|  |  |  |
| --- | --- | --- |
| **Данные** | **Типы** | **Примеры** |
| Целые числа | int | 73, 0 |
| Числа с плавающей точкой | float | 3.14, -2.79 |
| Строки | str | "Hello, world!" |
| Логические переменные | bool | True, False |
| Списки | list | [1,2,3,4] |
| Кортежи | tuple | (‘a’,’b’,’c’) |
| Словари | dict | {‘a’ : 1, ‘b’ : 2} |
| Множества | set | {‘a’, 1, ‘b’, 2} |

a = **3.14**

b = '3.14'

**print**(type(a))

# <class 'float'>

|  |  |
| --- | --- |
| **Неизменяемые типы** | **Изменяемые типы** |
| Целые числа (**int**) | Списки (**list**) |
| Числа с плавающей точкой (**float**) | Словари (**dict**) |
| Строки (**str**) | Множества (**set**) |
| Логические переменные (**bool**) |  |
| Кортежи (**tuple**) |  |

**print**(type(b))

# <class 'str'>

Все типы можно разделить на две группы: **изменяемые** и **неизменяемые** типы данных.

s = "python"

**print**(s[**0**])

# p

**print**(s[**1**:**4**])

# yth

--------------------------

**print**(**3** > **10**)

# False

**print**(**3** < **10**)

# True

**print**(**3** == **10**) # равны ли объекты?

# False

-----------------------------------

**print**('r' **in** 'world') # проверяем отдельный символ

# True

**print**('th' **in** 'python') # проверяем целую подстроку

# True

**print**('the' **in** 'python')

# False

-------

Для сохранения нескольких объектов (необязательно текстовых) в одну переменную можно использовать**кортежи** (**tuple**). Чтобы создать кортеж, нужно записать данные в круглые скобки через запятую:

date = (**1**, 'january', **2020**)

**print**(date[**0**])

# 1

**print**(date[**1**])

# january

**print**(date[**2**])

# 2020

s1 = "foo"

s2 = "bar"

Допишем вторую строку к первой:

s1 = s1+s2

**print**(s1)

# foobar

s1 = "foo"

**print**(id(s1), s1) #проверяем идентификатор

# 139953609727144, foo

s2 = "bar"

**print**(id(s2), s2) #проверяем идентификатор

# 139953609727088, bar

s1 = s1+s2

**print**(id(s1), s1) #проверяем идентификатор

# 139953459591296, foobar

-----------------

a = **5**/**2**

**print**(a)

# 2.5

|  |  |  |
| --- | --- | --- |
| **Операция** | **Обозначение** | **Пример** |
| Сложение | + | 7+5 = 12 |
| Разность | - | 7-5 = 2 |
| Умножение | \* | 7\*5 = 35 |
| Возведение в степень | \*\* | 7\*\*5 = 16807 |
| Целочисленное деление | // | 7 // 5 = 1 |
| Остаток от деления | % | 7 % 5 = 2 |

-----------

**целочисленное деление** и **взятие остатка от деления.**

Операция целочисленного деления **//** возвращает целую часть получившегося результата, отбрасывая всю дробную часть. Приведем несколько примеров:

**print**(**1** // **3**)

# 0

**print**(**3** // **3**)

# 1

**print**(**29** // **3**)

# 9

**print**(**1** % **3**) # ближайшее число, которое нацело делится на 3 - это ноль

# 1

**print**(**3** % **3**) # в этом примере сам делитель может нацело разделиться

# 0

**print**(**29** % **3**) # здесь ближайшее число - 27, и поэтому результат 29-27=2

# 2

Давайте еще раз посмотрим на все арифметические операции, которые можно применять к целым числам:

|  |  |  |
| --- | --- | --- |
| **Операция** | **Обозначение** | **Пример** |
| Сложение | + | 3.14+1 = 4.14 |
| Вычитание | - | 3.14-1 = 2.14 |
| Умножение | \* | 3.14\*2 = 6.28 |
| Возведение в степень | \*\* | 3.14\*\*2 = 9.8596 |
| Деление | / | 3.14/2 = 1.57 |

**Переменные**

**Изменяемые**

Список (list)

Словарь (dict)

Множество (set)

**Неизменяемые**

Целое число (int)

Число с плавающей запятой (float)

Строка (str)

Логическая переменная (bool)

Кортеж (tuple)

Рассмотрим такой пример:

**print**(**1**/**3**)

# 0.3333333333333333

Казалось бы, всё хорошо, как и ожидалось. В результате такого деления мы получаем десятичную дробь с бесконечным количеством цифр 3 после запятой. Попросим Python вычислить другое выражение:

**print**(**1.3**+**2.3**)

# 3.5999999999999996

person = {} # с помощью фигурных скобок можно создать словарь

# словарь заполняется по принципу - ключ:объект (через двоеточие)

person = {'name' : 'Ivan Petrov'}

# в него можно также добавлять новые объекты по ключу

person['age'] = **25**

person['email'] = 'ivan\_petrov@example.com'

person['phone'] = '8(800)555-35-35'

**print**(person)

# {'name': 'Ivan Petrov', 'age': 25, 'email': 'ivan\_petrov@example.com', 'phone': '8(800)555-35-35'}

Попытка извлечения объекта по несуществующему ключу приведет к ошибке:

**print**(person['address'])

# KeyError: 'address'

--Можно отдельно получить список ключей:

**print**(person.keys())

# dict\_keys(['name', 'age', 'email', 'phone'])

Или список значений:

**print**(person.values())

# dict\_values(['Ivan Petrov', 25, 'ivan\_petrov@example.com', '8(800)555-35-35'])

Из словаря аналогично спискам можно удалить объект по его ключу. Словарь является упорядоченным. В функцию pop() всегда нужно передавать ключ удаляемого объекта:

**print**(person)

# {'name': 'Ivan Petrov', 'age': 25, 'email': 'ivan\_petrov@example.com', 'phone': '8(800)555-35-35'}

person.pop('phone')

**print**(person)

# {'name': 'Ivan Petrov', 'age': 25, 'email': 'ivan\_petrov@example.com'}

title = input("Введите название книги:")

author = input("Введите фамилию автора:")

year = int(input("Введите год издания:"))

book = {'title' : title,

'author' : author,

'year' : year}

**print**(book)

Рассмотрим, как может храниться база данных абитуриентов, поступающих в университет. Информация о каждом абитуриенте может храниться в виде словаря с ключами ФИО, Количество баллов, Заявление (о согласии на зачисление):

abit1 = {"ФИО" : 'Фадеев О.Е.', "Количество баллов" : **283**, "Заявление" : True}

abit2 = {"ФИО" : 'Дружинин И.Я.', "Количество баллов" : **278**, "Заявление" : False}

abit3 = {"ФИО" : 'Афанасьев Д.Н.', "Количество баллов" : **276**, "Заявление" : True}

abits = [abit1, abit2, abit3]

**print**(abits)

# [{'ФИО': 'Фадеев О.Е.', 'Количество баллов': 283, 'Заявление': True}, {'ФИО': 'Дружинин И.Я.', 'Количество баллов': 278, 'Заявление': False}, {'ФИО': 'Афанасьев Д.Н.', 'Количество баллов': 276, 'Заявление': True}]

Этот список, по мере поступления документов, можно пополнять:

abit4 = {"ФИО" : 'Любимчиков А.Я.', "Количество баллов" : **269**, "Заявление" : True}

abits.append(abit4)

**print**(abits)

# [{'ФИО': 'Фадеев О.Е.', 'Количество баллов': 283, 'Заявление': True}, {'ФИО': 'Дружинин И.Я.', 'Количество баллов': 278, 'Заявление': False}, {'ФИО': 'Афанасьев Д.Н.', 'Количество баллов': 276, 'Заявление': True}, {'ФИО': 'Любимчиков А.Я.', 'Количество баллов': 269, 'Заявление': True}]

==

**множества** (set).

**Множество** — это неупорядоченный набор уникальных элементов. Иными словами, во множествах не могут повторяться элементы, а хранятся они в памяти компьютера в произвольном порядке.

Создать множество можно несколькими способами:

a = {'a', 'b', 'c', 'd'} # используя синтаксис { }

L = [**1**,**1**,**2**,**3**,**2**]

b = set(L)

**print**(b)

# {1,2,3}

вернуть обратно в списковое представление, опять же используя явное приведение тип

b\_list = list(b)

**print**(b\_list)

# [1,2,3]

==

Напишите программу, которая на вход принимает текст и выводит количество уникальных символов.

text = input("Введите текст:")

unique = list(set(text))

**print**("Количество уникальных символов: ", len(unique))

Множества в *Python* аналогичны математическим множествам, поэтому для них существует несколько собственных операций.

|  |  |  |
| --- | --- | --- |
| **Операция** | **Название** | **Смысл** |
| set.union(other) | Объединение | Возвращает множество, состоящее из элементов set и other. |
| set.intersection(other) | Пересечение | Возвращает множество, состоящее из элементов, которые встречаются и в set, и в other. |
| set.difference(other) | Разность | Возвращает множество элементов set, которые не встречаются в other. |
| set.symmetric\_difference(other) | Симметричная разность | Возвращает множество, включающее все элементы исходных множеств, которые не принадлежат обоим одновременно. |

Найдите ошибку в коде и перепишите строку с ошибкой полностью. Представленная ниже программа должна находить множество символов, которые встречаются в двух строках одновременно.

a = input("Введите первую строку: ")

b = input("Введите вторую строку: ")

a\_set, b\_set = set(a), set(b) # используем множественное присваивание

a\_and\_b = a\_set.union(b\_set)

**print**(a\_and\_b)

a\_and\_b = a\_set.intersection(b\_set)

alphabet: str = "abcdefqhijklmnopgrstuvwxyz"  
s = "hello"  
result = ""  
steps = 3  
  
for char in s:  
 letter\_number = alphabet.find(char)  
 letter\_number = (letter\_number + steps) % len(alphabet)  
 encoded\_letter = alphabet[letter\_number]  
 result += encoded\_letter  
 # print(letter\_number)  
 print(result)

# alphabet: str = "abcdefqhijklmnopgrstuvwxyz"  
s = 'z'  
result = ""  
steps = 3  
  
for char in s:  
 # letter\_number = alphabet.find(char)  
 letter\_number = ord(char) - ord('a')  
 letter\_number = (letter\_number + steps) % 26  
 encoded\_letter: object = chr(ord('a') + letter\_number)  
 result += encoded\_letter  
 # print(letter\_number)  
 print(result)

def send\_email(email: object, text: object) -> object:  
 *"""* ***:rtype****: object  
 """* if '@' not in email:  
 raise ValueError ("Incorrect Email")  
send\_email("tulinvm@gmail.com", "123 Hellou")

def send\_email(email: object, text: object) -> object:  
 *"""* ***:rtype****: object  
 """* if '@' not in email:  
 raise ValueError("Incorrect Email")  
try:  
 send\_email('tulinvm,gmail.com', "123 Hellou")  
except ValueError as e:  
 print(e)

**Идентичность**

Встроенная функция id() позволяет получить число, которое называется идентичностью. Каждый объект при создании получает некоторое значение идентичности, и оно не может измениться у одного конкретного объекта во время выполнения программы. Также можно сказать, что это число ассоциируется с адресом области памяти компьютера, в которой хранится данный объект.

L = ['a', 'b', 'c']

**print**(id(L))

L.append('d')

**print**(id(L))

4360377856

4360377856

a = 5  
b = 3+2  
print(id(a))  
print(id(b))  
c = id(a)  
d = id(b)  
e = c - d  
  
print(e)

0list\_1 = ['a', 'b', 'c']  
list\_2 = list\_1  
list\_3 = list(list\_1)  
print(list\_1)  
print(list\_2)  
print(list\_3)  
print(list\_1 == list\_2)  
print(list\_1 == list\_3)  
print(list\_1 is list\_2)  
print(list\_1 is list\_3)  
  
M = L.copy()  
  
print(M is L)  
# False

shopping\_center = ("Галерея", "Санкт-Петербург", "Лиговский пр., 30", ["H&M", "Zara"])

shopping\_center[-**1**].append("Uniqlo")

**print**(shopping\_center)

# ('Галерея', 'Санкт-Петербург', 'Лиговский пр., 30', ['H&M', 'Zara', 'Uniqlo'])

shopping\_center = ("Галерея", "Санкт-Петербург", "Лиговский пр., 30", ["H&M", "Zara"])  
list\_id\_before = id(shopping\_center[-1])  
  
shopping\_center[-1].append("Uniqlo")  
list\_id\_after = id(shopping\_center[-1])  
  
print(list\_id\_before == list\_id\_after)

True

## B3.1. Операторы сравнения и логические операторы

По количеству операндов, которые в них участвуют, выделяют следующие типы операторов:

* **унарные** — для их работы требуется одно значение (например, оператор **not** — познакомимся с ним чуть позже);
* **бинарные** — требуют для работы два значения (например, оператор сложения **+**);
* **тернарные** — используют три операнда. Тернарные операторы представлены в *Python* в том числе, но мы не будем разбирать их подробно
* Список операторов сравнения доступных в *Python* представлен в таблице ниже:

|  |  |
| --- | --- |
| **Оператор** | **Значение** |
| **<** | Меньше: *условие верно, если первый операнд меньше второго* |
| **>** | Больше: *условие верно, если первый операнд больше второго* |
| **<=** | Меньше или равно: *условие верно, если первый операнд меньше второго либо равен ему* |
| **>=** | Больше или равно: *условие верно, если первый операнд больше второго либо равен ему* |
| **==** | Равенство: *условие верно, если операнды равны* |
| **!=** | Неравенство: *условие верно, если операнды неравны* |

* **Важно!** Не путайте операцию присваивания значения переменной = с операцией сравнения ==.

**ЛОГИЧЕСКИЕ ОПЕРАТОРЫ**

Основных логических оператора три:

* **not** — логическое «НЕ» (отрицание);
* **and** — логическое «И»;
* **or** — логическое «ИЛИ».

Из них могут составляться более сложные логические операторы.

Также есть два дополнительных оператора: для проверки принадлежности (**in**) и тождественности (**is**).

|  |  |  |
| --- | --- | --- |
| **Оператор** | **Смысл** | **Комментарий** |
| **not** | Логическое «НЕ» | Возвращает противоположное значение. |
| **or** | Логическое «ИЛИ» | Возвращает True, если хотя бы одна из переменных True. |
| **and** | Логическое «И» | Возвращает True, если обе переменные True. |
| **in, not in** | Проверка принадлежности | Возвращает True, если проверяемая переменная содержится (или не содержится) в последовательности: списке, кортеже, строке... |
| **is, is not** | Проверка тождественности | Возвращает True, если проверяемые объекты эквивалентны (или не эквиваленты). То есть переменные ссылаются на один и тот же адрес в памяти компьютера. |

### Логическое «НЕ»

Операция логического «НЕ» по смыслу является отрицанием.

Естественно предположить, что если высказывание не истинно, то оно ложно, и наоборот. Именно так действует этот оператор.

Логические операторы удобно записывать в виде [таблиц истинности](https://ru.wikipedia.org/wiki/%D0%A2%D0%B0%D0%B1%D0%BB%D0%B8%D1%86%D0%B0_%D0%B8%D1%81%D1%82%D0%B8%D0%BD%D0%BD%D0%BE%D1%81%D1%82%D0%B8) — таких таблиц, где перечислены все возможные комбинации значений и результатов, которые может принимать определённая операция.

Как вы уже знаете, логическое «НЕ» является унарным оператором, то есть он требует присутствия рядом только одного значения.

Таблица истинности для оператора **not** выглядит следующим образом:

|  |  |
| --- | --- |
| **x** | **NOT x** |
| True | False |
| False | True |

--

**password = "dgh36gh7"**

**answer = input()**

**#место для написания кода**

**print (answer == password)**

**#Обратите внимание, что мы переводим введенное значение из строки в число с помощью функции int()**

**person\_age = int(input())**

**#Место для написания кода**

**print ((person\_age >= 18) and (person\_age <= 35))**

**ПОБИТОВЫЕ ОПЕРАТОРЫ**

Среди операторов языка Python есть так называемые **побитовые операторы**. Мы не будем останавливаться на них подробно, ограничимся краткими сведениями — для ознакомления.

Побитовыми они называются потому, что работают с данными, представленными в виде набора бит — тех самых единиц и нулей, с которыми работает ваш компьютер.

Таким образом, логические операции «НЕ», «ИЛИ» и «И» применяются над каждым отдельным битом.

Значения некоторых побитовых операторов представлены в таблице ниже.

|  |  |
| --- | --- |
| **Оператор** | **Значение** |
| **~** | Побитовое «НЕ» |
| **|** | Побитовое «ИЛИ» |
| **^** | Побитовое «ИСКЛЮЧИТЕЛЬНОЕ ИЛИ» |
| **&** | Побитовое «И» |
| **Тип данных** | **True** | **False** |
| **int** | Любое целое число отличное от нуля Например: -5, 10 | 0 |
| **float** | Любое число с плавающей запятой, отличное от нуля Например: -0.6, 45.5 | 0.0 |
| **str** | Любая непустая строка Например: "String" | "" |
| **list** | Любой не пустой список Например: **[1, 2, 3]** | [] |
| **dict** | Любой не пустой словарь Например: **{"one": 1, "two": 2}** | {} |
| **NoneType** | — | None |

**if** x > **0** **and** y > **0**:

**print**("Первая четверть")

**if** x < **0** **and** y > **0**:

**print**("Вторая четверть)

**if** x < **0** **and** y < **0**:

**print**("Третья четверть)

**if** x > **0** **and** y < **0**:

**print**("Четвертая четверть)

Функции.

**Метод** — это функция, которая применяется к определенному объекту, используя символ точку:

объект.метод()

Посмотрим как происходит добавление элементов в массив:

# допустим, у нас есть список, содержащий первые 4 буквы латинского алфавита

letters = ['a', 'b', 'c', 'd']

# с помощью метода append() мы добавляем еще один элемент в список

letters.append('e')

**print**(letters)

# ['a', 'b', 'c', 'd', 'e']

Как же получить последний элемент списка, если его точная длина заранее неизвестна? Существует два способа. Первый — использование длины списка. Её можно узнать с помощью встроенной функции len(), которая возвращает длину любого итерируемого объекта. К ним относят строки, списки, кортежи и объекты некоторых других типов данных.

С ними подробнее познакомимся в следующих модулях, а пока что можно получать радость от использования такой функции:

**print**(len(letters))

# 5

Как и ожидалось, длина списка равна 5. Тогда доступ к последнему элементу можно получить, если уменьшить эту длину на 1:

**print**(letters[len(letters)-**1**])

# e

letters.append('f') # добавляем еще одну букву

letters.append('g') # и еще одну

**print**(letters[len(letters)-**1**])

# g

Изменение структуры списка может происходить не только путем ее увеличения (добавления новых объектов), но и удаления уже существующих. Для этого можно использовать метод **pop()**:

**print**(letters)

# ['a', 'b', 'c', 'd', 'e', 'f', 'g']

letters.pop() # вызов метода без аргументов удаляет последний элемент списка

**print**(letters)

# ['a', 'b', 'c', 'd', 'e', 'f']

# был удален последний элемент

letters.pop(**0**) # или можно удалить элемент по его индексу

**print**(letters)

# ['b', 'c', 'd', 'e', 'f']

# был удален нулевой элемент

letters.pop(**3**) # и не обязательно удалять из начала или конца списка

**print**(letters)

# ['b', 'c', 'd', 'f']

# был удален элемент с индексом 3

С помощью **срезов** можно получать сразу несколько элементов списка.

|  |  |  |
| --- | --- | --- |
| **Срез** | **Как работает?** | **Пример** |
| [:] | Возвращает элементы полностью | [‘a’, ‘b’, ‘c’, ‘d’, ‘e’, ‘f’, ‘g’] |
| [2:] | Возвращает элементы списка, начиная с элемента индекса 2 и до конца списка | [‘c’, ‘d’, ‘e’, ‘f’, ‘g’] |
| [:3] | Возвращает элементы списка от его начала до элемента с индексом 3, **не включая** его | [‘a’, ‘b’, ‘c’] |
| [1:4] | Объединяя предыдущие два способа можно получить элементы из середины. В данном случае начиная с индекса 1 до индекса 4, не включительно. Иными словами, элементы с индексами 1,2 и 3 | [‘b’, ‘c’, ‘d’] |
| [::2] | Задает шаг, через который извлекаются элементы | [‘a’, ‘c’, ‘e’, ‘g’] |
| [::-1] | Используя отрицательный шаг, можно развернуть массив | [‘g’, ‘f’, ‘e’, ‘d’, ‘c’, ‘b’, ‘a’] |

L = ["а", "б", "в", **1**, **2**, **3**, **4**]

**print** (L[ ??? ])

# ["б", "в", 1]

Подсказка

  неверно

L = ["а", "б", "в", **1**, **2**, **3**, **4**]

**print** (L[ ??? ])

# ["а", 1, 4]

Подсказка

  верно

L = ["а", "б", "в", **1**, **2**, **3**, **4**]

**print** (L[ ??? ])

# [1, "в", "б", "а"]

Подсказка

  верно

L = ["а", "б", "в", **1**, **2**, **3**, **4**]

**print** (L[ ??? ])

# [4, 3, 2]

Подсказка

  верно

Возможности языка позволяют выполнить определенные действия для каждого элемента списка. Такую операцию можно проделать с помощью функцию **map()**:

map(function, list)

Первый аргумент map() — функция, которую нужно применить к каждому элементу списка, а сам список — второй аргумент. Возвращаемое значение этой функции — объект map, который можно преобразовать, например, обратно в список.

Рассмотрим пример:

# имеем список с числами с плавающей точкой

L = [**3.3**, **4.4**, **5.5**, **6.6**]

# печатаем сам объект map

**print**(map(round, L)) # к каждому элементу применяем функцию округления

# <map object at 0x7fd7e86eb6a0>

# и результат его преобразования в список

**print**(list(map(round, L)))

# [3, 4, 6, 7]

L = ['3.3', '4.4', '5.5', '6.6']

**print** (list (map ( ??? , L)))

  неверно

Однако, пользуясь функциями split() и map() можно выполнить нужное преобразование:

string = input("Введите числа через пробел:")

list\_of\_strings = string.split() # список строковых представлений чисел

list\_of\_numbers = list(map(int, list\_of\_strings)) # cписок чисел

**print**(sum(list\_of\_numbers[::**3**])) # sum() вычисляет сумму элементов списка

*Напишите программу, которая на вход получает последовательность чисел, а выводит модифицированный список:*

* 1. *Первое и последнее числа последовательности должны поменяться местами.*
  2. *В конец списка нужно добавить сумму всех чисел.*

*Посмотреть ответ для самопроверки*

*# все операции - деление строки по пробелам, преобразование к числам*

*# и приведение объекта map к типу список, можно делать в одной строке*

*L = list(map(float, input().split()))*

*# обмениваем первое и последнее число*

*# с помощью множественного присваивания*

*L[****0****], L[-****1****] = L[-****1****], L[****0****]*

*# находим сумму и добавляем ее в конец списка*

*L.append(sum(L))*

***print****(L)*

*Задание 2.5.7*

*1 point possible (graded)*

*Чему будет равен последний элемент списка, полученного в результате работы алгоритма из последней задачи, если на вход подается последовательность чисел:*

***1******1******2******3******5******8******13******21******34******55***

*  нет ответа*

*Отправить*

*В некоторых задачах доступны следующие действия: сохранение, сброс, показ подсказки или ответа. Соответствующие кнопки расположены рядом с кнопкой «Отправить».*

***Словари***

*Использование списков открывает много возможностей, но и имеет свои ограничения. Каждому элементу списка присваивается целочисленный индекс, по которому можно обращаться к нему, модифицировать и даже удалять. С другой стороны, использование целочисленного индекса не является всегда удобным.*

*Например, если мы хотим в одной переменной хранить информацию о человеке (имя, фамилию, электронную почту, номер телефона, почтовый адрес), то нумерация является неудобной и малоинформативной. Более того, в таком случае не принципиален и порядок этих данных — главное, чтобы к ним можно было удобно обратиться.*

*Для этой цели в Python предусмотрены****словари****(dict) — упорядоченные наборы объектов, доступных по ключу. Иными словами, словарь — это совокупность пар ключ-объект.*

*Сам объект, который хранится в словаре, может быть любым. Даже другим словарём. Но на ключи есть важное ограничение: ключ может быть только объектом неизменяемых типов данных, т.е. числом, строкой или кортежем.*

*Как и в случае списков, словарь можно создать пустым, можно сразу наполнить его объектами, а можно расширять постепенно:*

*person = {} # с помощью фигурных скобок можно создать словарь*

*# словарь заполняется по принципу - ключ:объект (через двоеточие)*

*person = {'name' : 'Ivan Petrov'}*

*# в него можно также добавлять новые объекты по ключу*

*person['age'] =* ***25***

*person['email'] = 'ivan\_petrov@example.com'*

*person['phone'] = '8(800)555-35-35'*

***print****(person)*

*# {'name': 'Ivan Petrov', 'age': 25, 'email': 'ivan\_petrov@example.com', 'phone': '8(800)555-35-35'}*

*Попытка извлечения объекта по несуществующему ключу приведет к ошибке:*

***print****(person['address'])*

*# KeyError: 'address'*

*Можно отдельно получить список ключей:*

***print****(person.keys())*

*# dict\_keys(['name', 'age', 'email', 'phone'])*

*Или список значений:*

***print****(person.values())*

*# dict\_values(['Ivan Petrov', 25, 'ivan\_petrov@example.com', '8(800)555-35-35'])*

*Из словаря аналогично спискам можно удалить объект по его ключу. Словарь является упорядоченным. В функцию pop() всегда нужно передавать ключ удаляемого объекта:*

***print****(person)*

*# {'name': 'Ivan Petrov', 'age': 25, 'email': 'ivan\_petrov@example.com', 'phone': '8(800)555-35-35'}*

*person.pop('phone')*

***print****(person)*

*# {'name': 'Ivan Petrov', 'age': 25, 'email': 'ivan\_petrov@example.com'}*

*Задание 2.5.8*

*1 point possible (graded)*

*Какой из перечисленных объектов****не****может быть ключом в словаре?*

*42*

*['id', 42]*

*'id'*

*('id', 42)*

*нет ответа*

*Отправить*

*В некоторых задачах доступны следующие действия: сохранение, сброс, показ подсказки или ответа. Соответствующие кнопки расположены рядом с кнопкой «Отправить».*

*Задание 2.5.9*

*1 point possible (graded)*

*Вместо знаков ??? вставьте название функции, которая удаляет объект из словаря по его ключу.*

*d.??? ('key')*

*  нет ответа*

*Отправить*

*В некоторых задачах доступны следующие действия: сохранение, сброс, показ подсказки или ответа. Соответствующие кнопки расположены рядом с кнопкой «Отправить».*

*Задание 2.5.10*

*1 point possible (graded)*

*Что выведет программа? Впишите получившуюся строку****без использования кавычек****.*

*d = {'day' :* ***22****, 'month' :* ***6****, 'year' :* ***2015****}*

***print****("||".join(d.keys()))*

*  нет ответа*

*Отправить*

*В некоторых задачах доступны следующие действия: сохранение, сброс, показ подсказки или ответа. Соответствующие кнопки расположены рядом с кнопкой «Отправить».*

*Задание 2.5.11 (Внешний источник)*

*Посмотреть ответ для самопроверки*

***Практический пример***

*Использование списков и словарей по-отдельности обеспечивает удобный и эффективный способ хранения данных, а их совместное использование открывает еще больше возможностей.*

*Рассмотрим, как может храниться база данных абитуриентов, поступающих в университет. Информация о каждом абитуриенте может храниться в виде словаря с ключами ФИО, Количество баллов, Заявление (о согласии на зачисление):*

*abit1 = {"ФИО" : 'Фадеев О.Е.', "Количество баллов" :* ***283****, "Заявление" : True}*

*abit2 = {"ФИО" : 'Дружинин И.Я.', "Количество баллов" :* ***278****, "Заявление" : False}*

*abit3 = {"ФИО" : 'Афанасьев Д.Н.', "Количество баллов" :* ***276****, "Заявление" : True}*

*abits = [abit1, abit2, abit3]*

***print****(abits)*

*# [{'ФИО': 'Фадеев О.Е.', 'Количество баллов': 283, 'Заявление': True}, {'ФИО': 'Дружинин И.Я.', 'Количество баллов': 278, 'Заявление': False}, {'ФИО': 'Афанасьев Д.Н.', 'Количество баллов': 276, 'Заявление': True}]*

*Этот список, по мере поступления документов, можно пополнять:*

*abit4 = {"ФИО" : 'Любимчиков А.Я.', "Количество баллов" :* ***269****, "Заявление" : True}*

*abits.append(abit4)*

***print****(abits)*

*# [{'ФИО': 'Фадеев О.Е.', 'Количество баллов': 283, 'Заявление': True}, {'ФИО': 'Дружинин И.Я.', 'Количество баллов': 278, 'Заявление': False}, {'ФИО': 'Афанасьев Д.Н.', 'Количество баллов': 276, 'Заявление': True}, {'ФИО': 'Любимчиков А.Я.', 'Количество баллов': 269, 'Заявление': True}]*

*Использование списка словарей с одним и тем же набором ключей позволяет обрабатывать все эти данные целиком, например, ранжировать по количеству баллов или фильтровать по наличию заявления. В следующих модулях мы научимся выполнять такие операции.*

***Уникальные элементы списка***

*Представим, что у нас есть список номеров и абонентов мобильного оператора, которым принадлежат эти номера. Всегда найдутся люди, на которых зарегистрировано несколько номеров. В таком случае в списке номеров-абонентов данные клиентов будут повторяться. Однако, например, для анализа клиентской базы может понадобиться выделить из общего списка только уникальные данные (скажем, уникальные фамилии).*

*Можно подойти к этой задаче с помощью циклов, пробегая по каждому элементу исходного списка и добавляя в новый только уникальные элементы. Уже звучит так, что для, казалось бы, такой простой операции нужно написать много кода. Не правда ли? Это правда. Руководствуясь философией Python — простое лучше, чем сложное, — хотелось бы иметь более простой способ это сделать. И он есть!*

*Для решения практических задач такого рода в Python есть ещё один изменяемый тип данных, который мы упоминали, но подробно не разбирали —****множества****(set).*

***Множество****— это неупорядоченный набор уникальных элементов. Иными словами, во множествах не могут повторяться элементы, а хранятся они в памяти компьютера в произвольном порядке.*

*Создать множество можно несколькими способами:*

*a = {'a', 'b', 'c', 'd'} # используя синтаксис { }*

*Или, что нам будет более полезно, множество можно создать из списка с помощью приведения типов:*

*L = [****1****,****1****,****2****,****3****,****2****]*

*b = set(L)*

***print****(b)*

*# {1,2,3}*

*В начале мы имели список из 5 элементов, два из которых встречались дважды. «Обернув» исходный список в множество, мы получили только уникальные элементы! И не потребовалось писать много строк кода, чтобы это сделать. Осталось только множество вернуть обратно в списковое представление, опять же используя явное приведение типов.*

*b\_list = list(b)*

***print****(b\_list)*

*# [1,2,3]*

*А для краткости все эти операции можно записать в одну строку, ведь в Python естественным образом заложена лаконичность кода.*

*c = list(set(L))*

***print****(c)*

*# [1,2,3]*

*Задание 2.5.12*

*Задание на самопроверку.*

*Напишите программу, которая на вход принимает текст и выводит количество уникальных символов.*

*Решение*

*2.5.13*

*1 point possible (graded)*

*Используя алгоритм из прошлой задачи, найдите количество уникальных символов в тексте. Скопируйте его к себе в консоль целиком.*

*The Zen of PythonBeautiful is better than ugly.Explicit is better than implicit.Simple is better than complex.Complex is better than complicated.Flat is better than nested.Sparse is better than dense.Readability counts.Special cases aren't special enough to break the rules.Although practicality beats purity.Errors should never pass silently.Unless explicitly silenced.In the face of ambiguity, refuse the temptation to guess.There should be one-- and preferably only one --obvious way to do it.Although that way may not be obvious at first unless you're Dutch.Now is better than never.Although never is often better than \*right\* now.If the implementation is hard to explain, it's a bad idea.If the implementation is easy to explain, it may be a good idea.Namespaces are one honking great idea -- let's do more of those!*

*  нет ответа*

*Loading*

*Отправить*

*В некоторых задачах доступны следующие действия: сохранение, сброс, показ подсказки или ответа. Соответствующие кнопки расположены рядом с кнопкой «Отправить».*

*Множества в Python аналогичны математическим множествам, поэтому для них существует несколько собственных операций.*

|  |  |  |
| --- | --- | --- |
| ***Операция*** | ***Название*** | ***Смысл*** |
| *set.union(other)* | *Объединение* | *Возвращает множество, состоящее из элементов set и other.* |
| *set.intersection(other)* | *Пересечение* | *Возвращает множество, состоящее из элементов, которые встречаются и в set, и в other.* |
| *set.difference(other)* | *Разность* | *Возвращает множество элементов set, которые не встречаются в other.* |
| *set.symmetric\_difference(other)* | *Симметричная разность* | *Возвращает множество, включающее все элементы исходных множеств, которые не принадлежат обоим одновременно.* |

*Чтобы не ходить далеко за примером, вернёмся к базе абонентов мобильного оператора. Пусть у нас есть множество абонентов (для простоты — фамилии) и множество должников, а мы хотим получить множество абонентов, не имеющих долгов.*

*abons = {"Иванов", "Петров", "Васильев", "Антонов"}*

*debtors = {"Петров", "Антонов"}*

*non\_debtors = abons.difference(debtors)*

***print****(non\_debtors)*

*# {'Васильев', 'Иванов'}*

*Задание 2.5.14*

*1 point possible (graded)*

*Найдите ошибку в коде и перепишите строку с ошибкой полностью. Представленная ниже программа должна находить множество символов, которые встречаются в двух строках одновременно.*

*a = input("Введите первую строку: ")*

*b = input("Введите вторую строку: ")*

*a\_set, b\_set = set(a), set(b) # используем множественное присваивание*

*a\_and\_b = a\_set.union(b\_set)*

***print****(a\_and\_b)*

*  нет ответа*

*Отправить*

*В некоторых задачах доступны следующие действия: сохранение, сброс, показ подсказки или ответа. Соответствующие кнопки расположены рядом с кнопкой «Отправить».*

*Задание 2.5.15*

*1 point possible (graded)*

*Напишите программу, которая на вход получает две последовательности целых чисел, а возвращает список элементов, встречающихся только в первой или во второй последовательности, но не в двух одновременно. Какую операцию над множествами вы использовали?*

*Как и в предыдущем задании, нужно подобрать подходящую функцию множества. Выглядеть это будет так: set.something(other), где something и есть выбранная вами функция. Именно её нужно ввести в поле ответа.****Обратите внимание:****необходимо указать только название функции, без скобок.*

*  нет ответа*

*Отправить*

*В некоторых задачах доступны следующие действия: сохранение, сброс, показ подсказки или ответа. Соответствующие кнопки расположены рядом с кнопкой «Отправить».*

***Проверьте себя***

*Напишите числа в порядке возрастания через пробел, которые выведет программа из предыдущего задания, если на вход подаются две последовательности чисел:*

***1******2******3******4******5******6******7******8***

***2******4******6******8******10******12***

*Ответ*

*НазадВперёд*

person = {} # с помощью фигурных скобок можно создать словарь

# словарь заполняется по принципу - ключ:объект (через двоеточие)

person = {'name' : 'Ivan Petrov'}

# в него можно также добавлять новые объекты по ключу

person['age'] = **25**

person['email'] = 'ivan\_petrov@example.com'

person['phone'] = '8(800)555-35-35'

**print**(person)

# {'name': 'Ivan Petrov', 'age': 25, 'email': 'ivan\_petrov@example.com', 'phone': '8(800)555-35-35'}

**print**(person['address'])

# KeyError: 'address'

**print**(person.keys())

# dict\_keys(['name', 'age', 'email', 'phone'])

**print**(person.values())

# dict\_values(['Ivan Petrov', 25, 'ivan\_petrov@example.com', '8(800)555-35-35'])

**print**(person)

# {'name': 'Ivan Petrov', 'age': 25, 'email': 'ivan\_petrov@example.com', 'phone': '8(800)555-35-35'}

person.pop('phone')

**print**(person)

# {'name': 'Ivan Petrov', 'age': 25, 'email': 'ivan\_petrov@example.com'}

Напишите программу, которая получает на вход название книги - title, фамилию автора - author и год выпуска - year.

Полученные данные должны быть преобразованы в словарь book с ключами: title, author, year. Причем year нужно преобраовать в тип int.

title = input("Введите название книги:")  
author = input("Введите фамилию автора:")  
year = int(input("Введите год издания:"))  
  
book = {'title': title,  
 'author': author,  
 'year': year}  
  
print(book)

print("Привет работаем")  
abit1 = {"ФИО" : 'Фадеев О.Е.', "Количество баллов" : 283, "Заявление" : True}  
abit2 = {"ФИО" : 'Дружинин И.Я.', "Количество баллов" : 278, "Заявление" : False}  
abit3 = {"ФИО" : 'Афанасьев Д.Н.', "Количество баллов" : 276, "Заявление" : True}  
  
  
abits = [abit1, abit2, abit3]  
  
print(abits)  
# [{'ФИО': 'Фадеев О.Е.', 'Количество баллов': 283, 'Заявление': True}, {'ФИО': 'Дружинин И.Я.', 'Количество баллов': 278, 'Заявление': False}, {'ФИО': 'Афанасьев Д.Н.', 'Количество баллов': 276, 'Заявление': True}]

Этот список, по мере поступления документов, можно пополнять:

abit4 = {"ФИО" : 'Любимчиков А.Я.', "Количество баллов" : **269**, "Заявление" : True}

abits.append(abit4)

**print**(abits)

# [{'ФИО': 'Фадеев О.Е.', 'Количество баллов': 283, 'Заявление': True}, {'ФИО': 'Дружинин И.Я.', 'Количество баллов': 278, 'Заявление': False}, {'ФИО': 'Афанасьев Д.Н.', 'Количество баллов': 276, 'Заявление': True}, {'ФИО': 'Любимчиков А.Я.', 'Количество баллов': 269, 'Заявление': True}]

**множества** (set).

**Множество** — это неупорядоченный набор уникальных элементов. Иными словами, во множествах не могут повторяться элементы, а хранятся они в памяти компьютера в произвольном порядке.

a = {'a', 'b', 'c', 'd'} # используя синтаксис { }

множество можно создать из списка с помощью приведения типов:

L = [**1**,**1**,**2**,**3**,**2**]

b = set(L)

**print**(b)

# {1,2,3}

Осталось только множество вернуть обратно в списковое представление, опять же используя явное приведение типов.

b\_list = list(b)

**print**(b\_list)

# [1,2,3]

А для краткости все эти операции можно записать в одну строку, ведь в Python естественным образом заложена лаконичность кода.

c = list(set(L))

**print**(c)

# [1,2,3]

Напишите программу, которая на вход принимает текст и выводит количество уникальных символов.

text = input("Введите текст:")

unique = list(set(text))

**print**("Количество уникальных символов: ", len(unique))

text = "The Zen of PythonBeautiful is better than ugly.Explicit is better than implicit.Simple is better than complex.Complex is better than complicated.Flat is better than nested.Sparse is better than dense.Readability counts.Special cases aren't special enough to break the rules.Although practicality beats purity.Errors should never pass silently.Unless explicitly silenced.In the face of ambiguity, refuse the temptation to guess.There should be one-- and preferably only one --obvious way to do it.Although that way may not be obvious at first unless you're Dutch.Now is better than never.Although never is often better than \*right\* now.If the implementation is hard to explain, it's a bad idea.If the implementation is easy to explain, it may be a good idea.Namespaces are one honking great idea -- let's do more of those!"  
#input("Введите текст:")  
print(text)  
unique = list(set(text))  
  
print("Количество уникальных символов: ", len(unique))

Множества в *Python* аналогичны математическим множествам, поэтому для них существует несколько собственных операций.

|  |  |  |
| --- | --- | --- |
| **Операция** | **Название** | **Смысл** |
| set.union(other) | Объединение | Возвращает множество, состоящее из элементов set и other. |
| set.intersection(other) | Пересечение | Возвращает множество, состоящее из элементов, которые встречаются и в set, и в other. |
| set.difference(other) | Разность | Возвращает множество элементов set, которые не встречаются в other. |
| set.symmetric\_difference(other) | Симметричная разность | Возвращает множество, включающее все элементы исходных множеств, которые не принадлежат обоим одновременно. |

Чтобы не ходить далеко за примером, вернёмся к базе абонентов мобильного оператора. Пусть у нас есть множество абонентов (для простоты — фамилии) и множество должников, а мы хотим получить множество абонентов, не имеющих долгов.

abons = {"Иванов", "Петров", "Васильев", "Антонов"}

debtors = {"Петров", "Антонов"}

non\_debtors = abons.difference(debtors)

**print**(non\_debtors)

# {'Васильев', 'Иванов'}

--

Найдите ошибку в коде и перепишите строку с ошибкой полностью. Представленная ниже программа должна находить множество символов, которые встречаются в двух строках одновременно.

a = input("Введите первую строку: ")

b = input("Введите вторую строку: ")

a\_set, b\_set = set(a), set(b) # используем множественное присваивание

a\_and\_b = a\_set.union(b\_set)

**print**(a\_and\_b)

#a = str(123456789)  
#a = list(str(123456789))  
#print(type(a))  
#for e in str(123456789):  
# print(e)  
#r = map(int.list(a))  
print('3' in str(123456789) and '0' in str(123456789))  
print('5' in str(123456789))  
# True

N = **123**

N\_str = str(N) # преобразуем число в строку

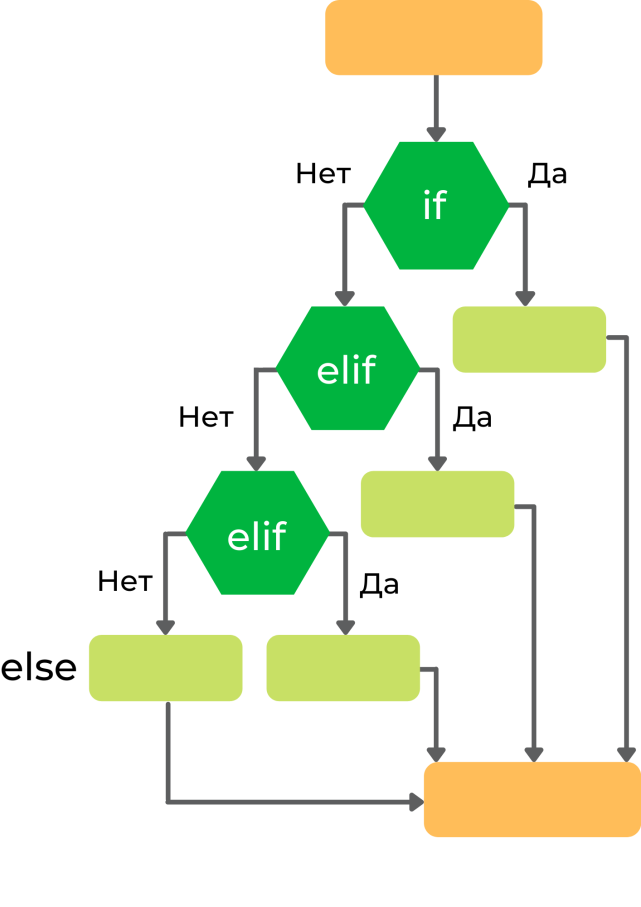
first\_digit = int(N\_str[**0**]) # берём самый левый элемент строки и преобразуем его в число

print(first\_digit % **2** == **0**) # выводим True в случае, если цифра делится на 2, иначе False

# False

К условному оператору **if-else** добавляется условие **elif**, которое можно интерпретировать как «а если…». Таких вложенных **elif** может быть сколько угодно, а блок **else** может быть только один, но по-прежнему является необязательным

Блок-схема оператора **if-elif-else** представлена ниже.



Если условие **if** или какого-либо **elif** по порядку выполняется, то программа сразу переходит в основную ветку программы (нижний оранжевый прямоугольник), а все нижестоящие **elif** пропускаются.

# хорошо

month **in** [**3**, **4**, **5**]

# плохо

month == **3** **or** month == **4** **or** month == **5**

month = int(input())

**if** month **in** [**3**, **4**, **5**]:

print("Весна")

**elif** month **in** [**6**, **7**, **8**]:

print("Лето")

**elif** month **in** [**9**, **10**, **11**]:

print("Осень")

**elif** month **in** [**12**, **1**, **2**]:

print("Зима")

speed = **15**

**if** speed <= **0**:

**print**("error")

**elif** speed >= **1** **and** speed <= **4**:

**print**("weak")

**elif** speed >= **5** **and** speed <=**10**:

**print**("moderate")

**elif** speed >= **11** **and** speed <= **18**:

**print**("strong")

**else**:

**print**("hurricane")

**Bad**

a = **42**

b = **41**

**if** a > b:

result = a

**else**:

result = b

**good**

result = a **if** a > b **else** b

|  |  |  |
| --- | --- | --- |
| **Приоритет** | **Оператор** | **Значение** |
| 1 | **()** | Скобки |
| 2 | **\*\*** | Экспонента |
| 3 | **+x**, **-x**, **~x** | Унарные операции **+**, **-** и **~** |
| 4 | **\***, **/**, **//**, **%** | Умножение, деление, остаток от деления и т. д. |
| 5 | **+**, **-** | Плюс и минус |
| 6 | **==**, **!=**, **>**, **>=**, **<**, **<=**, **is**, **is not**, **in**, **not in** | Операторы сравнения |
| 7 | **not** | Логическое «НЕТ» |
| 8 | **and** | Логическое «И» |
| 9 | **or** | Логическое «ИЛИ» |

Как правильнее всего проверить, что переменная *x* не равна **None**?

x **is** **not** **None**

строка *s* в операторе **if** не пуста

**if** **not** s:

balance = 35000  
login = input("Login")  
password = input("Password")  
account = {'log': 'admin', 'pas': '1234'}  
if login == account['log'] and password == account['pas']:  
 print(f'Hello {login}\nYour balance: {balance}')  
else:  
 print("Permission is denied")

a = input("Input = ")  
c = 10 if a == '100' else 20  
print('c = ', c)

#Запрашиваем ввод температуры

temperature = int(input("Input temperature: "))

#для указания начальных статусов дождливости воспользуемся False или None

rainy = **None**

heavyRain = **None**

#если температура <0 то про дождь спрашивать бессмысленно

**if** temperature > **0**:

rainy = input("Is rainy: ") == "yes"

#если идет дождь спросим насколько он серьезный

**if** rainy:

heavyRain = input("Is heavy rain: ") == "yes"

#реализуем логику по схеме

decision = "Не решил что брать. Останусь дома."

**if** (temperature) > **20** **and** (temperature < **30**) :

**if** rainy:

decision = "Взять футболку шорты и дождевик"

**else**:

decision = "Взять футболку и шорты"

**elif** temperature > **0**:

**if** rainy:

**if** heavyRain:

decision = "Взять пальто, резиновые сапоги и зонт"

**else**:

decision = "Взять пальто и дождевик"

**else**:

decision = "Взять пальто"

**else**:

decision = "Взять пуховик"

#Выведем наше решение на экран

print(decision)

**Мое решение/**

is\_rainy = True # дождь будет  
heavy\_rain = False # не сильный дождь  
temperature = input()  
if temperature >'20' and temperature <'30':  
 if '20' > temperature > '0':  
 if is\_rainy == True:  
 print('дождь будет')  
 else:  
 print('пальто')  
 else:  
 print('пушовик')  
  
else:  
 print(f'холодно')  
 if temperature <='20':  
 print(f'холодно1')  
 else:  
 print('not work')  
# if '20> temperature >'0':  
# else:  
 print('холодно')  
#isRain = input()  
#elif: print(temperature)  
#else:  
  
# is\_rainy = True # дождь будет  
# heavy\_rain = False # не сильный дождь  
  
# if is\_rainy:  
 # в данный блок дописали ещё один условный оператор  
# if heavy\_rain:  
# print("Брать зонт")  
# else:  
 # print("Надеть дождевик")  
 # else:  
 # print("Не брать зонт")  
  
  
#isRain.

S = 0 *# это наша переменная-счетчик, в которой мы будем считать сумму чисел*

n = 1 *# текущее натуральное число, с которого начинаем складывать натуральные числа*

*# заводим цикл while, который будет работать пока сумма не превысит 500*

**while** S < 500: *# делай пока ...*

S += n *# увеличиваем сумму, равносильно S = S + n*

n += 1 *# так как сумма ещё не достигла нужного значения, то увеличиваем переменную счетчик*

print("Ещё считаю ...")

print("Сумма равна: ", S)

print("Количество чисел: ", n-1)

*# хорошо*

n = 1

**while** **True**: *# в данной программе это условие всегда True, цикл будет бесконечным*

print("Hello World")

n += 1

**if** n > 10: *# условие, при достижении которого цикл while будет принудительно завершен*

**break**

# первоначальное значение счетчика  
count = 0  
# начальное значение числа 1  
num = 1  
#пока степень двойки меньше 10000  
while num < 10000:  
 #вычисляем следующую степень двойки  
 num \*= 2  
 #увеличивает счетчик на 1  
 count += 1  
#печатаем результаты  
print(count)  
print(num)

print(year\_count)

**for**

Так вот, функция **range** может работать тремя способами:

* **range(END);**
* **range(START, END);**
* **range(START, END, STEP).**

**RANGE(END)**

В первом случае она принимает одну переменную **END** и возвращает последовательность элементов от 0 до **END-1** шагом в 1.

|  |  |
| --- | --- |
| Пример | Соответствующий диапазон |
| **range(10)** | от 0 до 9 |
| **range(1, 10)** | от 1 до 9 |
| **range(1, 10, 2)** | от 1 до 9 с шагом 2, то есть 1, 3, 5, 7, 9 |

для списка [1, 2, 3] генератор создаст бесконечную последовательность 1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, ... .

**def** **repeat\_list**(list\_):

list\_values = list\_.copy()

**while** True:

value = list\_values.pop(**0**)

list\_values.append(value)

**yield** value

**for** i **in** repeat\_list([**1**, **2**, **3**]):

**print**(i)

**Итератор (iterator)** — это объект, который возвращает свои элементы по одному за раз.

Если вы не знаете, является ли объект итерируемым, можно использовать функцию iter(object) и передать ей в качестве аргумента объект для проверки. Если объект итерируемый, то вам вернется итератор, если нет, то будет соответствующая ошибка.

iter(int) # TypeError: 'type' object is not iterable

iter([**1**, **2**, **3**]) # <list\_iterator at 0x7fb593ca1940>

Системная функция next()позволяет получить следующий элемент от итератора.

Пример *C/C++/Java:*

**for** (int i = **0**; i < **10**; i++):

...

Пример *Python:*

**for** i **in** range(**10**):

**pass**

**Бесконечный счетчик из модуля itertools.**

def count(start, step):  
 counter = start  
 while True:  
 yield counter  
 counter += step  
my\_gen\_func = count(100, 10)  
for i in range(10):  
 print(next(my\_gen\_func))

**Функция высшего порядка**— в программировании это функция, принимающая в качестве аргументов другие функции или возвращающая другую функцию в качестве результата.

**def** **my\_func**(inside\_func):

...

inside\_func() # Вызов функции принятой в качестве аргумента

...

**def** **twice\_func**(inside\_func):

"""Функция, выполняющая дважды функцию принятую в качестве аргумента"""

inside\_func()

inside\_func()

**def** **hello**():

**print**("Hello")

test = twice\_func(hello)

# Hello

# Hello

**Замыкание в программировании**— это функция, в теле которой присутствуют ссылки на переменные, объявленные вне тела этой функции в окружающем коде и не являющиеся её аргументами.

Сделаем функцию, которая будет возвращать функцию, всегда прибавляющую одно и тоже число x:

def make\_adder(x):  
 def adder(n):  
 return x + n # захват переменной "x" из nonlocal области  
 return adder # возвращение функции в качестве результата  
  
# функция, которая будет к любому числу прибавлять пятёрку  
add\_5 = make\_adder(5)  
print(add\_5(10)) # 15  
print(add\_5(100)) # 105

**Декораторы**предназначены для подключения любого дополнительного поведения к основной функции, называемой декорируемой функцией, которое может выполняться до, после или даже вместо основной функции. При этом исходный код декорируемой функции никак не затрагивается.

**def** **my\_decorator**(a\_function\_to\_decorate):

# Здесь мы определяем новую функцию - «обертку». Она нам нужна, чтобы выполнять

# каждый раз при вызове оригинальной функции, а не только один раз

**def** **wrapper**():

# здесь поместим код, которые будет выполняться до вызова, потом вызов

# оригинальной функции, потом код после вызова

**print**("Я буду выполнен до основного вызова!")

result = a\_function\_to\_decorate() # не забываем вернуть значение исходной функции

**print**("Я буду выполнен после основного вызова!")

**return** result

**return** wrapper

**good**

**def** **my\_function**():

**print**("Я - оборачиваемая функция!")

**return** **0**

**print**(my\_function())

# Я - оборачиваемая функция!

# 0

decorated\_function = my\_decorator(my\_function) # декорирование функции

**print**(decorated\_function())

# Я буду выполнен до основного вызова!

# Я - оборачиваемая функция!

# Я буду выполнен после основного вызова!

# 0

авайте попробуем замерить время выполнения системной функции для возведения числа в степень 2 и соответствующего оператора.

**import** **time**

**def** **decorator\_time**(fn):

**def** **wrapper**():

**print**(f"Запустилась функция {fn}")

t0 = time.time()

result = fn()

dt = time.time() - t0

**print**(f"Функция выполнилась. Время: {dt:.10f}")

**return** dt # задекорированная функция будет возвращать время работы

**return** wrapper

**def** **pow\_2**():

**return** **10000000** \*\* **2**

**def** **in\_build\_pow**():

**return** pow(**10000000**, **2**)

pow\_2 = decorator\_time(pow\_2)

in\_build\_pow = decorator\_time(in\_build\_pow)

pow\_2()

# Запустилась функция <function pow\_2 at 0x7f938401b158>

# Функция выполнилась. Время: 0.0000011921

in\_build\_pow()

# Запустилась функция <function in\_build\_pow at 0x7f938401b620>

# Функция выполнилась. Время: 0.0000021458

Чтобы замерить время, будем использовать модуль time, в котором есть функция time(), возвращающая текущее время. Зная время до начала выполнения и сразу после, можно вычислить время работы функции.

Видим, что наша функция работает быстрее, но один раз это не показатель. Нужно выполнить серию запусков и найти среднее время, тогда можно делать какие-то выводы.

озьмите из предыдущего примера декорированные функции, которые возвращают время работы основной функции. Найдите среднее время выполнения для 100 выполнений каждой функции.

Решение

**import** **time**

N = **100**

**def** **decorator\_time**(fn):

**def** **wrapper**():

t0 = time.time()

result = fn()

dt = time.time() - t0

**return** dt

**return** wrapper

**def** **pow\_2**():

**return** **10000000** \*\* **2**

**def** **in\_build\_pow**():

**return** pow(**10000000**, **2**)

pow\_2 = decorator\_time(pow\_2)

in\_build\_pow = decorator\_time(in\_build\_pow)

mean\_pow\_2 = **0**

mean\_in\_build\_pow = **0**

**for** \_ **in** range(N):

mean\_pow\_2 += pow\_2()

mean\_in\_build\_pow += in\_build\_pow()

**print**(f"Функция {pow\_2} выполнялась {N} раз. Среднее время: {mean\_pow\_2 / N:.10f}")

**print**(f"Функция {in\_build\_pow} выполнялась {N} раз. Среднее время: {mean\_in\_build\_pow / N:.10f}")

Вы могли заметить, что время выполнения программы всегда рассчитывается как **0**. Так происходит, потому что иногда программы выполняются настолько быстро, что время не успевает измениться. Поэтому, чтобы увидеть изменения во времени, следует добавить количество вычислений.

## ****«Синтаксический сахар»****

Декораторы настолько часто используемая конструкция в Python, что ее оформили в качестве «синтаксического сахара».

**Синтаксический сахар**в языке программирования — это синтаксические возможности, применение которых не влияет на поведение программы, но делает использование языка более удобным для человека.

Пользоваться им можно так:

**@my\_decorator**

**def** **my\_function**():

**pass**

При этом будет происходить все то же самое, аналогичное.

my\_function = my\_decorator(my\_function)

Имейте в виду, что при использовании синтаксического сахара, на месте декорируемой функции появляется **задекорированная функция**!

**def** **my\_decorator**(fn):

**def** **wrapper**():

fn()

**return** wrapper # возвращается задекорированная функция, которая заменяет исходную

# выведем незадекорированную функцию

**def** **my\_function**():

**pass**

**print**(my\_function) # <function my\_function at 0x7f938401ba60>

# выведем задекорированную функцию

**@my\_decorator**

**def** **my\_function**():

**pass**

**print**(my\_function) # <function my\_decorator.<locals>.wrapper at 0x7f93837059d8>

Видим, что после декорирования, под названием исходной функции будет не сама функция, а функция, которая была внутри декоратора, в данном случае функция wrapper.

## ****Передача аргументов в декорируемую функцию****

До этого мы с вами декорировали только функции без аргументов. Но что будет, если мы попытаемся задекорировать функцию с аргументами?

**def** **do\_it\_twice**(func):

**def** **wrapper**():

func(arg)

func(arg)

**return** wrapper

**@do\_it\_twice**

**def** **say\_word**(word):

**print**(word)

say\_word("Oo!!!")

При выполнении данного кода будет выведена такая ошибка:

**TypeError**: wrapper() takes **0** positional arguments but **1** was given

Связана она с тем, что задекорированная функция wrapper не принимает ни одного аргумента в отличие от исходной. А именно задекорированная функция теперь скрывается за именем переменной say\_word. И каждый следующий вызов say\_word – это вызов wrapper. Следовательно, wrapper должен уметь принимать те же аргументы, что и исходная функция и передавать их в неё. Чтобы не задумываться над количеством аргументов и сделать наш декоратор универсальным, мы будем использовать \*args и \*\*kwargs.

# декоратор, в котором встроенная функция умеет принимать аргументы

**def** **do\_it\_twice**(func):

**def** **wrapper**(\*args, \*\*kwargs):

func(\*args, \*\*kwargs)

func(\*args, \*\*kwargs)

**return** wrapper

**@do\_it\_twice**

**def** **say\_word**(word):

**print**(word)

say\_word("Oo!!!")

# Oo!!!

# Oo!!!

Подведем итог по декораторам:

* + Декораторы добавляют дополнительное поведение функции без изменения её исходного кода
  + Декораторы —  вызовы дополнительных функций, поэтому они немного замедляют ваш код
  + Для передачи аргументов декорируемой функции используйте \*args и \*\*kwargs.

Вот универсальный шаблон для декоратора:

**def** **my\_decorator**(fn):

**print**("Этот код будет выведен один раз в момент декорирования функции")

**def** **wrapper**(\*args, \*\*kwargs):

**print**('Этот код будет выполняться перед каждым вызовом функции')

result = fn(\*args, \*\*kwargs)

**print**('Этот код будет выполняться после каждого вызова функции')

**return** result

**return** wrapper

Напишите декоратор, который будет подсчитывать количество вызовов декорируемой функции. Для хранения переменной содержащей, количество вызовов, используйте nonlocal область декоратора.

**def** **counter**(func):

count = **0**

**def** **wrapper**(\*args, \*\*kwargs):

nonlocal count

func(\*args, \*\*kwargs)

count += **1**

**print**(f"Функция {func} была вызвана {count} раз")

**return** wrapper

**@counter**

**def** **say\_word**(word):

**print**(word)

say\_word("Oo!!!")

# Oo!!!

# Функция <function say\_word at 0x7f93836d47b8> была вызвана 1 раз

say\_word("Oo!!!")

# Oo!!!

# Функция <function say\_word at 0x7f93836d47b8> была вызвана 2 раз

Напишите декоратор, который будет сохранять результаты выполнения декорируемой функции в словаре. Словарь должен находиться в nonlocal области в следующем формате: по ключу располагается аргумент функции, по значению результат работы функции, например, {n: f(n)}.

И при повторном вызове функции будет брать значение из словаря, а не вычислять заново. То есть словарь можно считать промежуточной памятью на время работы программы, где будут храниться ранее вычисленные значения. Исходная функция, которую нужно задекорировать имеет следующий вид и выполняет простое умножение на число 123456789.:

**def** **f**(n):

**return** n \* **123456789**

**def** **cache**(func):

cache\_dict = {}

**def** **wrapper**(num):

nonlocal cache\_dict

**if** num **not** **in** cache\_dict:

cache\_dict[num] = func(num)

**print**(f"Добавление результата в кэш: {cache\_dict[num]}")

**else**:

**print**(f"Возвращение результата из кэша: {cache\_dict[num]}")

**print**(f"Кэш {cache\_dict}")

**return** cache\_dict[num]

**return** wrapper

Напишите программу, которая на вход принимает текст и выводит количество уникальных символов.

text = input("Введите текст:")  
  
unique = set(text)  
  
print("Количество уникальных символов: ", len(unique))

Используя алгоритм из прошлой задачи, найдите количество уникальных символов в предложении.

text = ("The Zen of Python")  
  
unique = set(text)  
  
print("Количество уникальных символов: ", len(unique))

Множества в *Python* аналогичны математическим множествам, поэтому для них существует несколько собственных операций.

|  |  |  |
| --- | --- | --- |
| **Операция** | **Название** | **Смысл** |
| set.union(other) | Объединение | Возвращает множество, состоящее из элементов set и other. |
| set.intersection(other) | Пересечение | Возвращает множество, состоящее из элементов, которые встречаются и в set, и в other. |
| set.difference(other) | Разность | Возвращает множество элементов set, которые не встречаются в other. |
| set.symmetric\_difference(other) | Симметричная разность | Возвращает множество элементов, встречающиеся в одном из множеств, но не в обоих одновременно. |

Пусть у нас есть множество абонентов (для простоты — фамилии) и множество должников, а мы хотим получить множество абонентов, не имеющих долгов.

abons = {"Иванов", "Петров", "Васильев", "Антонов"}

debtors = {"Петров", "Антонов"}

non\_debtors = abons.difference(debtors)

**print**(non\_debtors)

# {'Васильев', 'Иванов'}

1. and: если все операнды являются истинными (ненулевые или непустые), то возвращается последнее истинное значение.

**print**( **1** **and** "hello" **and** [False])

# [False]

Несмотря на то, что последний операнд похож на False, он является непустым списком, а значит он истинный.

1. and: если один из операндов является ложным, то возвращается первый такой операнд.

**print**(**42** **and** **0** **and** '' **and** False)

# 0

1. or: если один из операндов является истинным, то возвращается первый такой операнд, а остальные игнорируются.

**print**([] **or** **3.14** **or** False)

# 3.14

Первый операнд (пустой список) является ложным, следующий  (ненулевое число) — истинным, а значит, возвращается именно он, а все остальные игнорируются (не вычисляются).

1. or: если все операнды являются ложными, то возвращается последний.

**print**(**0** **or** '' **or** False)

# False

**if** a **is** None:

b = **1**

**else**:

b = a

**or**

b = a **if** a **is** **not** None **else** **1**

**or**

b = a **or** **1**

a = "foo"

b = "bar"

**print**(**1** **and** a **or** b) # foo

a = ""

b = "bar"

**print**(**1** **and** a **or** b) # bar

# пусть a и b - переменные, которые мы хотим проверить  
a = 1  
b = None  
  
if a and b: # проверка истинности обеих переменных  
 print("Обе переменные истинные")  
 print(a,b)

a = int(input())

**if** type(a) == int:

**if** **100** <= a <= **999**:

**if** a % **2** == **0**:

**if** a % **3** == **0**:

**print**("Число удовлетворяет условиям")

**or**

**if** type(a) == int **and** **100** <= a <= **999** **and** a % **2** == **0** **and** a % **3** == **0**:

**print**("Число удовлетворяет условиям")

**or**

**if** **all**([type(a) == int,

**100** <= a <= **999**,

a % **2** == **0**,

a % **3** == **0**]):

**print**("Число удовлетворяет условиям")

Функция all([ ]) возвращает *True,* если все элементы списка являются истинными.

А что если нужно, чтобы был хотя бы один истинный? Тогда на помощь приходит функция any([ ])