[Модули 1](#_Toc154490181)

[Определение и подключение модулей 1](#_Toc154490182)

[Генерация байткода модулей 7](#_Toc154490183)

[Модуль random 9](#_Toc154490184)

[Модуль math 10](#_Toc154490185)

[Модуль locale 12](#_Toc154490186)

[Модуль decimal 14](#_Toc154490187)

[Модуль dataclass. Data-классы 18](#_Toc154490188)

[Строки 23](#_Toc154490189)

[Работа со строками 23](#_Toc154490190)

[Основные методы строк 29](#_Toc154490191)

[Форматирование 34](#_Toc154490192)

**Модули**

**Определение и подключение модулей**

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

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

Допустим, основной файл программы называется **main.py**. И мы хотим подключить к нему внешние модули.

Для этого сначала определим новый модуль: создадим в той же папке, где находится main.py, новый файл, который назовем **message.py**. По умолчанию интерпретатор Python ищет модули по ряду стандартных путей, один из которых - это папка главного, запускаемого скрипта. Поэтому, чтобы интерпретатор подхватил модуль message.py, для простоты оба файла поместим в один проект.

Соответственно модуль будет называться **message**. Определим в нем следующий код:

|  |  |
| --- | --- |
| 1  2  3  4  5 | hello = "Hello all"      def print\_message(text):      print(f"Message: {text}") |

Здесь определена переменная hello и функция print\_message, которая в качестве параметра получает некоторый текст и выводит его на консоль.

В основном файле программы - **main.py** используем данный модуль:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | import message      # подключаем модуль message    # выводим значение переменной hello  print(message.hello)        # Hello all  # обращаемся к функии print\_message  message.print\_message("Hello work")  # Message: Hello work |

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

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

Получив пространство имен модуля, мы сможем обратиться к его функциям по схеме

пространство\_имен.функция

Например, обращение к функции print\_message() из модуля message:

|  |  |
| --- | --- |
| 1 | message.print\_message("Hello work") |

И после этого мы можем запустить главный скрипт main.py, и он задействует модуль message.py. В частности, консольный вывод будет следующим:

Hello all

Message: Hello work

### Подключение функциональности модуля в глобальное пространство имен

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

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | from message import print\_message    # обращаемся к функии print\_message из модуля message  print\_message("Hello work")  # Message: Hello work    # переменная hello из модуля message не доступна, так как она не импортирована  # print(message.hello)  # print(hello) |

В данном случае мы импортируем из модуля message в глобальное пространство имен функцию print\_message(). Поэтому мы сможем ее использовать без указания пространства имен модуля как если бы она была определена в этом же файле.

Все остальные функции, переменные из модуля недоступны (как например, в примере выше переменная hello). Если мы хотим их также использовать, то их можно подключить по отдельности:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | from message import print\_message  from message import hello    # обращаемся к функции print\_message из модуля message  print\_message("Hello work")  # Message: Hello work    # обращаемся к переменной hello из модуля message  print(hello)    # Hello all |

Если необходимо импортировать в глобальное пространство имен весь функционал, то вместо названий отдельных функций и переменных можно использовать символ зводочки **\***:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | from message import \*    # обращаемся к функции print\_message из модуля message  print\_message("Hello work")  # Message: Hello work    # обращаемся к переменной hello из модуля message  print(hello)    # Hello all |

Но стоит отметить, что импорт в глобальное пространство имен чреват коллизиями имен функций. Например, если у нас том же файле определена функция с тем же именем до ее вызова, то будет вызываться функция, которая определена последней:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | from message import \*    print\_message("Hello work")  # Message: Hello work - применяется функция из модуля message    def print\_message(some\_text):      print(f"Text: {some\_text}")    print\_message("Hello work")  # Text: Hello work - применяется функция из текущего файла |

Таким образом, одноименная функция текущего файла скрывает функцию из подключенного модуля.

### Установка псевдонимов

При импорте модуля и его функциональности мы можем установить для них псевдонимы. Для этого применяется ключевое слово **as**, после которого указывается псевдоним. Например, установим псевдоним для модуля:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | import message as mes  # модуль message проецируется на псевдоним mes    # выводим значение переменной hello  print(mes.hello)        # Hello all  # обращаемся к функии print\_message  mes.print\_message("Hello work")  # Message: Hello work |

В данном случае пространство имен будет называться **mes**, и через этот псевдоним можно обращаться к функциональности модуля.

Подобным образом можно установить псевдонимы для отдельной функциональности модуля:

|  |  |
| --- | --- |
| 1  2  3  4  5 | from message import print\_message as display  from message import hello as welcome    print(welcome)          # Hello all - переменная hello из модуля message  display("Hello work")   # Message: Hello work - функция print\_message из модуля message |

Здесь для функции print\_message из модуля message устанавливается псевдоним display, а для переменной hello - псевдоним welcome. И через эти псевдонимы мы сможем к ним обращаться.

Псевдонимы могут быть полезны, когда нас не устраивают имена функций и переменных, например, они слишком длинные, и мы хотим их сократить, либо мы хотим дать им более описательные, с нашей точки зрения, имена. Либо если в текущем файле уже есть функциональность с теми же именами, и с помощью установки псевдонимов мы можем избежать конфликта имен. Например:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | from message import print\_message as display    def print\_message(some\_text):      print(f"Text: {some\_text}")    # функция print\_message из модуля message  display("Hello work")       # Message: Hello work    # функция print\_message из текущего файла  print\_message("Hello work")  # Text: Hello work |

### Имя модуля

В примере выше модуль main.py, который является главным, использует модуль message.py. При запуске модуля main.py программа выполнит всю необходимую работу. Однако, если мы запустим отдельно модуль message.py сам по себе, то ничего на консоли не увидим. Ведь модуль message просто определяет функцию и переменную и не выполняет никаких других действий. Но мы можем сделать так, чтобы модуль message.py мог использоваться как сам по себе, так и подключаться в другие модули.

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

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | hello = "Hello all"      def print\_message(text):      print(f"Message: {text}")      def main():      print\_message(hello)      if \_\_name\_\_ == "\_\_main\_\_":      main() |

В данном случае в модуль message.py для тестирования функциональности модуля добавлена функция main. И мы можем сразу запустить файл message.py отдельно от всех и протестировать код.

Следует обратить внимание на вызов функции main:

|  |  |
| --- | --- |
| 1  2 | if \_\_name\_\_ == "\_\_main\_\_":      main() |

Переменная **\_\_name\_\_** указывает на имя модуля. Для главного модуля, который непосредственно запускается, эта переменная всегда будет иметь значение **\_\_main\_\_** вне зависимости от имени файла.

Поэтому, если мы будем запускать скрипт message.py отдельно, сам по себе, то Python присвоит переменной \_\_name\_\_ значение \_\_main\_\_, далее в выражении if вызовет функцию main из этого же файла.

Однако если мы будем запускать другой скрипт, а этот - message.py - будем подключать в качестве вспомогательного, для message.py переменная \_\_name\_\_ будет иметь значение message. И соответственно метод main в файле message.py не будет работать.

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

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

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | import message      def main():      message.print\_message("Hello work")  # Message: Hello work      if \_\_name\_\_ == "\_\_main\_\_":      main() |

## Генерация байткода модулей

При выполнении скрипта на языке Python все выполнение в общем случае разбивается на две стадии:

1. Файл с кодом (файл с расширением **.py**) компилируется в промежуточный байткод.
2. Далее скомпилированный байткодом интерпретируется, то есть происходит собственно выполнение программы

При этом нам не надо явным образом генерировать никакой байткод, он создается неявно при выполнении скрипта Python. Если программа импортирует внешние модули/библиотеки и они импортируются первый раз, то их скомпилированный байткод сохраняется сохраняется в файле с расширением **.pyc** и кэшируется в каталоге **\_\_pycache\_\_** в папке, где расположен файл с кодом python. Если мы вносим в исходный файл библиотеки изменения, то Python перекомпилирует файл байткода. Если изменений в коде нет, то загружается ранее скомпилированный байткод из файла **\*.pyc**. Это позволяет оптимизировать работу с приложением, быстрее его компилировать и выполнять.

Однако байткод основного скрипта, который представляет основной файл программы и который передается интерпретатору python, не сохраняется в файле **\*.pyc** и перекомпилируется каждый раз при запуске приложения.

Допустим, в папке проекта у нас размещен файл **user.py** со простейшей функцией, которая принимает два параметра и выводит их значения:

|  |  |
| --- | --- |
| 1  2 | def printUser(username, userage):      print(f"Name: {username}  Age:{userage}") |

Подключим этот файл в главном модуле программы, который пусть называется **app.py**:

|  |  |
| --- | --- |
| 1  2  3  4  5 | import user  username = "Tom"  userage = 39    user.printUser(username, userage) |

При выполнении этого скрипта в папке проекте (где располагается модуль "user.py") будет создан каталог **\_\_pycache\_\_**. А в нем будет сгенерирован файл байткода, который будет наподобие следующего user.cpython-версия.pyc, где в качестве версии будет применяться версия используемого интерпретатора, например, 311 (для версии Python 3.11). Сгенерированный **pyc**-файл является бинарным, поэтому текстовом редакторе нет смысла его открывать.

### Ручная компиляция байткода

Хотя файл байткода создается автоматически, мы вручную можем его сгенерировать. Для этого есть несколько способов: компиляция с помощью скрипта **py\_compile** и компиляция с помощью модуля **compileall**.

Скрипт **py\_compile** применяется для компиляции отдельных файлов. Для компиляции произвольного скрипта **user.py** в файл с байткодом мы могли бы использовать следующую программу:

|  |  |
| --- | --- |
| 1  2  3 | import py\_compile    py\_compile.compile("user.py")   # передаем путь к скрипту |

Для компиляции в функцию compile() передаем путь к скрипту. После выполнения программы в текущей папке также будет сгенерирован каталог **\_\_pycache\_\_**, а в нем файл **user.cpython-311.pyc**

Модуль **compileall** применяется для компиляции всех файлов Python по определенным путям. Например, скомпилируем все файлы в каталоге **C:/python/files**

python -m compileall c:\python\files

По умолчанию компилируются даже те файлы, которые содержатся в подкаталогах. Если надо скомпилировать только те файлы, которые располагаются непосредственно в указанно папке, то применяется опция **-l**

python -m compileall c:\python\files -l

## Модуль random

Модуль random управляет генерацией случайных чисел. Его основные функции:

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

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

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | import random    number = random.random()  # значение от 0.0 до 1.0  print(number)  number = random.random() \* 100  # значение от 0.0 до 100.0  print(number) |

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

|  |  |
| --- | --- |
| 1  2  3  4 | import random    number = random.randint(20, 35)  # значение от 20 до 35  print(number) |

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

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

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | import random    number = random.randrange(10)  # значение от 0 до 10 не включая  print(number)  number = random.randrange(2, 10)  # значение в диапазоне 2, 3, 4, 5, 6, 7, 8, 9  print(number)  number = random.randrange(2, 10, 2)  # значение в диапазоне 2, 4, 6, 8  print(number) |

### Работа со списком

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

|  |  |
| --- | --- |
| 1  2  3  4  5 | numbers = [1, 2, 3, 4, 5, 6, 7, 8]  random.shuffle(numbers)  print(numbers)  random\_number = random.choice(numbers)  print(random\_number) |

## Модуль math

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

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

Пример применения некоторых функций:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33 | import math    # возведение числа 2 в степень 3  n1 = math.pow(2, 3)  print(n1)  # 8    # ту же самую операцию можно выполнить так  n2 = 2\*\*3  print(n2)    # квадратный корень числа  print(math.sqrt(9))  # 3    # ближайшее наибольшее целое число  print(math.ceil(4.56))  # 5    # ближайшее наименьшее целое число  print(math.floor(4.56))  # 4    # перевод из радиан в градусы  print(math.degrees(3.14159))  # 180    # перевод из градусов в радианы  print(math.radians(180))   # 3.1415.....  # косинус  print(math.cos(math.radians(60)))  # 0.5  # cинус  print(math.sin(math.radians(90)))   # 1.0  # тангенс  print(math.tan(math.radians(0)))    # 0.0    print(math.log(8,2))    # 3.0  print(math.log10(100))    # 2.0 |

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

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | import math  radius = 30  # площадь круга с радиусом 30  area = math.pi \* math.pow(radius, 2)  print(area)    # натуральный логарифм числа 10  number = math.log(10, math.e)  print(number) |

## Модуль locale

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

|  |  |
| --- | --- |
| 1  2  3  4 | # англосаксонская система  1,234.567  # европейская система  1.234,567 |

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

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

|  |  |
| --- | --- |
| 1 | setlocale(category, locale) |

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

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

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

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

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

Применяются следующие плейсхолдеры:

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

Перед каждым плейсхолдером ставится знак процента %, например:

|  |  |
| --- | --- |
| 1 | "%d" |

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

|  |  |
| --- | --- |
| 1 | %.2f        # два знака в дробной части |

Применим локализацию чисел и валют в немецкой культуре:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21 | import locale    locale.setlocale(locale.LC\_ALL, "de")        # для  Windows  # locale.setlocale(locale.LC\_ALL, "de\_DE")   # для MacOS    number = 12345.6789  formatted = locale.format\_string("%f", number)  print(formatted)    # 12345,678900    formatted = locale.format\_string("%.2f", number)  print(formatted)    # 12345,68    formatted = locale.format\_string("%d", number)  print(formatted)    # 12345    formatted = locale.format\_string("%e", number)  print(formatted)    # 1,234568e+04    money = 234.678  formatted = locale.currency(money)  print(formatted)    # 234,68 € |

Если вместо конкретного кода в качестве второго параметра передается пустая строка, то Python будет использовать культуру, которая применяется на текущей рабочей машине. А с помощью функции **getlocale()** можно получить эту культуру:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | import locale    locale.setlocale(locale.LC\_ALL, "")    number = 12345.6789  formatted = locale.format\_string("%.02f", number)  print(formatted)    # 12345,68  print(locale.getlocale())  # ('Russian\_Russia', '1251') - Windows  # ('ru\_RU', 'UTF-8')  - MacOS |

Стоит отметить, что в зависимости от системы вывод может отличаться.

## Модуль decimal

При работе с числами с плавающей точкой (то есть float) мы сталкиваемся с тем, что в результате вычислений мы получаем не совсем верный результат:

|  |  |
| --- | --- |
| 1  2 | number = 0.1 + 0.1 + 0.1  print(number)       # 0.30000000000000004 |

Проблему может решить использование функции **round()**, которая округлит число. Однако есть и другой способ, который заключается в использовании встроенного модуля **decimal**.

Ключевым компонентом для работы с числами в этом модуле является класс **Decimal**. Для его применения нам надо создать его объект с помощью конструктора. В конструктор передается строковое значение, которое представляет число:

|  |  |
| --- | --- |
| 1  2  3 | from decimal import Decimal    number = Decimal("0.1") |

После этого объект Decimal можно использовать в арифметических операциях:

|  |  |
| --- | --- |
| 1  2  3  4  5 | from decimal import Decimal    number = Decimal("0.1")  number = number + number + number  print(number)       # 0.3 |

В операциях с Decimal можно использовать целые числа:

|  |  |
| --- | --- |
| 1  2 | number = Decimal("0.1")  number = number + 2 |

Однако нельзя смешивать в операциях дробные числа float и Decimal:

|  |  |
| --- | --- |
| 1  2 | number = Decimal("0.1")  number = number + 0.1   # здесь возникнет ошибка |

С помощью дополнительных знаков мы можем определить, сколько будет символов в дробной части числа:

|  |  |
| --- | --- |
| 1  2  3 | number = Decimal("0.10")  number = 3 \* number  print(number)       # 0.30 |

Строка "0.10" определяет два знака в дробной части, даже если последние символы будут представлять ноль. Соответственно "0.100" представляет три знака в дробной части.

### Округление чисел

Объекты Decimal имеют метод **quantize()**, который позволяет округлять числа. В этот метод в качестве первого аргумента передается также объект Decimal, который указывает формат округления числа:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | from decimal import Decimal    number = Decimal("0.444")  number = number.quantize(Decimal("1.00"))  print(number)       # 0.44    number = Decimal("0.555678")  print(number.quantize(Decimal("1.00")))       # 0.56    number = Decimal("0.999")  print(number.quantize(Decimal("1.00")))       # 1.00 |

Используемая строка "1.00" указывает, что округление будет идти до двух знаков в дробной части.

По умолчанию округление описывается константой **ROUND\_HALF\_EVEN**, при котором округление происходит до ближайшего четного числа, если округляемая часть равна 5. Например:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | from decimal import Decimal, ROUND\_HALF\_EVEN      number = Decimal("10.025")      # 2 - ближайшее четное число  print(number.quantize(Decimal("1.00"), ROUND\_HALF\_EVEN))       # 10.02    number = Decimal("10.035")      # 4 - ближайшее четное число  print(number.quantize(Decimal("1.00"), ROUND\_HALF\_EVEN))       # 10.04 |

Стратегия округления передается в качестве второго параметра в quantize.

Строка "1.00" означает, что округление будет идти до двух чисел в дробной части. Но в первом случае "10.025" - вторым знаком идет 2 - четное число, поэтому, несмотря на то, что следующее число 5, двойка не округляется до тройки.

Во втором случае "10.035" - вторым знаком идет 3 - нечетное число, ближайшим четным числом будет 4, поэтому 35 округляется до 40.

Данное поведение при округлении, возможно, не всем покажется желательным, и в этом случае его можно переопределить, использовав одну из следующих констант:

* **ROUND\_HALF\_UP**: округляет число в сторону повышения, если после него идет число 5 или выше
* **ROUND\_HALF\_DOWN**: округляет число в сторону повышения, если после него идет число больше 5

|  |  |
| --- | --- |
| 1  2  3  4  5 | number = Decimal("10.026")  print(number.quantize(Decimal("1.00"), ROUND\_HALF\_DOWN))       # 10.03    number = Decimal("10.025")  print(number.quantize(Decimal("1.00"), ROUND\_HALF\_DOWN))       # 10.02 |

* **ROUND\_05UP**: округляет 0 до единицы, если после него идет число 5 и выше

|  |  |
| --- | --- |
| 1  2  3  4  5 | number = Decimal("10.005")  print(number.quantize(Decimal("1.00"), ROUND\_05UP))       # 10.01    number = Decimal("10.025")  print(number.quantize(Decimal("1.00"), ROUND\_05UP))       # 10.02 |

* **ROUND\_CEILING**: округляет число в большую сторону вне зависимости от того, какое число идет после него

|  |  |
| --- | --- |
| 1  2  3  4  5 | number = Decimal("10.021")  print(number.quantize(Decimal("1.00"), ROUND\_CEILING))       # 10.03    number = Decimal("10.025")  print(number.quantize(Decimal("1.00"), ROUND\_CEILING))       # 10.03 |

* **ROUND\_FLOOR**: не округляет число вне зависимости от того, какое число идет после него

|  |  |
| --- | --- |
| 1  2  3  4  5 | number = Decimal("10.021")  print(number.quantize(Decimal("1.00"), ROUND\_FLOOR))       # 10.02    number = Decimal("10.025")  print(number.quantize(Decimal("1.00"), ROUND\_FLOOR))       # 10.02 |

## Модуль dataclass. Data-классы

Модуль **dataclasses** предоставляет декоратор **dataclass**, который позволяет создавать **data-классы** - подобные позволяют значительно сократить шаблонный код классов. Как правило, такие классы предназначены для хранения некоторого состояния, некоторых данных и когда не требуется какое-то поведение в виде функций.

Рассмотрим простейший пример:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | class Person:      def \_\_init\_\_(self, name, age):          self.name = name          self.age = age    tom = Person("Tom", 38)  print(f"Name: {tom.name}  Age: {tom.age}")      # Name: Tom  Age: 38 |

Здесь определен класс Person, у которого в функции конструктора определены два атрибута: name и age. Далее создаем один объект этого класса и выводим значения его атрибутов на консоль.

Теперь изменим эту программу, сделав класс Person data-классом:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | from dataclasses import dataclass    @dataclass  class Person:      name: str      age: int    tom = Person("Tom", 38)  print(f"Name: {tom.name}  Age: {tom.age}")      # Name: Tom  Age: 38 |

Для создания data-класса импортируем из модуля dataclasses декоратор dataclass и применяем его к классу Person. И в этом случае в самом классе нам уже не надо указывать конструктор - функцию \_\_init\_\_. Мы просто указываем атрибуты. А Python потом сам сгенерирует конструктор, в который также мы можем передать значения для атрибутов объекта.

Таким образом, мы уже сократили определение класса и сделали его более простым. Но генерацией метода \_\_init\_\_ функциональность декоратора dataclass не ограничивается. В реальности data-класс

|  |  |
| --- | --- |
| 1  2  3  4 | @dataclass  class Person:      name: str      age: int |

будет аналогичен следующему:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | class Person:      def \_\_init\_\_(self, name, age):          self.name = name          self.age = age        def \_\_repr\_\_(self):          return f"Person(name={self.name!r}, age={self.age!r}"        def \_\_eq\_\_(self, other):          if other.\_\_class\_\_ is self.\_\_class\_\_:              return (self.name, self.age) == (other.name, other.age)          return NotImplemented |

В данном случае мы видим, что кроме функции \_\_init\_\_, также определяется функция \_\_repr\_\_() для возвращения строкового представления и функция \_\_eq\_\_() для сравнения двух объектов. Применение данных функций:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | from dataclasses import dataclass    @dataclass  class Person:      name: str      age: int      tom = Person("Tom", 38)  bob = Person("Bob", 42)  tomas = Person("Tom", 38)  print(tom == tomas)     # True  print(tom == bob)       # False  print(tom)              # Person(name="Tom", age=38) |

### Параметры декоратора dataclass

С помощью параметров декоратор dataclass позволяет сгенерировать дополнительный шаблонный код и вообще настроить генерацию кода:

|  |  |
| --- | --- |
| 1  2  3 | def dataclass(cls=None, /, \*, init=True, repr=True, eq=True, order=False,                unsafe\_hash=False, frozen=False, match\_args=True,                kw\_only=False, slots=False) |

Рассмотрим базовые параметры:

* **init**: если равно True, то генерируется функция \_\_init\_\_(). По умолчанию равно True
* **repr**: если равно True, то генерируется функция \_\_repr\_\_(), которая возвращает строковое представление объекта. По умолчанию равно True
* **eq**: если равно True, то генерируется функция \_\_eq\_\_(), которая сравнивает два объекта. По умолчанию равно True
* **order**: если равно True, то генерируются функции \_\_lt\_\_ (операция <), \_\_le\_\_ (<=), \_\_gt\_\_ (>), \_\_ge\_\_ (>=), которые применяются для упорядочивания объектов. По умолчанию равно False
* **unsafe\_hash**: если равно True, то генерируется функция \_\_hash\_\_(), которая возвращает хеш объекта. По умолчанию равно False

Кроме того, те функции, которые создаются по умолчанию, могут быть переопределены.

Применение параметров:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | from dataclasses import dataclass    @dataclass(unsafe\_hash=True, order=True)  class Person:      name: str      age: int      def \_\_repr\_\_(self):          return f"Person. Name: {self.name}  Age: {self.age}"      tom = Person("Tom", 38)  print(tom.\_\_hash\_\_())   # -421667297069596717  print(tom)              # Person. Name: Tom  Age: 38 |

В данном случае включаем генерирование хеша и функций упорядочивания, а также явным образом переопределяем функцию \_\_repr\_\_ для создания строкового представления объекта.

### значения по умолчанию

При необходимости атрибутам можно присвоить значения по умолчанию, если в конструкторе им не передаются значения:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | from dataclasses import dataclass    @dataclass  class Person:      name: str      age: int = 18      tom = Person("Tom", 38)  print(tom)              # Person(name="Tom", age=38)    bob = Person("Bob")  print(bob)              # Person(name="Bob", age=18) |

### Добавление дополнительного функционала

Хотя data-классы предназначены прежде всего для хранения различных данных, но также в них можно определять поведение с помощью дополнительных функций:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | from dataclasses import dataclass    @dataclass  class Person:      name: str      age: int        def say\_hello(self):          print(f"{self.name} says hello")      tom = Person("Tom", 38)  tom.say\_hello()     # Tom says hello |

# Строки

## Работа со строками

Строка представляет последовательность символов в кодировке Unicode, заключенных в кавычки. Причем для определения строк Python позволяет использовать как одинарные, так и двойные кавычики:

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

Если строка длинная, ее можно разбить 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  2  3 | text = ("Laudate omnes gentes laudate "          "Magnificat in secula ")  print(text) |

Если же мы хотим определить многострочный текст, то такой текст заключается в тройные двойные или одинарные кавычки:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | '''  Это комментарий  '''  text = '''Laudate omnes gentes laudate  Magnificat in secula  Et anima mea laudate  Magnificat in secula  '''  print(text) |

При использовани тройных одинарных кавычек не стоит путать их с комментариями: если текст в тройных одинарных кавычках присваивается переменной, то это строка, а не комментарий.

#### Управляющие последовательности в строке

Строка может содержать ряд специальных символов - управляющих последовательностей или escape-последовательности. Некоторые из них:

* **\**: позволяет добавить внутрь строки слеш
* **\'**: позволяет добавить внутрь строки одинарную кавычку
* **\"**: позволяет добавить внутрь строки двойную кавычку
* **\n**: осуществляет переход на новую строку
* **\t**: добавляет табуляцию (4 отступа)

Используем некоторые последовательностей:

|  |  |
| --- | --- |
| 1  2 | text = "Message:\n\"Hello World\""  print(text) |

Консольный вывод программы:

Message:

"Hello World"

Хотя подобные последовательности могут нам помочь в некоторых делах, например, поместить в строку кавычку, сделать табуляцию, перенос на другую строку. Но они также могут и мешать. Например:

|  |  |
| --- | --- |
| 1  2 | path = "C:\python\name.txt"  print(path) |

Здесь переменная path содержит некоторый путь к файлу. Однако внутри строки встречаются символы "\n", которые будут интерпретированы как управляющая последовательность. Так, мы получим следующий консольный вывод:

C:\python

ame.txt

Чтобы избежать подобной ситуации, перед строкой ставится символ **r**

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

### Вставка значений в строку

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

|  |  |
| --- | --- |
| 1  2  3  4 | userName = "Tom"  userAge = 37  user = f"name: {userName}  age: {userAge}"  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.

### Обращение к символам строки

И мы можем обратиться к отдельным символам строки по индексу в квадратных скобках:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | string = "hello world"  c0 = string[0]  # h  print(c0)  c6 = string[6]  # w  print(c6)    c11 = string[11]  # ошибка IndexError: string index out of range  print(c11) |

Индексация начинается с нуля, поэтому первый символ строки будет иметь индекс 0. А если мы попытаемся обратиться к индексу, которого нет в строке, то мы получим исключение IndexError. Например, в случае выше длина строки 11 символов, поэтому ее символы будут иметь индексы от 0 до 10.

Чтобы получить доступ к символам, начиная с конца строки, можно использовать отрицательные индексы. Так, индекс -1 будет представлять последний символ, а -2 - предпоследний символ и так далее:

|  |  |
| --- | --- |
| 1  2  3  4  5 | string = "hello world"  c1 = string[-1]  # d  print(c1)  c5 = string[-5]  # w  print(c5) |

При работе с символами следует учитывать, что строка - это неизменяемый (immutable) тип, поэтому если мы попробуем изменить какой-то отдельный символ строки, то мы получим ошибку, как в следующем случае:

|  |  |
| --- | --- |
| 1  2 | string = "hello world"  string[1] = "R" |

Мы можем только полностью переустановить значение строки, присвоив ей другое значение.

### Перебор строки

С помощью цикла **for** можно перебрать все символы строки:

|  |  |
| --- | --- |
| 1  2  3 | string = "hello world"  for char in string:      print(char) |

### Получение подстроки

При необходимости мы можем получить из строки не только отдельные символы, но и подстроку. Для этого используется следующий синтаксис:

* string[:end]: извлекается последовательность символов начиная с 0-го индекса по индекс end (не включая)
* string[start:end]: извлекается последовательность символов начиная с индекса start по индекс end (не включая)
* string[start:end:step]: извлекается последовательность символов начиная с индекса start по индекс end (не включая) через шаг step

Используем все варианты получения подстроки:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | string = "hello world"    # с 0 до 5 индекса  sub\_string1 = string[:5]  print(sub\_string1)      # hello    # со 2 до 5 индекса  sub\_string2 = string[2:5]  print(sub\_string2)      # llo    # с 2 по 9 индекса через один символ  sub\_string3 = string[2:9:2]  print(sub\_string3)      # lowr |

### Объединение строк

Одной из самых распространенных операций со строками является их объединение или конкатенация. Для объединения строк применяется операция сложения:

|  |  |
| --- | --- |
| 1  2  3  4 | name = "Tom"  surname = "Smith"  fullname = name + " " + surname  print(fullname)  # Tom Smith |

С объединением двух строк все просто, но что, если нам надо сложить строку и число? В этом случае необходимо привести число к строке с помощью функции **str()**:

|  |  |
| --- | --- |
| 1  2  3  4 | name = "Tom"  age = 33  info = "Name: " + name + " Age: " + str(age)  print(info)  # Name: Tom Age: 33 |

### Повторение строки

Для повторения строки определенное количество раз применяется операция умножения:

|  |  |
| --- | --- |
| 1  2 | print("a" \* 3)  # aaa  print("he" \* 4)  # hehehehe |

### Сравнение строк

Особо следует сказать о сравнении строк. При сравнении строк принимается во внимание символы и их регистр. Так, цифровой символ условно меньше, чем любой алфавитный символ. Алфавитный символ в верхнем регистре условно меньше, чем алфавитные символы в нижнем регистре. Например:

|  |  |
| --- | --- |
| 1  2  3  4  5 | str1 = "1a"  str2 = "aa"  str3 = "Aa"  print(str1 > str2)  # False, так как первый символ в str1 - цифра  print(str2 > str3)  # True, так как первый символ в str2 - в нижнем регистре |

Поэтому строка "1a" условно меньше, чем строка "aa". Вначале сравнение идет по первому символу. Если начальные символы обоих строк представляют цифры, то меньшей считается меньшая цифра, например, "1a" меньше, чем "2a".

Если начальные символы представляют алфавитные символы в одном и том же регистре, то смотрят по алфавиту. Так, "aa" меньше, чем "ba", а "ba" меньше, чем "ca".

Если первые символы одинаковые, в расчет берутся вторые символы при их наличии.

Зависимость от регистра не всегда желательна, так как по сути мы имеем дело с одинаковыми строками. В этом случае перед сравнением мы можем привести обе строки к одному из регистров.

Функция **lower()** приводит строку к нижнему регистру, а функция **upper()** - к верхнему.

|  |  |
| --- | --- |
| 1  2  3  4  5 | str1 = "Tom"  str2 = "tom"  print(str1 == str2)  # False - строки не равны    print(str1.lower() == str2.lower())  # True |

### Функции ord и len

Поскольку строка содержит символы Unicode, то с помощью функции **ord()** мы можем получить числовое значение для символа в кодировке Unicode:

|  |  |
| --- | --- |
| 1 | print(ord("A"))     # 65 |

Для получения длины строки можно использовать функцию **len()**:

|  |  |
| --- | --- |
| 1  2  3 | string = "hello world"  length = len(string)  print(length)   # 11 |

### Поиск в строке

С помощью выражения term in string можно найти подстроку term в строке string. Если подстрока найдена, то выражение вернет значение True, иначе возвращается значение False:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | string = "hello world"  exist = "hello" in string  print(exist)    # True    exist = "sword" in string  print(exist)    # False |

## Основные методы строк

Рассмотрим основные методы строк, которые мы можем применить в приложениях:

* **isalpha()**: возвращает True, если строка состоит только из алфавитных символов
* **islower()**: возвращает True, если строка состоит только из символов в нижнем регистре
* **isupper()**: возвращает True, если все символы строки в верхнем регистре
* **isdigit()**: возвращает True, если все символы строки - цифры
* **isnumeric()**: возвращает True, если строка представляет собой число
* **startswith(str)**: возвращает True, если строка начинается с подстроки str
* **endswith(str)**: возвращает True, если строка заканчивается на подстроку str
* **lower()**: переводит строку в нижний регистр
* **upper()**: переводит строку в вехний регистр
* **title()**: начальные символы всех слов в строке переводятся в верхний регистр
* **capitalize()**: переводит в верхний регистр первую букву только самого первого слова строки
* **lstrip()**: удаляет начальные пробелы из строки
* **rstrip()**: удаляет конечные пробелы из строки
* **strip()**: удаляет начальные и конечные пробелы из строки
* **ljust(width)**: если длина строки меньше параметра width, то справа от строки добавляются пробелы, чтобы дополнить значение width, а сама строка выравнивается по левому краю
* **rjust(width)**: если длина строки меньше параметра width, то слева от строки добавляются пробелы, чтобы дополнить значение width, а сама строка выравнивается по правому краю
* **center(width)**: если длина строки меньше параметра width, то слева и справа от строки равномерно добавляются пробелы, чтобы дополнить значение width, а сама строка выравнивается по центру
* **find(str[, start [, end])**: возвращает индекс подстроки в строке. Если подстрока не найдена, возвращается число -1
* **replace(old, new[, num])**: заменяет в строке одну подстроку на другую
* **split([delimeter[, num]])**: разбивает строку на подстроки в зависимости от разделителя
* **partition(delimeter)**: разбивает строку по разделителю на три подстроки и возвращает кортеж из трех элементов - подстрока до разделителя, разделитель и подстрока после разделителя
* **join(strs)**: объединяет строки в одну строку, вставляя между ними определенный разделитель

Например, если мы ожидаем ввод с клавиатуры числа, то перед преобразованием введенной строки в число можно проверить, с помощью метода isnumeric() введено ли в действительности число, и если так, то выполнить операцию преобразования:

|  |  |
| --- | --- |
| 1  2  3  4 | string = input("Введите число: ")  if string.isnumeric():      number = int(string)      print(number) |

Проверка, начинается или оканчивается строка на определенную подстроку:

|  |  |
| --- | --- |
| 1  2  3  4 | file\_name = "hello.py"    starts\_with\_hello = file\_name.startswith("hello")   # True  ends\_with\_exe = file\_name.endswith("exe")           # False |

Удаление пробелов в начале и в конце строки:

|  |  |
| --- | --- |
| 1  2  3 | string = "   hello  world!  "  string = string.strip()  print(string)           # hello  world! |

Дополнение строки пробелами и выравнивание:

|  |  |
| --- | --- |
| 1  2 | print("iPhone 7:", "52000".rjust(10))  print("Huawei P10:", "36000".rjust(10)) |

Консольный вывод:

iPhone 7: 52000

Huawei P10: 36000

### Поиск в строке

Для поиска подстроки в строке в Python применяется метод **find()**, который возвращает индекс первого вхождения подстроки в строку и имеет три формы:

* find(str): поиск подстроки str ведется с начала строки до ее конца
* find(str, start): параметр start задает начальный индекс, с которого будет производиться поиск
* find(str, start, end): параметр end задает конечный индекс, до которого будет идти поиск

Если подстрока не найдена, метод возвращает -1:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | welcome = "Hello world! Goodbye world!"  index = welcome.find("wor")  print(index)       # 6    # поиск с 10-го индекса  index = welcome.find("wor",10)  print(index)       # 21    # поиск с 10 по 15 индекс  index = welcome.find("wor",10,15)  print(index)       # -1 |

### Замена в строке

Для замены в строке одной подстроки на другую применяется метод **replace()**:

* replace(old, new): заменяет подстроку old на new
* replace(old, new, num): параметр num указывает, сколько вхождений подстроки old надо заменить на new. По умолчанию num равно -1, что соответствует первой версии метода и приводит к замене всех вхождений.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | phone = "+1-234-567-89-10"    # замена дефисов на пробел  edited\_phone = phone.replace("-", " ")  print(edited\_phone)     # +1 234 567 89 10    # удаление дефисов  edited\_phone = phone.replace("-", "")  print(edited\_phone)     # +12345678910    # замена только первого дефиса  edited\_phone = phone.replace("-", "", 1)  print(edited\_phone)     # +1234-567-89-10 |

### Разделение на подстроки

Метод **split()** разбивает строку на список подстрок в зависимости от разделителя. В качестве разделителя может выступать любой символ или последовательность символов. Данный метод имеет следующие формы:

* split(): в качестве разделителя используется пробел
* split(delimeter): в качестве разделителя используется delimeter
* split(delimeter, num): параметр num указывает, сколько вхождений delimeter используется для разделения. Оставшаяся часть строки добавляется в список без разделения на подстроки

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | text = "Это был огромный, в два обхвата дуб, с обломанными ветвями и с обломанной корой"  # разделение по пробелам  splitted\_text = text.split()  print(splitted\_text)  print(splitted\_text[6])     # дуб,    # разбиение по запятым  splitted\_text = text.split(",")  print(splitted\_text)  print(splitted\_text[1])     # в два обхвата дуб    # разбиение по первым пяти пробелам  splitted\_text = text.split(" ", 5)  print(splitted\_text)  print(splitted\_text[5])     # обхвата дуб, с обломанными ветвями и с обломанной корой |

Еще один метод - **partition()** разбивает строку по разделителю на три подстроки и возвращает кортеж из трех элементов - подстрока до разделителя, разделитель и подстрока после разделителя:

|  |  |
| --- | --- |
| 1  2  3  4 | text = "Это был огромный, в два обхвата дуб, с обломанными ветвями и с обломанной корой"  text\_parts = text.partition("дуб")  print(text\_parts)  # ('Это был огромный, в два обхвата ', 'дуб', ', с обломанными ветвями и с обломанной корой') |

Если разделитель с строке не найден, то возвращается кортеж с одной строкой.

### Соединение строк

При рассмотрении простейших операций со строками было показано, как объединять строки с помощью операции сложения. Другую возможность для соединения строк представляет метод **join()**: он объединяет список строк. Причем текущая строка, у которой вызывается данный метод, используется в качестве разделителя:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | words = ["Let", "me", "speak", "from", "my", "heart", "in", "English"]    # разделитель - пробел  sentence = " ".join(words)  print(sentence)  # Let me speak from my heart in English    # разделитель - вертикальная черта  sentence = " | ".join(words)  print(sentence)  # Let | me | speak | from | my | heart | in | English |

Вместо списка в метод join можно передать простую строку, тогда разделитель будет вставляться между символами этой строки:

|  |  |
| --- | --- |
| 1  2  3 | word = "hello"  joined\_word = "|".join(word)  print(joined\_word)      # h|e|l|l|o |

## Форматирование

В прошлых темах было рассмотрено, как можно вставлять в строку некоторые значения, предваряя строку символом **f**:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | first\_name="Tom"  text = f"Hello, {first\_name}."  print(text)     # Hello, Tom.    name="Bob"  age=23  info = f"Name: {name}\t Age: {age}"  print(info)     # Name: Bob  Age: 23 |

Но также в Python есть альтернативный способ, который предоставляет метод **format()**. Этот метод позволяет форматировать строку, вставляя в нее на место плейсхолдеров определенные значения.

Для вставки в строку используются специальные параметры, которые обрамляются фигурными скобками ({}).

### Именованные параметры

В форматируемой строке мы можем определять параметры, в методе format() передавать для этих параметров значения:

|  |  |
| --- | --- |
| 1  2  3  4  5 | text = "Hello, {first\_name}.".format(first\_name="Tom")  print(text)     # Hello, Tom.    info = "Name: {name}\t Age: {age}".format(name="Bob", age=23)  print(info)     # Name: Bob  Age: 23 |

Причем в метод формат аргументы определяются с тем же именем, что и параметры в строке. Так, если параметр называется first\_name, как в первом случае, то аргумент, которому присваивается значение, также называется first\_name.

### Параметры по позиции

Мы также можем последовательно передавать в метод format набор аргументов, а в самой форматируемой строке вставлять эти аргумента, указывая в фигурных скобках их номер (нумерация начинается с нуля):

|  |  |
| --- | --- |
| 1  2 | info = "Name: {0}\t Age: {1}".format("Bob", 23)  print(info)     # Name: Bob  Age: 23 |

При этом аргументы можно вставлять в строку множество раз:

|  |  |
| --- | --- |
| 1 | text = "Hello, {0} {0} {0}.".format("Tom") |

### Подстановки

Еще один способ передачи форматируемых значений в строку представляет использование подстановок или специальных плейсхолдеров, на место которых вставляются определенные значения. Для форматирования мы можем использовать следующие плейсхолдеры:

* **s**: для вставки строк
* **d**: для вставки целых чисел
* **f**: для вставки дробных чисел. Для этого типа также можно определить через точку количество знаков в дробной части.
* **%**: умножает значение на 100 и добавляет знак процента
* **e**: выводит число в экспоненциальной записи

Общий синтаксис плейсхолдера следующий:

|  |  |
| --- | --- |
| 1 | {:плейсхолдер} |

В зависимости от плейсхолдера можно добавлять дополнительные параметры. Например, для форматирования чисел float можно использовать следующие параметры

|  |  |
| --- | --- |
| 1 | {:[количество\_символов][запятая][.число\_знаков\_в\_дробной\_части] плейсхолдер} |

При вызове метода format в него в качестве аргументов передаются значения, которые вставляются на место плейсхолдеров:

|  |  |
| --- | --- |
| 1  2  3  4 | welcome = "Hello {:s}"  name = "Tom"  formatted\_welcome = welcome.format(name)  print(formatted\_welcome)        # Hello Tom |

В качестве результата метод format() возвращает новую отформатированную строку.

Форматирование целых чисел:

|  |  |
| --- | --- |
| 1  2  3  4 | source = "{:d} символов"  number = 5  target = source.format(number)  print(target)   # 5 символов |

Если форматируемое число больше 999, то мы можем указать в определении плейсхолдера, что мы хотим использовать запятую в качестве разделителя разрядов:

|  |  |
| --- | --- |
| 1  2 | source = "{:,d} символов"  print(source.format(5000))   # 5,000 символов |

Причем плейсхолдеры можно использовать и в f-строках:

|  |  |
| --- | --- |
| 1  2  3 | n = 5000  source = f"{n:,d} символов"  print(source)   # 5,000 символов |

Для дробных чисел, то есть таких, которые представляют тип float, перед кодом плейсхолдера после точки можно указать, сколько знаков в дробной части мы хотим вывести:

|  |  |
| --- | --- |
| 1  2  3  4  5 | number = 23.8589578  print("{:.2f}".format(number))   # 23.86  print("{:.3f}".format(number))   # 23.859  print("{:.4f}".format(number))   # 23.8590  print("{:,.2f}".format(10001.23554))    # 10,001.24 |

Еще один параметр позволяет установить минимальную ширину форматируемого значения в символах:

|  |  |
| --- | --- |
| 1  2 | print("{:10.2f}".format(23.8589578))    #     23.86  print("{:8d}".format(25))               #      25 |

Аналогичный пример с f-строками:

|  |  |
| --- | --- |
| 1  2  3  4 | n1 = 23.8589578  print(f"{n1:10.2f}")    #     23.86  n2 = 25  print(f"{n2:8d}")       #      25 |

Для вывода процентов лучше воспользоваться кодом "%":

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | number = .12345  print("{:%}".format(number))        # 12.345000%  print("{:.0%}".format(number))      # 12%  print("{:.1%}".format(number))      # 12.3%    print(f"{number:%}")        # 12.345000%  print(f"{number:.0%}")      # 12%  print(f"{number:.1%}")      # 12.3% |

Для вывода числа в экспоненциальной записи применяется плейсхолдер "e":

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | number = 12345.6789  print("{:e}".format(number))        # 1.234568e+04  print("{:.0e}".format(number))      # 1e+04  print("{:.1e}".format(number))      # 1.2e+04    print(f"{number:e}")        # 1.234568e+04  print(f"{number:.0e}")      # 1e+04  print(f"{number:.1e}")      # 1.2e+04 |

### Форматирование без метода format

Существует также еще один способ форматирования с помощью следующего синтаксиса:

|  |  |
| --- | --- |
| 1 | строка%(параметр1, параметр2,..параметрN) |

То есть в начале идет строка, которая содержит те же плейсхолдеры, которые были рассмотрены выше (за исключением плейсхолдера %), после строки ставится знак процента %, а затем список значений, которые вставляются в строку. Фактически знак процента представляют операцию, в результате которой образуется новая строка:

|  |  |
| --- | --- |
| 1  2 | info = "Имя: %s \t Возраст: %d" % ("Tom", 35)  print(info)   # Имя: Tom     Возраст: 35 |

Рядом с плейсхолдером указывается знак процента и в отличие от функции format здесь не требуются фигурные скобки.

Причем способы форматирования чисел здесь также применяются:

|  |  |
| --- | --- |
| 1  2 | number = 23.8589578  print("%0.2f  - %e" % (number, number))   # 23.86  - 2.385896e+01 |