В Python существуют несколько видов данных. Основные типы данных, с которыми вы столкнетесь – это **string**, **ingteger**, **float**, **list**, **dict** и **tuple**:

**Как создать строку**

Строки всегда создаются одним из трех способов. Вы можете использовать одинарные, двойные и тройные скобки. Давайте посмотрим

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | my\_string = "Добро пожаловать в Python!"  another\_string = 'Я новый текст тут...'    a\_long\_string = '''А это у нас  новая строка  в троичных скобках''' |

Строка с **тремя скобками** может быть создана с использованием трех одинарных скобок или трех двойных скобок. Так или иначе, с их помощью программист может писать строки в нескольких линиях. Если вы впишете это, вы увидите, что выдача сохраняет разрыв строк. Если вам нужно использовать одинарные скобки в вашей строке, то впишите двойные скобки. Давайте посмотрим на пример:

|  |  |
| --- | --- |
| 1  2  3 | my\_string = "I'm a Python programmer!"  otherString = 'Слово "Python" обычно подразумевает змею'  tripleString = """В такой "строке" мы можем 'использовать' все.""" |

Данный код демонстрирует то, как вы можете вписать одинарные или двойные скобки в строку. Существует еще один способ создания строки, при помощи **метода str**. Как это работает:

|  |  |
| --- | --- |
| 1  2 | my\_number = 123  my\_string = str(my\_number) |

Если вы впишете данный код в ваш интерпретатор, вы увидите, что вы изменили значение интегратора на строку и присвоили ее переменной my\_string. Это называется **кастинг**, или конвертирование. Вы можете конвертировать некоторые типы данных в другие, например числа в строки. Но вы также заметите, что вы не всегда можете делать обратное, например, конвертировать строку вроде ‘*ABC*’ в целое число. Если вы сделаете это, то получите ошибку вроде той, что указана в этом примере:

|  |  |
| --- | --- |
| 1  2  3  4  5 | int('ABC')    Traceback (most recent call last):      File "<string>", line 1, in <fragment>  ValueError: invalid literal for int() with base 10: 'ABC' |

|  |  |
| --- | --- |
| 1 | x = int("123") |

То все должно работать. Обратите внимание на то, что строка – это один из неизменных **типов Python**. Это значит, что вы не можете менять содержимое строки после ее создания. Давайте попробуем сделать это и посмотрим, что получится:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | my\_string = "abc"  my\_string[0] = "d"    Traceback (most recent call last):      File "<string>", line 1, in <fragment>  TypeError: 'str' object does not support item assignment |

Здесь мы пытаемся **изменить первую букву** с «*а»* на «*d*«, в итоге это привело к [ошибке TypeError](https://python-scripts.com/try-except-finally), которая не дает нам сделать это. Теперь вы можете подумать, что присвоение новой строке то же значение и есть изменение строки. Давайте взглянем, правда ли это:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | my\_string = "abc"  a = id(my\_string)  print(a) # 19397208    my\_string = "def"  b = id(my\_string)  print(b) # 25558288    my\_string = my\_string + "ghi"  c = id(my\_string)  print(c) # 31345312 |

Проверив id объекта, мы можем определить, что когда мы присваиваем новое значение переменной, то это меняет тождество. Обратите внимание, что в версии Python, начиная с 2.0, строки могут содержать только **символы ASCII**. Если вам нужен **Unicode**, тогда вы должны вписывать **u** перед вашей строкой. Пример:

|  |  |
| --- | --- |
| 1  2 | # -\*- coding: utf-8 -\*-  my\_unicode\_string = u"Это юникод!" |

В Python, начиная с версии 3, все **строки являются юникодом**.

**Конкатенация строк**

Конкатенация – это важный момент, это означает соединение или добавление двух объектов вместе. В нашем случае, нам нужно узнать, как добавить две строки вместе. Как вы можете догадаться, в Python эта операция очень простая:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | # -\*- coding: utf-8 -\*-    string\_one = "Собака съела "  string\_two = "мою книгу!"  string\_three = string\_one + string\_two    print(string\_three) # Собака съела мою книгу! |

Оператор **+** конкатенирует две строки в одну

**Методы строк**

Строка является **объектом** в Python. Например, допустим, у вас есть следующая строка:

|  |  |
| --- | --- |
| 1 | my\_string = "This is a string!" |

Теперь вам нужно сделать так, чтобы вся эта строка была в верхнем регистре. Чтобы сделать это, все, что вам нужно, это вызвать метод **upper**(), вот так:

|  |  |
| --- | --- |
| 1 | my\_string.upper() |

Если вы открыли ваш интерпретатор, вы также можете сделать то же самое:

|  |  |
| --- | --- |
| 1 | "This is a string!".upper() |

Существует великое множество других методов строк. Например, если вам нужно, что бы все было в **нижнем регистре**, вам нужно использовать метод **lower**(). Если вы хотите удалить все начальные и конечные пробелы, вам понадобится метод **strip**(). Для получения списка всех методов строк, впишите следующую команду в ваш интерпретатор:

|  |  |
| --- | --- |
| 1 | dir(my\_string) |

Вы увидите что-то на подобие этого:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | [‘\_\_add\_\_’, ‘\_\_class\_\_’, ‘\_\_contains\_\_’, ‘\_\_delattr\_\_’, ‘\_\_doc\_\_’, ‘\_\_eq\_\_’, ‘\_\_format\_\_’,  ‘\_\_ge\_\_’, ‘\_\_getattribute\_\_’, ‘\_\_getitem\_\_’, ‘\_\_getnewargs\_\_’, ‘\_\_getslice\_\_’, ‘\_\_gt\_\_’,  ‘\_\_hash\_\_’, ‘\_\_init\_\_’, ‘\_\_le\_\_’, ‘\_\_len\_\_’, ‘\_\_lt\_\_’, ‘\_\_mod\_\_’, ‘\_\_mul\_\_’, ‘\_\_ne\_\_’,  ‘\_\_new\_\_’, ‘\_\_reduce\_\_’, ‘\_\_reduce\_ex\_\_’, ‘\_\_repr\_\_’, ‘\_\_rmod\_\_’, ‘\_\_rmul\_\_’, ‘\_\_-  setattr\_\_’, ‘\_\_sizeof\_\_’, ‘\_\_str\_\_’, ‘\_\_subclasshook\_\_’, ‘\_formatter\_field\_name\_split’,  ‘\_formatter\_parser’, ‘capitalize’, ‘center’, ‘count’, ‘decode’, ‘encode’, ‘endswith’, ‘expandtabs’,  ‘find’, ‘format’, ‘index’, ‘isalnum’, ‘isalpha’, ‘isdigit’, ‘islower’, ‘isspace’,  ‘istitle’, ‘isupper’, ‘join’, ‘ljust’, ‘lower’, ‘lstrip’, ‘partition’, ‘replace’, ‘rfind’, ‘rindex’,  ‘rjust’, ‘rpartition’, ‘rsplit’, ‘rstrip’, ‘split’, ‘splitlines’, ‘startswith’, ‘strip’, ‘swapcase’,  ‘title’, ‘translate’, ‘upper’, ‘zfill’] |

Вы можете спокойно игнорировать методы, которые начинаются и заканчиваются двойным подчеркиванием, например **\_\_add\_\_**. Они не используются в ежедневном программировании в Python. Лучше обратите внимание на другие. Если вы хотите узнать, что делает тот или иной метод, просто обратитесь к справке. Например, если вы хотите узнать, зачем вам **capitalize**, впишите следующее, чтобы узнать:

|  |  |
| --- | --- |
| 1 | help(my\_string.capitalize) |

Вы получите следующую информацию:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | Help on built-in function capitalize:    capitalize(...)      S.capitalize() -> string    Выдача копии строки S только с заглавной буквой. |

Вы только что узнали кое-что о разделе, под названием **интроспекция**. Python может исследовать все свои объекты, что делает его очень легким в использовании. В основном, **интроспекция** позволяет вам спрашивать Python о нём. Вам может быть интересно, как сказать о том, какой тип переменной был использован (другими словами **int** или **string**). Вы можете спросить об этом у Python!

|  |  |
| --- | --- |
| 1 | type(my\_string) # <type 'str'> |

Как вы видите, тип переменной **my\_string** является **str**!

**Нарезка строк**

Одной из тем, которую вы часто будете делать на практике, является **нарезка строк**. Помню, меня удивило то, как часто мне нужно было узнать, как это делается в повседневной работе. Давайте посмотрим, как нарезка сработает в следующей строке:

|  |  |
| --- | --- |
| 1 | my\_string = "I like Python!" |

Каждый символ в строке может стать доступным при помощи слайсинга (**нарезки**). Например, если вам нужно взять только первый символ, вы можете сделать это следующим образом:

|  |  |
| --- | --- |
| 1 | print( my\_string[0:1] ) # I |

Таким образом, мы берем первый символ в строке до второго символа, но, не включая его. Да, в Python **отсчет ведется с нуля**. Это проще понять, если мы определим позицию каждого символа в таблице:

|  |  |
| --- | --- |
| 1 | 0 1 2 3 4 5 6 7 8 9 10 11 12 13 - I l i k e P y t h o n ! |

Таким образом, у нас есть строка длиной в 14 символов, начиная с нуля и до тринадцати. Давайте приведем несколько примеров, чтобы понять это лучше.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | my\_string[:1] # 'I'  my\_string[0:12] # 'I like Pytho'  my\_string[0:13] # 'I like Python'  my\_string[0:14] # 'I like Python!'  my\_string[0:-5] # 'I like Py'  my\_string[:] # 'I like Python!'  my\_string[2:] # 'like Python!' |

Как видно в данных примерах, мы можем назначить срез, лишь **указав его начало** (другими словами, *my\_string[2:]*), конец среза (*my\_string[:1]*), или оба (*my\_string[0:13]*). Мы можем даже использовать **отрицательные значения**, которые начинаются с конца строки. Так что в примере, где мы указали *my\_string[0:-5]*, начало ведется с нуля и заканчивается 5 символами, перед концом строки. Вы можете задаться вопросом «*Зачем мне это и где это можно применить?*». Лично я использовал это для разбора записей с фиксированной шириной в файлах, или ситуативно для **парсинга сложных названий файлов**, с очень специфическими наименованиями. Также я использовал это для **парсинга значений** в бинарных файлах. Любая работа, которая включает в себя обработку текстовых файлов, может быть намного проще, если вы понимаете, как работает нарезка и как эффективно использовать данный инструмент. Вы также можете получить доступ к отдельным символам в строке с помощью **индексации**. Например:

|  |  |
| --- | --- |
| 1 | print(my\_string[0]) # I |

Данный код выдаст первый символ в строке.

**Форматирование строк**

Форматирование строк (также известно как **замещение**) – это замещение значений в базовой строке. Большую часть времени вы будете вставлять строки внутри строк, однако, вам также понадобиться вставлять целые числа и числа с запятыми в строки весьма часто. Существует два способа достичь этой цели. Начнем с старого способа, после чего перейдем к новому:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | # -\*- coding: utf-8 -\*-    my\_string = "Я люблю %s" % "Python"  print(my\_string) # Я люблю Python    var = "яблоки"  newString = "Я ем %s" % var  print(newString) # Я ем яблоки    another\_string = "Я люблю %s и %s" % ("Python", var)  print(another\_string) # Я люблю Python и яблоки |

Как вы могли догадаться, **%** — это очень важная часть вышеописанного кода. Этот символ указывает Python, что вы скоро вставите текст на его место. Если вы будете следовать за строкой со знаком процента и другой строкой или переменной, тогда Python попытается вставить ее в строку. Вы можете вставить несколько строк, добавив несколько знаков процента в свою строку. Это видно в последнем примере. Обратите внимание на то, что когда вы добавляете больше одной строки, вам нужно **закрыть эти строки в круглые скобки**. Теперь взглянем на то, что случится, если мы вставим недостаточное количество строк:

|  |  |
| --- | --- |
| 1  2  3  4  5 | another\_string = "Я люблю %s и %s" % "Python"    Traceback (most recent call last):      File "<string>", line 1, in <fragment>  TypeError: not enough arguments for format string |

Мы не передали **необходимое количество аргументов** для [форматирования строки](https://python-scripts.com/string). Если вы внимательно взгляните на пример, вы увидите, что у нас есть два экземпляра **%**, но для того, чтобы вставить строки, вам нужно передать столько же %, сколько у нас строк. Теперь вы готовы к тому, чтобы узнать больше о вставке целых чисел, и чисел с запятыми. Давайте взглянем.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | my\_string = "%i + %i = %i" % (1,2,3)  print(my\_string) # '1 + 2 = 3'    float\_string = "%f" % (1.23)  print(float\_string) # '1.230000'    float\_string2 = "%.2f" % (1.23)  print(float\_string2) # '1.23'    float\_string3 = "%.2f" % (1.237)  print(float\_string3) # '1.24' |

Первый пример достаточно простой. Мы [создали строку](https://python-scripts.com/string), которая принимает три аргумента, и мы передаем их. В случае, если вы еще не поняли, Python не делает никаких дополнений в первом примере. Во втором примере, мы передаем число с запятой. Обратите внимание на то, что результат включает множество дополнительных нулей (1.230000). Нам это не нужно, так что мы указываем Python ограничить выдачу до двух десятичных значений в третьем примере (“**%.2f**”). Последний пример показывает, что Python **округлит числа** для вас, если вы передадите ему дробь, что лучше, чем два десятичных значения. Давайте взглянем на то, что произойдет, если мы передадим неправильные данные:

|  |  |
| --- | --- |
| 1  2  3  4 | int\_float\_err = "%i + %f" % ("1", "2.00")  Traceback (most recent call last):      File "<string>", line 1, in <fragment>  TypeError: %d format: a number is required, not str |

В данном примере мы передали две строки вместо целого числа и дроби. Это привело к **ошибке TypeError**, что говорит нам о том, что Python ждал от нас чисел. Это указывает на отсутствие передачи целого числа, так что мы исправим это, по крайней мере, попытаемся:

|  |  |
| --- | --- |
| 1  2  3  4  5 | int\_float\_err = "%i + %f" % (1, "2.00")    Traceback (most recent call last):      File "<string>", line 1, in <fragment>  TypeError: float argument required, not str |

Мы получили ту же ошибку, но под другим предлогом, в котором написано, что мы должны передать дробь. Как мы видим, Python предоставляет нам полезную информацию о том, что же пошло не так и как это исправить. Если вы исправите вложения надлежащим образом, тогда вы сможете запустить этот пример. Давайте перейдем к новому методу **форматирования строк**.

**Шаблоны и новая методика форматирования строк**

Этот метод был добавлен в Python 2.4 в виде шаблонов строк, но в качестве обычного метода string, работающего через **метод format** в версии 2.6. Так что это не самый свежий метод, просто обновленный.

|  |  |
| --- | --- |
| 1 | print("%(lang)s is fun!" % {"lang":"Python"}) # Python is fun! |

Должно быть это выглядит странно, но на самом деле мы сменили наши **%** на **%(lang)**, с тем отличием, что данный объект идет в комплекте с переменной. Вторая часть пример вызывает словарь Python. В основном, это пара **key:value**, так что когда Python ищет ключ **lang** в строке и в указанном словаре ключей, он заменяет этот ключ его значением. Давайте взглянем на следующие примеры:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | a = "%(value)s %(value)s %(value)s !" % {"value":"SPAM"}  print(a) # SPAM SPAM SPAM !    b = "%(x)i + %(y)i = %(z)i" % {"x":1, "y":2}  print(b)    Traceback (most recent call last):     File "<string>", line 1, in <fragment>  KeyError: 'z'    c = "%(x)i + %(y)i = %(z)i" % {"x":1, "y":2, "z":3}  print(c) # 1 + 2 = 3 |

В первом примере вы могли заметить, что мы передали только одно значение, но оно было вставлено три раза. Это одно из **преимуществ** использования шаблонов. Второй пример был загвоздкой, в которой мы **забыли передать** ключ z. В третьем примере эта проблема была исправлена с соответствующим результатом. Теперь давайте взглянем на то, что мы можем сделать, по аналогии с методом **форматирования строк**:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | a = "Python is as simple as {0}, {1}, {2}".format("a", "b", "c")  print(a) # 'Python is as simple as a, b, c'    b = "Python is as simple as {1}, {0}, {2}".format("a", "b", "c")  print(b) # 'Python is as simple as b, a, c'    xy = {"x":0, "y":10}  c = "Graph a point at where x={x} and y={y}".format(\*\*xy)  print(c) # Graph a point at where x=0 and y=10 |

В двух первых примерах вы можете увидеть, что мы можем передать объекты позиционно. Если мы перестроим порядок, мы получим немного другую выдачу. В последнем примере мы использовали словарь также, как мы использовали шаблоны ранее. Однако, нам нужно **извлечь словарь** при помощи двойной звездочки, чтобы он работал правильно. Существует множество других случаев, в которых используются строки, такие как определение ширины, выравнивание текста, конвертация в разные базы и многое другое.

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
* **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:

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

Проверка, начинается или оканчивается строка на определенную подстроку:

|  |
| --- |
| 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:

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

Дополнение строки пробелами и выравнивание:

|  |
| --- |
| 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:

|  |
| --- |
| 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

|  |
| --- |
| 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

|  |
| --- |
| 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 |

**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:

|  |
| --- |
| 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:

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