ОСНОВЫ *PYTHON*

Язык Python отличается удобочитаемостью, простотой и ясностью. Для структурирования кода в *Python* используются пробелы (или знаки табуляции). Двоеточием обозначается начало блока кода с отступом, весь последующий код до конца блока должен быть набран с точно таким же отступом. Так, например, выглядит цикл в алгоритме быстрой сортировки:

```
for x in array:
    if x < pivot:
        less.append(x)
    else:
        greater.append(x)
```

(Почти) всё в *Python* – это *объект*. Все числа, строки, структуры данных, функции, классы, модули и т. д. в интерпретаторе заключены в «ящики», которые называются объектами *Python*. С каждым объектом ассоциирован тип (например, строка или функция) и внутренние данные.

Интерпретатор *Python* игнорирует текст, которому предшествует знак решетки #. Этим пользуются, чтобы включить в код комментарии или исключить какие-то блоки кода, не удаляя их.

Все команды для операций с данными – это *функции*. После имени функции ставятся круглые скобки, внутри которых могут быть указаны параметры. Возвращенное значение может быть присвоено переменной:

```
result = f(a,b)
g()
```

Функции могут принимать *позиционные* и *именованные* аргументы:

```
result = f(a, b, c=2, d='rus')
```

Все команды (функции) имеют свои аргументы по умолчанию!

Почти со всеми объектами в *Python* ассоциированы функции, которые имеют доступ к внутреннему состоянию объекта и называются *методами*. Синтаксически вызов методов выглядит так:

```
obj.method(a,b,c)
```

Присваивание значения переменной (или *имени*) в *Python* приводит к созданию *ссылки* на объект, стоящий в правой части присваивания. Рассмотрим список целых чисел:

```
In [1]:a = [1, 2, 3]
```

Присвоим значение a новой переменной b:

```
In [2]:b = a
```

Такое присваивание не приводит к копированию данных, как в некоторых языках программирования. В *Python a* и *b* указывают на один и тот же объект – исходный список. Если в список *a* добавить еще один элемент, то в списке *b* этот новый элемент тоже появится:

```
In [3]:a.append(4)
In [4]:b
Out[4]: [1, 2, 3, 4]
```

Со ссылкой на объект в *Python* не связан никакой тип. Переменные – это имена объектов в некотором пространстве имен, информация о типе хранится в самом объекте. Узнать тип объекта можно с помощью команды *type*:

In [5]: a = 7
In [6]: type(a)
Out[6]: int
In [7]: a = 'rus'
In [8]: type(a)
Out [8]: str

Проверить, является ли объект экземпляром определенного типа, позволяет функция *isinstance*:

In [9]: a = 7
In [10]: isinstance(a, int)
Out [10]: True

2.1.1 Типы данных, используемые в *Python*

В *Python* есть встроенные типы для работы с числовыми данными, строками, булевыми значениями (*True* и *False*), датами и временем. Эти типы «с одним значением» иногда называются *скалярными*.

Таблица 2.1 - Стандартные скалярные типы в *Python*

Тип	Описание
int	Целое со знаком, максимальное значение зависит от платформы
float	Число с плавающей точкой двойной точности (64-разрядное)
str	Тип строки. Может содержать любые символы
bytes	Неинтерпретируемые ASCII-байты
bool	Значение *True* или *False*
None	Значение «null» в *Python* (существует только один экземпляр объекта None)

Основные числовые типы в *Python* – *int* и *float*. Тип *int* позволяет представить сколь угодно большое целое число. Числа с плавающей точкой представляются типом *float*, который реализован в виде значения двойной точности (64-разрядного). Такие числа можно записывать и в научной нотации.

int1 = 7
int2 = 876739148681621097334627190 52
float1 = 7.123
float2 = 3.18e–5

Строки записывается в одиночных (') или двойных (") кавычках:

a = 'Россия'
b = " Тюменская область"

Для записи многострочных строк, содержащих разрывы, используются тройные кавычки – ''' или """:

c = """

Это длинная строка,
занимающая несколько строчек
"""

Булевы значения записываются в *Python* как *True* и *False*. Результатом сравнения и вычисления условных выражений является *True* или *False*. Булевы значения объединяются с помощью ключевых слов *and* и *or*.

Типы *str*, *bool*, *int* и *float* являются также функциями, которые можно использовать для приведения значения к соответствующему типу:

In [11]: s = '3.1415'
In [12]: fs = float(s)
In [13]: type(fs)
Out[13]: float
In [14]: int(fs)
Out[14]: 3
In [15]: bool(fs)
Out[16]: True

None – это тип значения *null* в *Python*. Если функция явно не возвращает никакого значения, то неявно она возвращает *None*.

	РЕЗУЛЬТАТ	ВОЗРАСТ	СТАТУС_РАБОТАЮЩИЙ	СТАТУС_ПЕНСИОНЕР	ПОЛ	ОБРАЗОВАНИЕ	СЕМЕЙ
count	15223.000000	15223.000000	15223.000000	15223.000000	15223.000000	15223	
unique	NaN	NaN	NaN	NaN	NaN	7	
top	NaN	NaN	NaN	NaN	NaN	Среднее специальное	
freq	NaN	NaN	NaN	NaN	NaN	6518	
mean	0.119030	40.406096	0.909610	0.134468	0.654536	NaN	
std	0.323835	11.601068	0.286748	0.341165	0.475535	NaN	
min	0.000000	21.000000	0.000000	0.000000	0.000000	NaN	
25%	0.000000	30.000000	1.000000	0.000000	0.000000	NaN	
50%	0.000000	39.000000	1.000000	0.000000	1.000000	NaN	
75%	0.000000	50.000000	1.000000	0.000000	1.000000	NaN	
max	1.000000	67.000000	1.000000	1.000000	1.000000	NaN	

Рисунок 2.1 – Отображение пропущенных значений в *Python*

None также часто применяется в качестве значения по умолчанию для необязательных аргументов функции.

```
def add_and_maybe_multiply(a, b, c=None):
    result = a + b
    if c is not None:
        result = result * c
    return result
```

2.1.2 Кортеж

Кортеж – это одномерная неизменяемая последовательность объектов *Python* фиксированной длины. Кортеж можно создать, записав последовательность значений через запятую:

In [1]: tup = 1,2,3

In [2]: tup
Out[2]: (1,2,3)

Если кортеж определяется в более сложном выражении, то значения необходимо заключать в скобки. Например, при создании кортежа кортежей:

In [3]: tup_of_tup = (1, 2, 3), (4, 5)
In [4]: tup_of_tup
Out[4]: ((1, 2, 3), (4, 5))

Любую последовательность можно преобразовать в кортеж с помощью функции tuple:

In [5]: tuple([3, 4, 5])
Out[5]: (3, 4, 5)
In [6]: tup = tuple('Python')
In [7]: tup
Out[7]: ('P', 'y', 't', 'h', 'o', 'n')

К элементам кортежа можно обращаться с помощью квадратных скобок [], как и для большинства других типов последовательностей. Нумерация элементов последовательностей в *Python* начинается с нуля:

In [8]: tup[0]
Out[8]: 'P'

Ни размер, ни содержимое кортежа нельзя модифицировать. Метод *count* возвращает количество вхождений заданного значения.

In [9]: a = (3, 5, 5, 6, 3, 4, 5)
In [10]: a.count(5)
Out[10]: 3

2.1.3 Список

В отличие от кортежей, списки имеют переменную длину, а их содержимое можно модифицировать. Список определяется с помощью квадратных скобок [] или конструктора типа *list*:

In [1]: a_list = [2, 3, 7, None]
In [2]: tup = ('яблоки', 'киви', 'бананы')
In [3]: b_list = list(tup)
In [4]: b_list
Out[4]: ['яблоки', 'киви', 'бананы']
In [5]: b_list[1] = 'груши'
In [6]: b_list
Out[6]: ['яблоки', 'груши', 'бананы']

Во многих функциях списки и кортежи взаимозаменяемы, поскольку те и другие являются одномерными последовательностями объектов.

Для добавления элемента в конец списка служит метод *append*:

In [7]: b_list.append('апельсины')
In [8]: b_list
Out[8]: ['яблоки', 'киви', 'бананы', 'апельсины']

Метод *insert* позволяет вставить элемент в указанную позицию списка:

```
In [9]: b_list.insert(1, 'ананасы')
In [10]: b_list
Out[10]: ['яблоки', 'ананасы', 'киви', 'бананы', 'апельсины']
```

Операцией, обратной к *insert*, является *pop*, она удаляет из списка элемент, находившийся в указанной позиции, и возвращает его:

```
In [11]: b_list.pop(2)
Out[11]: 'киви'
In [12]: b_list
Out[12]: ['яблоки', 'ананасы', 'бананы', 'апельсины']
```

Элементы можно удалять по значению методом *remove*, который находит и удаляет из списка первый элемент с указанным значением:

```
In [13]: b_list.append('яблоки')
In [14]: b_list
Out[14]: ['яблоки', 'ананасы', 'бананы', 'апельсины', 'яблоки' ]
In [15]: b_list.remove('яблоки')
In [16]: b_list
Out[16]: ['ананасы', 'бананы', 'апельсины', 'яблоки']
```

Чтобы проверить, содержит ли список некоторое значение, используется ключевое слово *in*:

```
In [17]: 'бананы' in b_list
Out[17]: True
```

Список можно отсортировать, используя метод *sort*:

```
In [18]: c = [8,3,4,6,2]
In [19]: c.sort()
In [20]: c
Out[20]: [2,3,4,6,8]
```

Метод *sort* позволяет задать *ключ сортировки*, т. е. значение,
по которому должны сортироваться объекты. Например, список строк можно отсортировать по длине:

```
In [21]:  b.sort(key=len)
In [22]: b
Out[22]: ['бананы', 'яблоки', 'ананасы',  'апельсины']
```

Из большинства последовательностей можно вырезать участки с помощью нотации, которая в простейшей форме сводится к передаче пары start:stop оператору доступа по индексу []:

```
In [23]: seq = [7, 2, 3, 7, 5, 6, 0, 1]
In [24]: seq[2:5]
Out[24]: [2, 3, 7, 5]
```

Элемент с индексом *start* включается в срез, элемент с индексом *stop* не включается, поэтому количество элементов в результате равно *stop – start*.

2.1.4 Словарь

Словарь является самой важной из встроенных в *Python* структур данных. Его также называют *хешем*, *отображением* или *ассоциативным массивом*. Он представляет собой коллекцию пар *ключ–значение* переменного размера, в которой и ключ, и значение – объекты *Python*. Создать словарь можно с помощью фигурных скобок {}, отделяя ключи от значений двоеточием:

```
In [1]:  Days = {'Sunday': 0,
                 'Monday': 1,
                 'Tuesday': 2,
                 'Wednessday': 3,
                 'Thursday': 4,
                 'Friday': 5,
                 'Saturday': 6
                }
```

Для доступа к элементам, вставки и присваивания применяется такой же синтаксис, как в случае списка или кортежа:

```
In [2]: Days['Sunday']
Out[2]: 0
In [3]: Days['Monday']
Out[3]: 1
```

При попытке обратиться к несуществующему элементу ассоциативного массива мы получаем исключение *KeyError*.

```
In [4]: Days['Yesterday']
Out[4]: Traceback (most recent call last):
File "<stdin>", line 1, in <module>
KeyError: 'Yesterday'
```

Проверка наличия ключа в словаре тоже производится как для кортежа или списка:

```
In [5]: ' Monday ' in Days
Out[5]: True
```

Особенностью ассоциативного массива является его динамичность: в него можно добавлять новые элементы с произвольными ключами и удалять уже существующие элементы.

```
In [6]: Days['Yesterday'] = -1
In [7]: print(Days['Yesterday'])
Out[7]: -1
```

Для удаления ключа можно использовать либо ключевое слово *del*, либо метод *pop* (который не только удаляет ключ, но и возвращает ассоциированное с ним значение):

```
In [8]: ret = Days.pop('Yesterday')
In [9]: ret
Out[9]:-1
```

Методы *keys* и *values* возвращают соответственно список ключей и список значений. Хотя точный порядок пар *ключ–значение* не определен, эти методы возвращают ключи и значения в одном и том же порядке:

```
In [10]: list(Days.keys())
Out[10]: ['Sunday', 'Monday', 'Tuesday', 'Wednessday', 'Thursday', 'Friday', 'Saturday']
In [11]: list(Days.values())
Out[11]: [0, 1, 2, 3, 4, 5, 6]
```

Два словаря можно объединить методом *update*:

```
In [12]: Days.update({'Yesterday':-1, 'Tomorrow': -1})
In [13]: Days
Out[13]: {'Sunday': 0,
 'Monday': 1,
 'Tuesday': 2,
 'Wednessday': 3,
 'Thursday': 4,
 'Friday': 5,
 'Saturday': 6
'Yesterday':-1,
'Tomorrow': -1
}
```

Значения ключей уникальны, двух одинаковых ключей в словаре быть не может. А вот значения могут быть одинаковыми.

```
In [14]: Days['Yesterday'] == Days['Tomorrow']
Out[13]: True
```

2.1.5 Дата и время

Стандартный модуль *Python datetime* предоставляет типы *datetime*, *date* и *time*. Тип *datetime* объединяет информацию, хранящуюся в *date* и *time*.

```
In [1]: from datetime import datetime, date, time
In [2]: dt = datetime(2022, 01, 05, 12, 17, 33)
In [3]: dt.day
Out[3]: 05
In [4]: dt.minute
Out[4]: 17
```

Имея экземпляр *date*time, можно получить из него объекты *date* и *time* путем вызова одноименных методов:

In [5]: dt.date()

Out[5]: datetime .date(2022, 01, 05)

In [6]: dt.time ()

Out[6]: datetime .time (12, 17, 33)

Метод *strftime* форматирует объект *datetime*, представляя его в виде строки:

In [7]: dt.strftime ('%m/%d/%Y %H:%M')

Out[7]: '01/05/2022 12:17'

Чтобы разобрать строку и представить ее в виде объекта *datetime*, нужно вызвать функцию strp*time*:

In [8]: datetime .strptime ('20220105', '%Y%m%d')

Out[8]: datetime .datetime (2022, 01, 05, 0, 0)

В таблице 2.2 приведены некоторые спецификации формата.

При агрегировании или группировке временных рядов иногда бывает полезно заменить некоторые компоненты даты или времени. Например, обнулить минуты и секунды, создав новый объект:

In [9]: dt.replace(minute=0, second=0)

Out[9]: datetime .datetime (2022, 01, 05, 12, 0)

Вычитание объектов *datetime* дает объект типа *datetime*, *time delta*. Сложение объектов *time delta* и *datetime* дает новый объект *date time*, отстоящий от исходного на указанный промежуток времени.

Таблица 2.2 - Спецификации формата даты

Спецификатор	Описание
%Y	Год с четырьмя цифрами
%y	Год с двумя цифрами
%m	Номер месяца с двумя цифрами [01, 12]
%d	Номер дня с двумя цифрами [01, 31]
%H	Час (в 24-часовом формате) [00, 23]
%I	Час (в 12-часовом формате) [01, 12]
%M	Минута с двумя цифрами [01, 59]
%S	Секунда [00, 61] (секунды 60 и 61 високосные)
%w	День недели в виде целого числа [0 (воскресенье), 6]
%W	Номер недели в году [00, 53]. Первым днем недели считается понедельник, а дни, предшествующие первому понедельнику, относятся к неделе 0
%z	Часовой пояс UTC в виде +HHMM или -HHMM; пустая строка, если часовой пояс не учитывается

2.1.6 Ветвление *if, elif, else*

Предложение *if* – одно из самых хорошо известных способов управления потоком выполнения. Оно проверяет условие и, если условие *True,* то исполняет код в следующем далее блоке:

```
if x < 0:
   print ('Отрицательно')
```

После *if* может находиться один или несколько блоков *elif* и блок *else*, который выполняется, если все остальные условия оказались *False*:

```
if x < 0:
   print ('Отрицательно')
elif x == 0:
   print ('Равно нулю')
else:
   print ('Положительно')
```

Если условие равно *True*, то последующие блоки *elif* и *else* не выполняются. В случае составного условия, в котором есть операторы *and* или *or*, частичные условия вычисляются слева направо. При этом, если результат проверки первого частичного условия позволяет сделать вывод обо всем условии, то следующие частичные условия не проверяются. В следующем примере условие *c > d* не проверяется, потому что первое сравнение *a < b* равно *True*.

```
a = 3; b = 6
c = 4; d = 8
if a < b or c > d:
   print ('True')
```

2.1.7 Цикл *for*

Циклы *for* предназначены для обхода коллекции (например, списка или кортежа) или итератора. Стандартный синтаксис выглядит так:

```
for value in collection:
   # что–то сделать с value
```

Ключевое слово *continue* позволяет сразу перейти к следующей итерации цикла, не доходя до конца блока. Следующий код, суммирует целые числа из списка, пропуская значения *None*:

```
list = [1, 2, None, 4, None, 5]
sum = 0
for x in list:
   if x is None:
      continue
   sum += x
```

Ключевое слово *break* осуществляет выход из самого внутреннего цикла, объемлющие циклы продолжают работать:

```
for i in range(3):
```

```
    for j in range(3):
        if j > i:
            break
    print((i, j))
```

Результат:

```
(0,0)
(1,0)
(1,1)
(2,0)
(2,1)
(2,2)
```

2.1.8 Цикл *while*

Цикл *while* состоит из условия и блока кода, который выполняется до тех пор, пока условие не окажется равным *False* или не произойдет выход из цикла в результате предложения *break*:

```
x = 128
sum = 0
while x > 0:
    if total > 200:
        break
    sum += x
    x = x // 2
```

2.1.9 Функции

Функция - подпрограмма, к которой можно обратиться из другого места программы. Для создания функции используется ключевое слово *def*, после которого указывается *имя* и *список аргументов* в круглых скобках. Тело функции выделяется также как тело условия (или цикла) четырьмя пробелами.

```
def имя функции(Список параметров):
    Система команд
    return выражение
```

Определим функцию вычисления суммы цифр числа:

```
def sumD(n): # определение функции с параметром
    sumD = 0 #
    while n!= 0: # проверка условия n ≠ 0
        sumD += n % 10 # увеличение sumD на остаток
                       от деления n на 10
        n = n // 10  #  деление n на 10 с отбрасыванием остатка
    return sumD # возврат значения функции
# основная программа
print (sumD(int(input())) # вызов функции с параметром
```