в python: **None** литерал. Этот литерал является так называемым NoneType объектом, и он используется для представления отсутствия значения

# Логический

Логический тип данных может принимать одно из двух значений: истина или ложь. В python имеются две константы с понятными именами **True** (от англ. True — истина) и **False** (от англ. False — ложь), которые можно использовать для непосредственного присвоения логических значений.
Из-за некоторых обстоятельств, связанных с наследием, оставшимся от python 2, логические значения могут трактоваться как числа. True как 1, и False как 0.

```
>>> size = 1
>>> size < 0
False
>>> size = 0
>>> size < 0
False
>>> size = -1
>>> size < 0
True
```

```
>>> True + True
2
>>> True - False
1
>>> True * False
0
>>> True / False
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: int division or modulo by zero
```

Типы данных

# Строки

Тип str представляет строки. Строка представляет последовательность символов, заключенную в одинарные или двойные кавычки, например "hello" и 'hello'.

```
1 message = "Hello World!"
2 print(message)  # Hello World!
3
4 name = 'Tom'
5 print(name)  # Tom
```

При этом если строка имеет много символов, можем разбить ее на части и разместить их на разных строках кода. В этом случае вся строка заключается в круглые скобки, а ее отдельные части - в кавычки.

```
1 text = ("Laudate omnes gentes laudate "
2         "Magnificat in secula ")
3 print(text)
```

Есть еще один способ создания строк, который называется multi-line строки. Чтобы описать такую «многострочную строку», нужно заключить ее в тройные кавычки — """ или '''

```
4 text = '''Laudate omnes gentes laudate
5 Magnificat in secula
6 Et anima mea laudate
7 Magnificat in secula
8 '''
```

# Строки

Экранированные последовательности – это последовательности символов, определяющие специальные символы которые тяжело ввести с клавиатуры или отобразить на экране. К таким символам можно отнести, например, символ новой строки, символ клавиши BackSpace и прочее.
Строка может содержать ряд специальных символов - управляющих последовательностей.

| Последовательность | Назначение |
|---|---|
| \n | Новая строка |
| \\ | Символ обратного слеша |
| \' | Апостроф ' |
| \" | Кавычка " |
| \a | Звуковой сигнал |
| \b | Забой (символ клавиши BackSpace) |
| \f | Перевод формата |
| \r | Возврат каретки |
| \t | Горизонтальная табуляция |
| \v | Вертикальная табуляция |

"Сырые" строки - подавляют экранирование
Если перед открывающей кавычкой стоит символ 'r' (в любом регистре), то механизм экранирования отключается.

```
1 path = r"C:\python\name.txt"
2 print(path)
```

**Специальные символы**

# Вставка значений в строку

Python позволяет встраивать в строку значения других переменных. Для этого внутри строки переменные размещаются в фигурных скобках {}, а перед всей строкой ставится символ f:

```
1  userName = "Tom"
2  userAge = 37
3  user = f"name: {userName}  age: {userAge}"
4  print(user)   # name: Tom  age: 37
```

В данном случае на место {userName} будет вставляться значение переменной userName. Аналогично на вместо {userAge} будет вставляться значение переменной userAge.

**Форматируемые строки**

Начиная с Python 3.8 после переменной можно поставить символ равенства (=) – и в строку будет подставлено имя этой переменной, знак равенства и значение переменной:

```
>>> b = 12.5
>>> f"Значение переменной {b =}"
'Значение переменной b =12.5'
```

# Функции и методы строк

| Операция | Пример кода |
|---|---|
| Конкатенация (сложение) | >>> S1 = 'spam'<br>>>> S2 = 'eggs'<br>>>> print(S1 + S2)<br>'spameggs' |
| Дублирование строки | >>> print('spam' * 3)<br>spamspamspam |
| Длина строки | >>> len('spam')<br>4 |
| Доступ по индексу | >>> S = 'spam'<br>>>> S[0]<br>'s' |
| Извлечение среза | >>> s = 'spameggs'<br>>>> s[3:5]<br>'me' |

Базовые операции

# Списки

```
>>> a_list = ['a', 'b', 'c', 'd', 'e']          # Объявление списка
>>> a_list[1:3]                                 # Срез со второго по третий символ
['b', 'c']
>>> a_list += ['f']                             # Добавление элемента в список
['a', 'b', 'c', 'd', 'e', 'f']
>>> a_list.append('g')                          # Еще один вариант добавления элементов
['a', 'b', 'c', 'd', 'e', 'f', 'g']
>>> a_list.extend(['h', 'I'])                   # И еще один
['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'I']
>>> a_list.insert(0, 'a')                       # Добавление элемента в указанную позицию
['a', 'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'I']
>>> a_list.count('a')                           # Индексы вхождений элемента в список
0, 1
>>> a_list.index('a')                           #Индекс первого вхождения в список
0
```

```
>>> 'j' in a_list                               # Проверка на вхождение элемента в список
False
>>> len(a_list)                                 # Длинна списка
10
>>> a_list.remove('a')                          # Удаление элемента из списка
['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'I']
>>> a_list.pop([1])                             # Выдергиваем элемент из списка
'b'
>>> a_list.reverse()                            # Отзеркаливание списка
['i', 'h', 'g', 'f', 'e', 'd', 'c', 'b', 'a']
>>> a_list.copy()                               # Копирование списка
['i', 'h', 'g', 'f', 'e', 'd', 'c', 'b', 'a']
>>> a_list.clear()                              # Очищение списка
[]
```

# Типы данных в Python. Кортежи.

Кортеж (tuple) представляет последовательность элементов, которая во многом похожа на список за тем исключением, что кортеж является неизменяемым (immutable) типом. Поэтому мы не можем добавлять или удалять элементы в кортеже, изменять его.

Зачем они нужны:

- Защита от дурака (неизменяемы)
- Меньший размер в памяти
- Можно использовать в качестве ключей словаря
- Присваивание значений нескольким переменным

У кортежей отсутствуют методы.

Для создания кортежа используются круглые скобки, в которые помещаются его значения, разделенные запятыми:

```
tom = ("Tom", 23)
print(tom)      # ("Tom", 23)
```

Также для определения кортежа мы можем просто перечислить значения через запятую без применения скобок:

```
tom = "Tom", 23
print(tom)      # ("Tom", 23)
```

Если кортеж состоит из одного элемента, то после единственного элемента кортежа необходимо поставить запятую:

```
tom = ("Tom",)
```

**Кортежи**

Для создания кортежа из другого набора элементов, например, из списка, можно передать список в функцию tuple(), которая возвратит кортеж:

```
data = ["Tom", 37, "Google"]
tom = tuple(data)
print(tom)      # ("Tom", 37, "Google")
```

С помощью встроенной функции len() можно получить длину кортежа:

```
tom = ("Tom", 37, "Google")
print(len(tom))     # 3
```

Обращение к элементам в кортеже происходит также, как и в списке, по индексу. Индексация начинается также с нуля при получении элементов с начала списка и с -1 при получении элементов с конца списка:

```
tom = ("Tom", 37, "Google", "software developer")
print(tom[0])       # Tom
print(tom[1])       # 37
print(tom[-1])      # software developer
```

Кортежи

При необходимости мы можем разложить кортеж на отдельные переменные:

```
name, age, company, position = ("Tom", 37, "Google", "software developer")
print(name)          # Tom
print(age)           # 37
print(position)      # software developer
print(company)      # Google
```

Как и в списках, можно получить часть кортежа в виде другого кортежа:

```
tom = ("Tom", 37, "Google", "software developer")

# получем подкортеж с 1 по 3 элемента (не включая)
print(tom[1:3])     # (37, "Google")

# получем подкортеж с 0 по 3 элемента (не включая)
print(tom[:3])     # ("Tom", 37, "Google")

# получем подкортеж с 1 по послдений элемент
print(tom[1:])     # (37, "Google", "software developer")
```

Кортежи

# Кортеж как параметр и результат функций

Особенно удобно использовать кортежи, когда необходимо возвратить из функции сразу несколько значений. Когда функция возвращает несколько значений, фактически она возвращает в кортеж:

```
def get_user():
    name = "Tom"
    age = 22
    company = "Google"
    return name, age, company


user = get_user()
print(user)     # ("Tom", 37, "Google")
```

При передаче кортежа в функцию с помощью оператора * его можно разложить на отдельные значения, которые передаются параметрам функции:

```
def print_person(name, age, company):
    print(f"Name: {name}  Age: {age}  Company: {company}")

tom = ("Tom", 22)
print_person(*tom, "Microsoft")     # Name: Tom  Age: 22  Company: Microsoft

bob = ("Bob", 41, "Apple")
print_person(*bob)      # Name: Bob  Age: 41  Company: Apple
```

Для перебора кортежа можно использовать стандартные циклы for и while.

```
1 tom = ("Tom", 22, "Google")
2 for item in tom:
3     print(item)
```

```
1 tom = ("Tom", 22, "Google")
2
3 i = 0
4 while i < len(tom):
5     print(tom[i])
6     i += 1
```

Как для списка с помощью выражения элемент in кортеж можно проверить наличие элемента в кортеже:

```
1 user = ("Tom", 22, "Google")
2 name = "Tom"
3 if name in user:
4     print("Пользователя зовут Tom")
5 else:
6     print("Пользователь имеет другое имя")
```

**Кортежи**

# ТИПЫ ДАННЫХ В PYTHON. МНОЖЕСТВА.

Множество (set) представляют еще один вид набора, который хранит только уникальные элементы. Для определения множества используются фигурные скобки, в которых перечисляются элементы:

```
1 users = {"Tom", "Bob", "Alice", "Tom"}
2 print(users)    # {"Alice", "Bob", "Tom"}
```

Основные способы использования — проверка на вхождение и устранение дублирующихся элементов. Объекты этого типа поддерживают обычные математические операции над множествами, такие как объединение, пересечение, разность и симметрическая разность.

**Множества**

# ТИПЫ ДАННЫХ В PYTHON. МНОЖЕСТВА.

Также для определения множества может применяться функция set(), в которую передается список или кортеж элементов:

```
people = ["Mike", "Bill", "Ted"]
users = set(people)
print(users)    # {"Mike", "Bill", "Ted"}
```

Для создания пустого множество также используется set().

Для добавления одиночного элемента вызывается метод add():

```
users = set()
users.add("Sam")
```

Множества

Для удаления одного элемента вызывается метод remove(), в который передается удаляемый элемент. Но следует учитывать, что если такого элемента не окажется в множестве, то будет сгенерирована ошибка. Поэтому перед удалением следует проверять на наличие элемента с помощью оператора in:

```
users = {"Tom", "Bob", "Alice"}

user = "Tom"
if user in users:
    users.remove(user)
print(users)    # {"Bob", "Alice"}
```

Также для удаления можно использовать метод discard(), который не будет генерировать исключения при отсутствии элемента:

```
users = {"Tom", "Bob", "Alice"}

users.discard("Tim")    # элемент "Tim" отсутствует, и метод ничего не делает
print(users)    #  {"Tom", "Bob", "Alice"}

users.discard("Tom")    # элемент "Tom" есть, и метод удаляет элемент
print(users)    #  {"Bob", "Alice"}
```

Множества

| Методы и операции | Значение |
|---|---|
| A \| B<br>A.union(B) | Возвращает множество, являющееся объединением множеств A и B. |
| A \|= B<br>A.update(B) | Добавляет в множество A все элементы из множества B. |
| A & B<br>A.intersection(B) | Возвращает множество, являющееся пересечением множеств A и B. |
| A &= B<br>A.intersection_update(B) | Оставляет в множестве A только те элементы, которые есть в множестве B. |
| A - B<br>A.difference(B) | Возвращает разность множеств A и B (элементы, входящие в A, но не входящие в B). |
| A -= B<br>A.difference_update(B) | Удаляет из множества A все элементы, входящие в B. |

| Методы и операции | Значение |
|---|---|
| A ^ B<br>A.symmetric_difference(B) | Возвращает симметрическую разность множеств A и B (элементы, входящие в A или в B, но не в оба из них одновременно). |
| A ^= B<br>A.symmetric_difference_update(B) | Записывает в A симметрическую разность множеств A и B. |
| A <= B<br>A.issubset(B) | Возвращает true, если A является подмножеством B. |
| A >= B<br>A.issuperset(B) | Возвращает true, если B является подмножеством A. |
| A < B | Эквивалентно A <= B and A != B |
| A > B | Эквивалентно A >= B and A != B |

Множества

# ТИПЫ ДАННЫХ В PYTHON 3. СЛОВАРИ.

Словарь (dictionary) — это ассоциативный массив или хеш. Это неупорядоченное множество пар ключ: значение с требованием уникальности ключей. Пара фигурных скобок {} создает пустой словарь.

```
dictionary = { ключ1:значение1, ключ2:значение2, ....}
```

В отличие от последовательностей, доступ к элементам словаря производится по ключу, а не по индексу, ключ может быть любого типа, ключ не допускает изменений.

```
>>> users = {1: "Tom", 2: "Bob", 3: "Bill"}
>>> emails = {"tom@gmail.com": "Tom", "bob@gmai.com": "Bob", "sam@gmail.com": "Sam"}
>>> objects = {1: "Tom", "2": True, 3: 100.6}
>>> objects = {}
>>> objects = dict()
```

Словари

| Метод | Значение |
|---|---|
| dict() | создание словаря |
| len() | возвращает число пар |
| clear() | удаляет все значения из словаря |
| copy() | создает псевдокопию словаря |
| deepcopy() | создает полную копию словаря |
| fromkeys() | создание словаря |
| get() | получить значение по ключу |
| has_key() | проверка значения по ключу |
| items() | возвращает список значений |
| iteriyems() | возвращает итератор |
| keys() | возвращает список ключей |
| iterkeys() | возвращает итератор ключей |
| pop() | извлекает значение по ключу |

| Метод | Значение |
|---|---|
| popitem() | извлекает произвольное значение |
| update() | изменяет словарь |
| values() | возвращает список значений |
| itervalues() | возвращает итератор на список значений |
| del() | оператор, удаляет пару по ключу |

Словари

# КОМПЛЕКСНЫЕ СЛОВАРИ

Кроме простейших объектов типа чисел и строк словари также могут хранить и более сложные объекты - те же списки, кортежи или другие словари:

```
users = {
    "Tom": {
        "phone": "+971478745",
        "email": "tom12@gmail.com"
    },
    "Bob": {
        "phone": "+876390444",
        "email": "bob@gmail.com",
        "skype": "bob123"
    }
}
```

В данном случае значение каждого элемента словаря в свою очередь представляет отдельный словарь.

# КОМПЛЕКСНЫЕ СЛОВАРИ

Для обращения к элементам вложенного словаря соответственно необходимо использовать два ключа:

```
old_email = users["Tom"]["email"]
users["Tom"]["email"] = "supertom@gmail.com"
print(users["Tom"])     # { phone": "+971478745", "email": "supertom@gmail.com }
```

Чтобы избежать ошибки при извлечении, необходимо проверять наличие ключа в словаре:

```
key = "skype"
if key in users["Tom"]:
    print(users["Tom"]["skype"])
else:
    print("skype is not found")
```

# ПРЕОБРАЗОВАНИЕ СПИСКОВ И КОРТЕЖЕЙ В СЛОВАРЬ

Несмотря на то, что словарь и список - непохожие по структуре типы, но тем не менее существует возможности для отдельных видов списков преобразования их в словарь с помощью встроенной функции dict(). Для этого список должен хранить набор вложенных списков.

Каждый вложенный список должен состоять из двух элементов - при конвертации в словарь первый элемент станет ключом, а второй - значением:

```
users_list = [
    ["+111123455", "Tom"],
    ["+384767557", "Bob"],
    ["+958758767", "Alice"]
]
users_dict = dict(users_list)
print(users_dict)      # {"+111123455": "Tom", "+384767557": "Bob",
```

# ПРЕОБРАЗОВАНИЕ СПИСКОВ И КОРТЕЖЕЙ В СЛОВАРЬ

Подобным образом можно преобразовать в словарь двухмерные кортежи, которые в свою очередь содержать кортежи из двух элементов:

```
1  users_tuple = (
2      ("+111123455", "Tom"),
3      ("+384767557", "Bob"),
4      ("+958758767", "Alice")
5  )
6  users_dict = dict(users_tuple)
7  print(users_dict)
```

# ОПЕРАЦИИ

1. **Выражение** представляет собой комбинацию значений (или переменных, операторов, вызовы функций), который вычисляет значение, например, 1 + 2.

2. **Операторы** специальные символы или ключевые слова, которые способны работать на ценностях и выполнять (математические) операции, например, (*) оператор умножает два значения: x * y.

3. **Унарный** оператор – это оператор только с одним операндом, например -1, или +3.

4. **Бинарный** оператор – это оператор с двумя операндами, например 4 + 5, или 12 % 5.

3. **Арифметические операторы** в python:

**+** сложение,
**-** вычитание,
***** умножение,
**/** классическое деление - возвращает значение с плавающей точкой, если одно из значений имеет тип с плавающей точкой,
**%** модуль - делит левый операнд на правый операнд и возвращает остаток операции, например, 5 % 2 = 1,
****** возведение в степень - левый операнд, возведенный в степень правого операнда, например 2 ** 3 = 2 * 2 * 2 = 8,
**//** деление, которое возвращает число, полученное в результате деления, но округленное до ближайшего целого числа, например, 3 // 2.0 = 1.0

6. Некоторые операторы действуют раньше других - **иерархия приоритетов** :

- Одинарные +и -имеют самый высокий приоритет
- Далее: **,
- Далее: *, /, и %,
- А затем самый низкий приоритет: бинарный +и -.

7. **Подвыражения в скобках** всегда вычисляются первым, например, 15 - 1 * (5 * (1 + 2)) = 0.

8. **Возведение в степени** оператор использует **правосторонние связывания** , например, 2 ** 2 ** 3 = 256.

9. Побитовые &, |, ^

<<, >>, ~ – сдвиги и инверсия

10. Сравнения <, >, <=, >=, !=, ==

11. Логические and, or, not

12. Двойные сравнения a < b < c

**Операции**

# Практика

# Задача 1:

С начала суток прошло n секунд (n — случайное целое).

Найти количество полных минут, прошедших с начала суток.

1. Импортируем библиотеку random чтобы получить случайные целое число n.

```
import random
```

2. Создаем переменную n, в которую функция randint сгенерирует случайное число (кол-во секунд) в диапазоне от 0 до 86400 :

```
n = random.randint(0,86400)
```

3. Выведем в консоль сгенерированное число n:

```
print("Число секунд: ", n)
```

4. Создаем переменную и сохраняем в нее результат целочисленного деления n (кол-во секунд) на 60:

```
m = n//60
```

5. Выводим в консоль полученное значение:

```
print("Полных минут: ", m)
```

Результат:

```
Число секунд:  65222
Полных минут:  1087
```

**Задача 1**

# PRINT() И INPUT()

Функция **print()** отправляет данные на консоль:

Для вывода строки используются кавычками
"I am a string", или 'I am a string, too‘

Функция **input()** получает данные с консоли .
Может использоваться с входными параметрами и без них. Это позволяет вам написать сообщение перед вводом пользователя, например:

```
Name = input("enter your name: ")
print("hello, " + name + ". nice to meet you!")
```

# УСЛОВНЫЙ ОПЕРАТОР

- одно if, например:

```
x = 10

if x == 10: # condition
    print("x is equal to 10") # executed if the condition is True
```

- серия if, например:

```
x = 10

if x > 5: # condition one
    print("x is greater than 5") # executed if condition one is True

if x < 10: # condition two
    print("x is less than 10") # executed if condition two is True

if x == 10: # condition three
    print("x is equal to 10") # executed if condition three is True
```

# УСЛОВНЫЙ ОПЕРАТОР

- if-else

```
x = 10

if x < 10: # condition
    print("x is less than 10") # executed if the condition is True

else:
    print("x is greater than or equal to 10") # executed if the condition is False
```

```
x = 10

if x > 5: # True
    print("x > 5")

if x > 8: # True
    print("x > 8")

if x > 10: # False
    print("x > 10")

else:
    print("else will be executed")
```

- серия if, за которыми следует else, например:

Каждый if тестируется отдельно. Тело else выполняется , если последний if – False.

# УСЛОВНЫЙ ОПЕРАТОР

- if-elif-else

```
x = 10

if x == 10: # True
    print("x == 10")

if x > 15: # False
    print("x > 15")

elif x > 10: # False
    print("x > 10")

elif x > 5: # True
    print("x > 5")

else:
    print("else will not be executed")
```

# УСЛОВНЫЙ ОПЕРАТОР

- Вложенные условные операторы, например:

```
x = 10


if x > 5: # True
    if x == 6: # False
        print("nested: x == 6")
    elif x == 10: # True
        print("nested: x == 10")
    else:
        print("nested: else")
else:
    print("else")
```

Если условие для if - False, программа проверяет условия последующих elif блоков - первый elif блок, который True выполняется. Если все условия выполнены False, else блок будет выполнен.

# ЦИКЛ WHILE

**while** - один из самых универсальных циклов в python, поэтому довольно медленный. Выполняет тело цикла до тех пор, пока условие истинно.

```
number = 1

while number < 5:
    print(f"number = {number}")
    number += 1
print("Работа программы завершена")
```

```
number = 1
number = 2
number = 3
number = 4
Работа программы завершена
```

# ЦИКЛ WHILE

Для цикла while также можно определить дополнительный блок else, инструкции которого выполняются, когда условие равно False:

```
number = 1

while number < 5:
    print(f"number = {number}")
    number += 1
else:
    print(f"number = {number}. Работа цикла завершена")
print("Работа программы завершена")
```

```
number = 1
number = 2
number = 3
number = 4
number =5. Работа цикла завершена
Работа программы завершена
```

# ЦИКЛ FOR

Цикл **for** немного сложнее и менее универсальный, но выполняется гораздо быстрее цикла **while**.

Этот цикл проходится по любому итерируемому объекту (например строке или списку), и во время каждого прохода выполняет тело цикла.

```
message = "Hello"

for c in message:
    print(c)
```

```
H
e
l
l
o
```

# ЦИКЛ FOR

Цикл **for** также может иметь дополнительный блок **else**, который выполняется после завершения цикла:

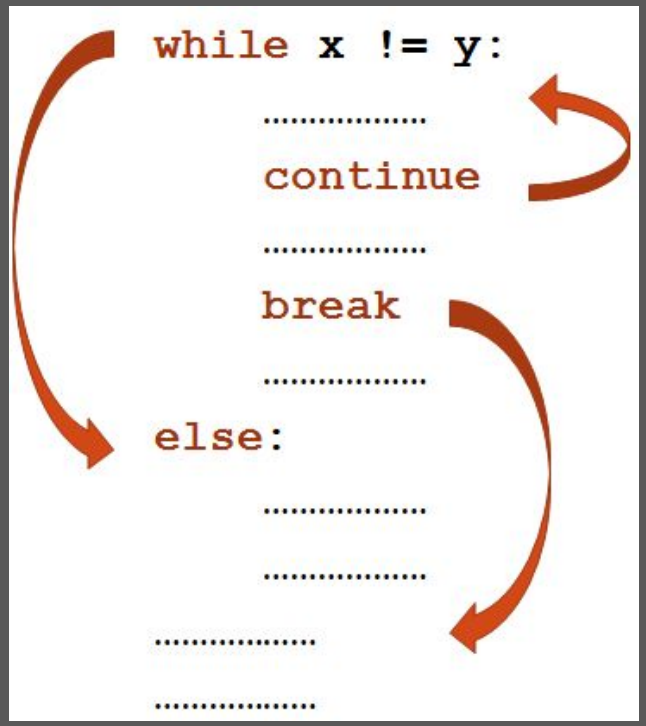
```
message = "Hello"
for c in message:
    print(c)
else:
    print(f"Последний символ: {c}. Цикл завершен");
print("Работа программы завершена")  # инструкция не имеет отступа, поэтому не относится к else
```

```
H
e
l
l
o
Последний символ: o. Цикл завершен
Работа программы завершена
```

# Операторы continue и break

- Оператор continue начинает следующий проход цикла, минуя оставшееся тело цикла (for или while).

- Оператор break досрочно прерывает цикл.



# Операторы continue и break

Оператор **continue** начинает следующий проход цикла, минуя оставшееся тело цикла.

```
for i in 'hello world':
    if i =='o':
        continue
    print(i)
```

```
h
e
l
l

w
r
l
d
```

Оператор **break** досрочно прерывает цикл.

```
for i in 'hello world':
    if i =='o':
        break
    print(i)
```

```
h
e
l
l
```

# Практика

# Задача 2:

Получить от пользователя номер месяца в первом квартале и вывести в консоль его название на английском языке.

Если получили неверный аргумент, то вывести в консоль сообщение об этом.

1. Используем функцию input() для ввода данных, в ней же строкой инструктирует пользователя.

```
number_of_mounth = input("введите номер месяца ")
```

2. Ветвлением обрабатываем полученные данные и выводим название месяца на английском :

```
if number_of_mounth == "1":
    print('January')
elif number_of_mounth == "2":
    print('February')
elif number_of_mounth == "3":
    print('March')
```

3. Прописываем случай, если получили неверный аргумент и выводим в консоль сообщение об этом:

```
else:
    print("Это месяц не в 1 квартале")
```

**Задача 2**

Код целиком:

```
number_of_mounth = input("введите номер месяца ")
if number_of_mounth == "1":
    print('January')
elif number_of_mounth == "2":
    print('February')
elif number_of_mounth == "3":
    print('March')
else:
    print("Это месяц не в 1 квартале")
```

```
введите номер месяца 2
February
```

```
введите номер месяца 6
Это месяц не в 1 квартале
```

Задача 2

# ФУНКЦИИ

**Функции** представляют блок кода, который выполняет определенную задачу и который можно повторно использовать в других частях программы.

```
#define a function
def func1():
    print ("I am learning Python Function")

func1()
#print func1()
#print func1
```

Function definition

Function Call

```
Run  Python10.1
"C:\Users\DK\Desktop\Python code\Python Test\Python 10\Python10
 10/Python10 Code/Python10.1.py"
I am learning Python Function
```

Function output

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

# Параметры функции

```
def print_person(name, age):
    print(f"Name: {name}")
    print(f"Age: {age}")


print_person("Tom", 37)
```

```
Name: Tom
Age: 37
```

- Значение по умолчанию

Если функция имеет несколько параметров, то необязательные параметры должны идти после обязательных.

```
def print_person(name, age = 18):
    print(f"Name: {name}  Age: {age}")


print_person("Bob")
print_person("Tom", 37)
```

# Неопределенное количество параметров

С помощью символа звездочки можно определить параметр, через который можно передавать неопределенное количество значений. Это может быть полезно, когда мы хотим, чтобы функция получала несколько значений, но мы точно не знаем, сколько именно. Например, определим функцию подсчета суммы чисел:

```
def sum(*numbers):
    result = 0
    for n in numbers:
        result += n
    print(f"sum = {result}")


sum(1, 2, 3, 4, 5)      # sum = 15
sum(3, 4, 5, 6)         # sum = 18
```

# Возвращение результата

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

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

```
def print_person(name, age):
    if age > 120 or age < 1:
        print("Invalid age")
        return
    print(f"Name: {name}  Age: {age}")


print_person("Tom", 22)
print_person("Bob", -102)
```

Оператор return не только возвращает значение, но и производит выход из функции.

```
Name: Tom  Age: 22
Invalid age
```

# Локальные функции

Функции могут определяться внутри других функций - внутренние функции или локальные. Они используются только внутри той функции, в которой определены.

```
def print_messages():
    # определение локальных функций
    def say_hello(): print("Hello")
    def say_goodbye(): print("Good Bye")
    # вызов локальных функций
    say_hello()
    say_goodbye()

# Вызов функции print_messages
print_messages()

#say_hello() # вне функции print_messages функция say_hello не доступна
```

Здесь функции say_hello() и say_goodbye() определены внутри функции print_messages() и поэтому по отношению к ней являются локальными. Соответственно они могут использоваться только внутри функции print_messages().

# Декораторы

Декораторы в python представляют функцию, которая в качестве параметра получает функцию и в качестве результата также возвращает функцию. Декораторы позволяют модифицировать выполняемую функцию, значения ее параметров и ее результат без изменения исходного кода этой функции.

```
# определение функции декоратора
def select(input_func):
    def output_func():       # определяем функцию, которая будет выполняться вместо оригинальной
        print("*****************")  # перед выводом оригинальной функции выводим всякую звездочки
        input_func()                # вызов оригинальной функции
        print("*****************")  # после вывода оригинальной функции выводим всякую звездочки
    return output_func      # возвращаем новую функцию

# определение оригинальной функции
@select          # применение декоратора select
def hello():
    print("Hello World")

# вызов оригинальной функции
hello()
```

```
*****************
Hello World
*****************
```

Декораторы

# Получение результата функции

Подобным образом можно получить результат функции и при необходимости изменить его:

```
# определение функции декоратора
def check(input_func):
    def output_func(*args):
        result = input_func(*args)    # передаем функции значения для параметров
        if result < 0: result = 0    # если результат функции меньше нуля, то возвращаем 0
        return result
    return output_func

# определение оригинальной функции
@check
def sum(a, b):
    return a + b

# вызов оригинальной функции
result1 = sum(10, 20)
print(result1)          # 30

result2 = sum(10, -20)
print(result2)          # 0
```

# Практика

# Задача 3:

Получить случайные целые положительные числа n и k.

Используя только операции сложения и вычитания, найти:

- частное от деления нацело n на k
- остаток от деления n на k.

При решении данной задачи использовать цикл While.

1. Импортируем библиотеку random чтобы получить случайные положительные числа n и k.

```
import random
```

2. Создаем переменные n (делимое) и k (делитель), в которые функция randrange генерация случайных чисел в соответствующих диапазонах:

```
n = random.randrange(55,99)
k= random.randrange(1,5)
```

3. Выведем в консоль сгенерированные числа n и k:

```
print('n = ', n)
print('k = ', k)
```

4. Создадим две переменные: r - сохраним значение первого числа (n), q - будет выступать в роли счетчика, сколько раз удалось разделить на 2 число (частное):

```
r = n
q = 0
```

**Задача 3**

5. Используем цикл While с условием пока делимое(r) больше или равно делителю(k) выполнять:
   - надо от делимого отнимать делитель, будет уменьшаться, пока не станет остатком
   - счетчик увеличивать с каждой итерацией на 1

```
while r >= k:
    r -= k
    q += 1
```

6. Вывести в консоль полученные значения, частное и остаток:

```
print("Частное: ", q)
print("Остаток: ", r)
```

Код целиком:

```
import random
n = random.randrange(55,99)
k= random.randrange(1,5)
print('n = ', n)
print('k = ', k)
r = n # сохраняем в новую переменную 1 число
q = 0 # счетчик - сколько раз удалось разделить на 2 число (частное)
while r >= k:#пока 1 число > или = 2
    r -= k#надо от 1 отнимать 2, будет уменьшаться, пока не станет остатком
    q += 1#счетчик +1
print("Частное: ", q)
print("Остаток: ", r)
```

```
n =  83
k =  4
Частное:  20
Остаток:  3
```

# Задача 4:

Описать процедуру Min_max(m, n), записывающую в переменную m минимальное из значений m и n, а в переменную n — максимальное из этих значений (n и m — вещественные параметры, являющиеся одновременно входными и выходными).
Не использовать встроенные функции для нахождения максимума и минимума.

Вызвав четыре раза эту процедуру, найти минимальное и максимальное из данных чисел a, s, d, f.

1. Импортируем библиотеку random.

```
import random
```

2. Создаем процедуру (функцию), которая принимает два параметра m и n:

```
def Min_max(m,n):
```

3. В теле функции создаем пустой список L, в которых будем хранить отсортированные значения :

```
def Min_max(m,n):
    L = []
```

4. При помощи ветвления сортируем значения m и n от меньшего к большему, возвращаем список:

```
if m < n:
    L.append(m)
    L.append(n)
else:
    L.append(n)
    L.append(m)
return L
```

**Задача 4**

4. Перед тем как вызвать функцию создадим четыре переменные, генерируем для них случайные числа в выбранном диапазоне. Выведем значения этих переменных в консоль:

```
a = random.randint(-10,10)
s = random.randint(-10,10)
d = random.randint(-10,10)
f = random.randint(-10,10)
print("a = ",a)
print("s = ",s)
print("d = ",d)
print("f = ",f)
```

5. Сортируем попарно значения первых двух переменных, третий и четв

```
a1,s1 = Min_max(a,s)
d1,f1 = Min_max(d,f)
```

6. Сохраняем 0 элемент списка в min1, 1 элемент списка в max1 и выводим значения максимума и минимума в консоль:

```
Min1,x = Min_max(a1,d1)
x, Max1 = Min_max(s1,f1)
print("Minimum = ",Min1)
print("Maximum = ",Max1)
```

**Задача 4**

Код целиком:

```
import random
def Min_max(m,n):
    L = []              #создание пустого списка
    if m < n:           # отсортировать от меньшего к большему
        L.append(m)
        L.append(n)
    else:
        L.append(n)
        L.append(m)
    return L
a = random.randint(-10,10)
s = random.randint(-10,10)
d = random.randint(-10,10)
f = random.randint(-10,10)
print("a = ",a)
print("s = ",s)
print("d = ",d)
print("f = ",f)
print()

a1,s1 = Min_max(a,s)
d1,f1 = Min_max(d,f)

Min1,x = Min_max(a1,d1) #сохраняем 0 элемент списка в min1
x, Max1 = Min_max(s1,f1)#сохраняем 1 элемент списка в max1
print("Minimum = ",Min1)
print("Maximum = ",Max1)
```

```
a =  2
s =  -9
d =  7
f =  9

Minimum =  -9
Maximum =  9
```

Задача 4