GeekBrains

Основы Python

Урок 2. Некоторые встроенные типы и операции с ними

На этом уроке:

1. Рассмотрим особенности наиболее часто встречающихся типов в языке Python.
2. Раскроем смысл понятий «объект», «атрибут», «метод».
3. Рассмотрим отличие кортежей и списков.
4. Разберём примеры работы со списками и строками.

Оглавление

Всё — объект

Объекты в Python — интроспекция. Списки

- Тип объекта
- Методы объекта
- Методы списков
- Реверс списков
- Срезы на примере списков
- Сортировка списков

Кортежи

*Нюансы присваивания в Python: copy и deepcopy

Работа со строками в Python: форматирование

- Коротко о строках в Python
- Форматирование строк: оператор %
- Форматирование строк: метод .format()
- Форматирование строк: f-строки

Работа со строками в Python: полезные методы

- Разделяй: метод .split()
- Соединяй: метод .join()
- Всё вверх или вниз: методы .upper() и .lower()
- Правим регистр: методы .title() и .capitalize()
- Реверс строк

Практическое задание

Дополнительные материалы

Используемая литература

Всё — объект

Для разминки вспомним пример из первого урока:

```
a = 14
print(type(a))  # <class 'int'>
```

Пришло время серьезно обсудить слово `class` из этого примера. В языке Python нет примитивных типов данных. Всё является объектом. Объект создаётся на основе некоторого описания, оно называется `class`. Несмотря на кажущуюся сложность, концепция объектов в программировании проста: они очень похожи на объекты реального мира. У объекта есть некоторые характеристики — мы их называем атрибутами. Также с объектами можно что-то делать — такие действия-глаголы называются методами.

В качестве примера рассмотрим пользователя сайта. В реальной жизни это человек, у которого есть имя, возраст, фотография, хобби. Компьютерное представление пользователя может быть реализовано в виде объекта класса `User`. Какие у него могут быть атрибуты? Как и у человека, только на английском языке: `name`, `age`, `avatar`, `interests`. Аналогично с методами: `.create()`, `.change_password()`, `.delete()`. Самая главная особенность любого объекта — он хранит свое состояние. Очень важно это сразу понять. Представим, что вы слушали трек на плеере и выключили его. Если после выключения воспроизведение начинается с того же самого места, значит, плеер запомнил своё состояние. Так может быть и с объектом пользователя — при создании мы задаём значения его атрибутов и получаем некоторое состояние. Оно будет сохранено, и в любой момент можно прочитать значение любого атрибута созданного пользователя.

Мы подошли к очень важному моменту — вопросу об изменяемости состояния объекта. То, что называют `mutable` и `immutable`. Можем ли мы менять состояние плеера? Да, можно менять громкость, выбирать новый трек, настраивать способ воспроизведения. Значит, плеер — `mutable` (изменяемый). Можем ли мы поменять возраст пользователя сайта, пароль, фотографию? Да, значит, объект пользователя тоже `mutable`.

Теперь другой пример. Мы печатаем текст в редакторе на компьютере. Можем его править? Да. Опять `mutable`. Если распечатать текст на бумаге — всё, править больше нельзя. Получили неизменяемый (`immutable`) объект. Можно внести правки на компьютере и заново распечатать, но это уже будет **ДРУГОЙ** объект. Очень важно понять это. Еще одна аналогия — CD и CD-RW-диски. Что будет `mutable`, а что `immutable`?

Примечание: если концепция деления всех объектов по признаку их изменяемости пока вам не очень понятна — не страшно. Постепенно придёт понимание, откуда она появилась и почему очень важна в современной разработке.

Объекты в Python — интроспекция. Списки

Тип объекта

Вспомните, при помощи какой функции мы определяли тип объекта на первом уроке. Так как в Python используется динамическая типизация, в программах часто нужно выполнить проверку типа переменных. Например, чтобы выполнить дополнительные преобразования: получили строку, а нам нужно число для выполнения арифметических операций.

Проверку типа можно сделать двумя способами:

```
winter_months = ['декабрь', 'январь', 'февраль']
print(type(winter_months) == list)  # True
print(isinstance(winter_months, list))  # True
```

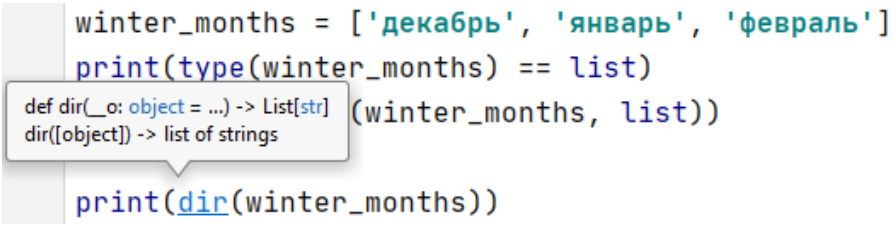
Первый способ очевиден: сравниваем результат функции `type()` с интересующим классом. **Важно:** класс нужно писать не в кавычках! Второй способ проще, и, следовательно, предпочтительнее — используем специальную функцию `isinstance()`.

Методы объекта

Мы умеем проверять тип объекта — можно ли узнать, какие у него есть методы и атрибуты? Да, для этого есть функция dir(). Продолжим пример:

```
...
print(dir(winter_months))
# ['__add__', '__class__', '__contains__', '__delattr__', '__delitem__',
#'__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__',
#'__getitem__', '__gt__', '__hash__', '__iadd__', '__imul__', '__init__',
#'__init_subclass__', '__iter__', '__le__', '__len__', '__lt__', '__mul__',
#'__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__',
#'__reversed__', '__rmul__', '__setattr__', '__setitem__', '__sizeof__',
#'__str__', '__subclasshook__', 'append', 'clear', 'copy', 'count', 'extend',
#'index', 'insert', 'pop', 'remove', 'reverse', 'sort']
```

Старайтесь сразу понимать, какой тип данных возвращает функция. В PyCharm есть трюк: можно, удерживая клавишу <Ctrl>, подвести курсор мыши к имени функции — должны увидеть контекстную справку:



Итак, функция `dir()` вернула нам список. В этом списке имена методов, которые можно вызывать для данного объекта. Некоторые имена начинаются и заканчиваются двойным нижним подчеркиванием — их называют «магическими», или dunder-методами. Пока эти методы останутся за кулисами нашего курса. Вы должны лишь знать, что они существуют. Роль остальных методов списков должна быть понятна. Если это не так, нужно подтянуть технический английский.

Методы списков

Давайте продолжим пример:

```
...
winter_months.append(12)
print(winter_months)  # ['декабрь', 'январь', 'февраль', 12]
winter_months.extend([1, 2])
print(winter_months)  # ['декабрь', 'январь', 'февраль', 12, 1, 2]
```

Метод `.append()`, как и положено, добавил объект в конец списка. Подумайте, зачем есть метод `.extend()`. Очень важно понимать тонкости — они сыграют огромную роль в будущем. Метод `.extend()` позволяет добавлять объекты «оптом». Альтернативный пример для чёткости:

```
...
winter_months.append(12)
print(winter_months)  # ['декабрь', 'январь', 'февраль', 12]
winter_months.append([1, 2])
print(winter_months)  # ['декабрь', 'январь', 'февраль', 12, [1, 2]]
```

Если вы не заметили разницы, смотрите ещё и ещё. Как добавились месяцы `1` и `2` теперь? Они добавились как **отдельный список**. По сути это ещё один уровень вложенности. В классическом программировании есть даже понятие «двумерный массив». Но это не то, что нам нужно, если хотим сохранить одно измерение и добавлять месяцы друг за другом. Именно эту задачу и решает метод `.extend()`.

Теперь поговорим еще об одном интересном и неочевидном методе списков — `.pop()`. Он работает как операция `cut` в редакторах — вырезает последний объект из списка и возвращает как результат:

```
winter_months = ['декабрь', 'январь', 'февраль']
```

```
cur_winter_month = winter_months.pop()
print(winter_months)  # ['декабрь', 'январь']
print(cur_winter_month)  # февраль
```

В этом примере мы забрали из списка последний объект и присвоили его переменной `cur_winter_month`. Зачем это нужно? Очень много алгоритмов построено на работе со стеком. Для их реализации идеально подходят списки благодаря методу `.pop()`. Если сейчас не очень понятно — узнаете про стек позже, на курсе «Алгоритмы и структуры данных». Вопросы про него очень любят задавать на собеседованиях.

На самом деле при помощи метода `.pop()` можно вырезать элемент из любой ячейки списка:

```
winter_months = ['декабрь', 'январь', 'февраль']
cur_winter_month = winter_months.pop(1)
print(winter_months)  # ['декабрь', 'февраль']
print(cur_winter_month)  # январь
```

Здесь мы явно указали индекс ячейки — числу `1` соответствует вторая ячейка (вспомнили, почему?). Очень часто в Python функции и методы имеют значения аргументов по умолчанию. Если для метода `.pop()` не указываем аргумент, он работает с последним элементом списка. Приведённый пример «плохой» с точки зрения алгоритмической сложности. Это очень важный материал для собеседований, и вы можете вернуться к его серьезному изучению позже. Но уже сейчас надо задумываться: «Хорошо ли я делаю с точки зрения О-большое»?

Представим, что у нас есть список из 10 000 элементов. У каждого элемента, разумеется, есть свой номер (индекс). Мы при помощи `.pop(0)` вырезаем первый элемент. Что теперь будет с индексами оставшихся 9 999 элементов? Они будут пересчитываться. То есть будет выполнено 9 999 операций изменения индекса. Мы сделали одно действие, которое привело к 9 999 действиям, — это не хорошо. Говоря языком алгоритмической сложности, операция `.pop(n)` стоит `O(n)`, в то время как операция `.pop()` стоит `O(1)`. Вывод: если вам захотелось написать где-то `.pop(0)`, надо очень хорошо подумать, как этого избежать.

Ещё пара полезных методов у списков — `.count()` и `.index()`. Пример:

```
basket_prices = [3000.0, 1580.0, 3000.0, 2785.8]
print(basket_prices.count(3000.0))  # 2
print(basket_prices.index(3000.0))  # 0
```

Как вы уже догадались, метод `.count()` считает, сколько объектов есть в списке. Если их нет, вернет `0`. Узнать индекс **первого(!)** вхождения объекта в список вы можете при помощи метода `.index()`. Этот метод может вызывать ошибку, если объекта в списке нет. Будьте аккуратны:

```
basket_prices = [3000.0, 1580.0, 3000.0, 2785.8]
print(basket_prices.index(4000.0))  # ... ValueError: 4000.0 is not in list...
```

Завершаем рассказ о списках редко используемыми методами `.insert()` и `.remove()`.

Первый позволяет вставить объект в нужную ячейку. По сути это расширенная версия `.append()`:

```
winter_months = ['декабрь', 'февраль']
print(winter_months)  # ['декабрь', 'февраль']
winter_months.insert(1, 'январь')
print(winter_months)  # ['декабрь', 'январь', 'февраль']
```

Добавили в ячейку и индексом `1` новое значение. Индексы ячеек, которые правее, пересчитались (плюс `1` к каждому). А если бы их было 9 999? Как и в случае с методом `.pop()`, наше действие привело бы к 9 999 дополнительным. Поэтому метод `.insert()` тоже дорогой с точки зрения алгоритмической сложности.

Метод `.remove()` позволяет удалить объект из списка. Вы можете спросить: в чём разница с методом `.pop()`? В метод `.pop()` мы передаем индекс элемента, а в метод `.remove()` — сам объект:

```
winter_months = ['декабрь', 'январь', 'январь', 'январь', 'февраль']
while winter_months.count('январь') > 1:
   winter_months.remove('январь')
   print(winter_months)
# ['декабрь', 'январь', 'январь', 'февраль']
# ['декабрь', 'январь', 'февраль']
```

Если объекта нет, этот метод вызовет ошибку:

```
...
winter_months.remove('апрель')
# ... ValueError: list.remove(x): x not in list ...
```

Реверс списков

В современной разработке есть важный термин: выполнение действий на месте, «in place». Это важная и непростая тема. Начнем с конкретного примера:

```
winter_months = ['декабрь', 'январь', 'февраль']
print(id(winter_months), winter_months)
# 4088392 ['декабрь', 'январь', 'февраль']
winter_months.reverse()
print(id(winter_months), winter_months)
# 4088392 ['февраль', 'январь', 'декабрь']
```

Что очевидно в этом примере? Объект остался тот же самый. Но порядок элементов изменился — мы реверсировали список при помощи метода `.reverse()`. Это и есть операция `in place`. Мы работали с исходным списком, не создавая новый. Это нужно для экономии памяти. Если размер списка будет 1 000 000 элементов, создание реверсированной копии может занять существенную часть оперативной памяти. Операции `in place` недопустимы, если в алгоритме нужно сохранить оригинальные данные. Еще один случай — требования по скорости. Алгоритмы `in place` могут быть медленнее, чем те, в которых создается новый объект.

Рассмотрим ещё один пример:

```
winter_months = ['декабрь', 'январь', 'февраль']
print(id(winter_months), winter_months)
# 34038344 ['декабрь', 'январь', 'февраль']
winter_months_reversed = list(reversed(winter_months))
print(id(winter_months_reversed), winter_months_reversed)
# 40793864 ['февраль', 'январь', 'декабрь']
```

Здесь мы также добились реверса списка, но при этом появился новый объект. Будьте готовы объяснить разницу на собеседовании. Пусть вас не смущает вызов функции `list()`, которая преобразовала результат работы функции `reversed()` в список. Подробности дальше.

Срезы на примере списков

Часто возникает необходимость поработать с частью списка, строки или другого контейнера. В Python для этих целей существуют срезы:

```
year_months = ['янв', 'фев', 'мар', 'апр', 'май', 'июн',
               'июл', 'авг', 'сен', 'окт', 'ноя', 'дек']
first_quarter = year_months[0:3]
print(first_quarter)  # ['янв', 'фев', 'мар']
other_first_quarter = year_months[:3]
print(other_first_quarter)  # ['янв', 'фев', 'мар']
```

В квадратных скобках указываем стартовую и финишную позиции среза. Как вы заметили из примера, `0` можно не писать, если срез от начала списка. Как понимать позиции — вопрос непростой. Можно думать, что левую границу включаем, а правую не включаем, ведь индексу `3` соответствует `'апр'`, а

он не вошел в срез. Есть версия, что это номера промежутков между элементами, тогда все красиво: `0` — промежуток до первого элемента, `3` — промежуток между `3-м` и `4-м` элементами. Всё, что внутри промежутка, берём. Но эта схема не будет верной для отрицательных индексов. Продолжаем пример:

```
...
second_quarter = year_months[3:6]
print(second_quarter)  # ['апр', 'май', 'июн']
last_quarter = year_months[9:]
print(last_quarter)  # ['окт', 'ноя', 'дек']
```

Заметили, что для среза, включающего последний элемент, не нужно писать индекс? Можно даже написать число, превышающее размер списка, — ошибки не будет:

```
...
second_half_year = year_months[6:1000]
print(second_half_year)  # ['июл', 'авг', 'сен', 'окт', 'ноя', 'дек']
```

Можно делать срезы с определённым шагом:

```
...
odd_months = year_months[::2]
print(odd_months)  # ['янв', 'мар', 'май', 'июл', 'сен', 'ноя']
even_months = year_months[1::2]
print(even_months)  # ['фев', 'апр', 'июн', 'авг', 'окт', 'дек']
```

В первом примере мы не указали ни левую, ни правую границы — значит, берём все элементы с шагом `2`. Шаг — третий аргумент среза. Если мы не задаем его явно, по умолчанию он равен `1`. Во втором примере задали левую границу — второй элемент списка, дальше тоже с шагом `2`. А что будет, если задать шаг `-1`? Реверс:

```
...
print(id(year_months))  # 195803464
year_reversed = year_months[::-1]
print(year_reversed)  # ['дек', 'ноя', 'окт', 'сен', 'авг', 'июл', ... ]
print(id(year_months), id(year_reversed))  # 195803464 195804232
```

Зачем мы здесь использовали функцию `id()`? Важно выяснить, создаётся ли новый объект при выполнении среза? Ответ: да. Значит это не `in place` алгоритм. Теперь вы знаете ещё один способ реверса.

Общая формула срезов `obj[start=0, stop=len(obj), step=1]`. Значение после знака «=» — значение по умолчанию. Оно применяется, если параметр не задан явно. Посмотрите на это в примерах выше.

Важно! Срезы можно применять и к строкам.

Сортировка списков

Знания, которые вы получили в разделе «Реверс списков», можно закрепить для решения задачи сортировки. Всё аналогично:

```
winter_months = ['декабрь', 'январь', 'февраль']
print(id(winter_months))  # 195803208
winter_months.sort()
print(id(winter_months))  # 195803208
print(winter_months)  # ['декабрь', 'февраль', 'январь']
```

Как работает метод `.sort()`? Вы уже должны понимать это из кода. Это `in-place` сортировка, то есть новый объект не создаётся. **Будьте аккуратны**: метод `.sort()`, как и метод `.reverse()`, возвращает `None`.

Замечание: Строки сортируются, как список контактов в телефоне. У каждой буквы есть код (в `Python` используется кодировка `utf-8`) — при сортировке сравниваются коды букв. Сначала сортируем по первой букве, потом по второй и т.д. Код символа можно узнать при помощи функции `ord()`.

Аналогом функции `reversed()` является функция `sorted()`:

```
winter_months = ['декабрь', 'январь', 'февраль']
print(id(winter_months))  # 195807944
winter_months_s = sorted(winter_months, reverse=True)
print(id(winter_months_s), type(winter_months_s))  # 195807816 <class 'list'>
print(winter_months_s)  # ['январь', 'февраль', 'декабрь']
```

Видим, что появился новый объект — отсортированный список. В этом примере показали, что можно задавать дополнительный аргумент для сортировки `reverse`. Он работает и для метода `.sort()`. **Следует отметить**, что, несмотря на сходство, функции `reversed()` и `sorted()` работают внутри по-разному. Первая возвращает итератор (узнаем подробнее в будущем), а вторая — список. Это может быть важно на собеседовании.

Кортежи

В Python есть тип данных, очень похожий на список: tuple — кортеж. Это тоже контейнер, но с особенностью: он **неизменяемый** (immutable — уже обсуждали сегодня)! Это значит, что к нему нельзя добавлять элементы, нельзя удалять, нельзя изменять содержимое ячеек. Что можно? Получать значения по индексу и итерировать в циклах — всё как у списков. Зачем было урезать функционал списков? Во-первых, уменьшился размер объекта:

```
import sys

some_list = ['hello', True, 'word', 1, 2.2]
print(type(some_list), sys.getsizeof(some_list), some_list)
# <class 'list'> 104 ['hello', True, 'word', 1, 2.2]
some_tuple = ('hello', True, 'word', 1, 2.2)
print(type(some_tuple), sys.getsizeof(some_tuple), some_tuple)
# <class 'tuple'> 88 ('hello', True, 'word', 1, 2.2)
```

Здесь использовали ключевое слово `import` для импорта модуля `sys`, в котором есть функция `getsizeof()`, позволяющая узнать размер объекта в Python. Разговор о модулях был на первом уроке. Вот и первый конкретный пример. Выводы из примера: размер списка 104 байта, а кортежа — 88 байт. Информация хранится одна и та же. Также вы должны были заметить, что кортежи создаются внутри круглых скобок, а не квадратных, как у списков. Отметим важный факт: кортеж создается **запятой** (!!!), а **круглые скобки** — это дополнительный элемент синтаксиса, полезный для форматирования в несколько строк:

```
some_tuple = ('hello', True, 'word', 1, 2.2)
some_tuple_2 = 'hello', True, 'word', 1, 2.2
```

Совет: попробуйте посмотреть результат функции `dir()` для кортежа — сколько методов вы увидели? Меньше, чем у списков?

Во-вторых, бывают ситуации, когда можно использовать только неизменяемый тип данных, например, в качестве ключа словаря (о них позже). Главное помнить, что неизменяемые типы данных проще в реализации и, как правило, дают преимущество в объёме занимаемой памяти и скорости.

Ещё один пример:

```
some_list = ['hello', True, 'word', 1, 2.2]
some_list[0] = 'hi'
print(some_list)  # ['hi', True, 'word', 1, 2.2]
some_tuple = ('hello', True, 'word', 1, 2.2)
some_tuple[0] = 'hi'
print(some_tuple)
# ...TypeError: 'tuple' object does not support item assignment
```

Для списка спокойно присвоили новое значение в первую ячейку. Если вам непонятно, почему в случае с кортежем мы увидели ошибку — значит, надо ещё раз почитать про `mutable` и `immutable`. В будущем, если увидели такую ошибку — значит, работаете с неизменяемым типом.

Некоторые **изменяемые типы данных в Python**: списки (`list`), словари (`dict`), множества (`set`).

Некоторые **неизменяемые типы данных в Python**: строки (`str`), целые числа (`int`), вещественные числа (`float`), неизменяемые множества (`frozenset`).

*Нюансы присваивания в Python: copy и deepcopy

Python бывает коварен. Рассмотрим пример:

```
a = [['hello'], 10]
b = a
print(id(a), id(b))  # 3891848 3891848
b[1] = 15
print(a, b)  # [['hello'], 15] [['hello'], 15]
print(id(a), id(b))  # 3891848 3891848
```

Неожиданно изменения в переменной `b` коснулись и переменной `a`. Причина — при присваивании Python не создает новый объект, а просто связывает переменную `b` с тем же объектом, что и переменную `a`. Цель — оптимизация производительности, экономия памяти. Это известный прием copy-on-write. Что делать, если нужна реальная копия (подумайте несколько раз: а нужна ли)? Создать её или при помощи среза (лайфхак):

```
a = [['hello'], 10]
b = a[:]
print(id(a), id(b))  # 4874888 239496520
print(id(a[0]), id(b[0]))  # 4874824 4874824
b[1] = 15
print(a, b)  # [['hello'], 10] [['hello'], 15]
```

Неспроста этот раздел помечен звёздочкой. Видим, что теперь `a` и `b` — разные объекты. Но список, который лежит в первой ячейке каждого из них, — это всё ещё один и тот же объект. Это ещё одна оптимизация Python — он создаёт поверхностную (shallow) копию объекта. Альтернативный срезу вариант с использованием метода `.copy()`:

```
a = [['hello'], 10]
b = a.copy()
print(id(a), id(b))  # 30433928 65563016
b[1] = 15
print(a, b)  # [['hello'], 10] [['hello'], 15]
```

или модуля copy:

```
from copy import copy

a = [['hello'], 10]
b = copy(a)
print(id(a), id(b))  # 33907336 63072968
b[1] = 15
print(a, b)  # [['hello'], 10] [['hello'], 15]
```

Преимущество модуля copy проявится, когда вы захотите сделать полную копию (deep copy):

```
from copy import deepcopy

a = [['hello'], 10]
b = deepcopy(a)
print(id(a), id(b))  # 30433928 239233608
print(id(a[0]), id(b[0]))  # 30433864 30934216
b[1] = 15
print(a, b)  # [['hello'], 10] [['hello'], 15]
```

Теперь и находящиеся в первых ячейках списки — это разные объекты. На самом деле `deepcopy` не так часто используется, но вопрос на собеседовании задать могут. Нужно понимать нюансы. Тем, кто только начинает изучать программирование, рекомендуем почитать этот раздел ещё раз позже — сейчас, скорее всего, будет много новой информации (это нормально).

Работа со строками в Python: форматирование

Коротко о строках в Python

После списков и кортежей рассмотрим еще один очень популярный в Python-разработке класс `str` — строки. Строки — это **неизменяемые последовательности** unicode-символов. В этом предложении очень много информации. Постарайтесь прочитать его несколько раз. Раз строки — это последовательности, их можно итерировать в цикле for in. Для строк доступен функционал срезов, как и для списков. Строки неизменяемы, как кортежи. Код любого символа можно посмотреть при помощи функции `ord()`, обратная ей функция `chr()` возвращает символ по коду:

```
message = 'привет всем'
print(message[:message.index(' ')])  # привет
print(ord(message[0]))  # 1087
print(chr(ord(message[0]) - 32))  # П
print(message[::-1])  # месв тевирп
```

Сделали срез до первого пробела (метод `.index()` работает, как и для списков). Узнали код первой буквы строки и нашли букву с кодом на 32 меньше. Оказалось, это такая же буква, но заглавная. Этот лайфхак можно иногда использовать для замены больших букв маленькими и наоборот. Разумеется, надо проверить, работает ли это для всех букв интересующего алфавита. Вы уже догадались, как в цикле `for in range()` вывести все русские или английские буквы?

*Пример, над которым рекомендуем задуматься:

```
raw_message = ['python', 'современный', 'язык']
message = ''
for item in raw_message:
   message += item  # new object!
   message += ' '  # new object!
print(message)  # python современный язык
```

Сложение строк в Python можно делать при помощи оператора `+`, иначе это называется конкатенация. Но что не так в этом цикле? В отличие от изменяемых списков, для которых операция добавления (метод `.append()`) не приводит к созданию нового объекта, строки неизменяемы. Это значит, что при конкатенации **всегда** создается новый объект класса `str`. Это достаточно «дорогая» операция, связанная с выделением новой памяти и созданием в ней определённой структуры данных. Представьте, что вы заняты важным делом и кто-то внезапно отвлекает вас, заставляя переключиться на новую задачу, — хорошо ли это? Ещё один момент: когда вы создаёте новый объект и старый уже не нужен (а в нашем цикле так и будет), он будет уничтожен сборщиком мусора. То есть в этом примере новые объекты строк создаются и тут же уничтожаются. В итоге мы получим желаемый результат — собранную из элементов списка строку, но хорошо ли мы решили задачу? Нет. Начинающим разработчикам рекомендуем позже переосмыслить этот пример. Как быть? Использовать специальный метод строк `.join()`:

```
raw_message = ['python', 'современный', 'язык']
message = ' '.join(raw_message)
print(message)  # python современный язык
```

В этом примере в очередной раз можно убедиться в интуитивной понятности Python, правда, если прочитать код справа-налево: элементы `raw_message` соединить при помощи `' '`.

Форматирование строк: оператор %

Давайте представим, что нам нужно сформировать много-много поздравлений вида: «Уважаемый, <Имя>! Поздравляем с <год> годом!» Где <Имя> и <год> — некоторые меняющиеся значения. Первое, что приходит в голову, — использовать конкатенацию:

```
name, year = 'Иван', 2021
greeting = 'Уважаемый, ' + name + '! Поздравляем с ' + str(year) + ' годом!'
print(greeting)
# Уважаемый, Иван! Поздравляем с 2021 годом!
```

Этот пример, с одной стороны, простой — склеили строку по кусочкам. С другой стороны, есть нюансы: использовали функцию `str()` для преобразования целого числа в строку. Python — язык со строгой (сильной) типизацией. Это значит, что строки можно складывать только со строками. Также мы применили здесь известный трюк — параллельное присваивание. Если мы слева и справа от оператора присваивания разместим кортежи **одинакового** размера, первой переменной слева будет присвоено значение первого элемента справа, второй — второго и т.д. Такой синтаксис очень часто используют в Python. Те, кто не знают, написали бы так:

```
name = 'Иван'
year = 2021
```

Согласитесь, это скучно и неэффективно. Но будьте аккуратны: важно, чтобы размер последовательностей справа и слева был одинаков! У нас с обеих сторон по два элемента.

Вернёмся к примеру со строкой. Легко ли вам представить результат конкатенации? Обилие знаков `+` и кавычек сводит к нулю читабельность этого кода. Поэтому были реализованы специальные способы формирования динамических строк — их называют форматированием.

Первый, самый старый — использование printf-подобного оператора `%`:

```
greeting = 'Уважаемый, %s! Поздравляем с %d годом!' % (name, year)
print(greeting)
# Уважаемый, Иван! Поздравляем с 2021 годом!
```

Функция вывода в консоль printf в том или ином виде существует во многих современных языках, берущих свое начало от Си. Разработчики Python тоже не обошли её стороной. В строке мы размещаем так называемые спецификаторы для преобразования (conversion) величин, начинающиеся с символа `%`:

Преобразование	Значение
`'d'`	знаковое десятичное целое
`'i'`	знаковое десятичное целое

`'e'`	число с плавающей запятой, экспоненциальный формат (нижний регистр)
`'E'`	число с плавающей запятой, экспоненциальный формат (верхний регистр)
`'f'`	число с плавающей запятой
`'F'`	число с плавающей запятой
`'c'`	единичный символ (целое число или строка, состоящая из одного символа)
`'r'`	строка (используется Python функция `repr()`)
`'s'`	строка (используется Python функция `str()`).
`'a'`	строка (используется Python функция `ascii()`).
`'%'`	без конвертации, для вывода символа %

После оператора `%` пишем одну переменную или кортеж, если необходимо выполнить подстановку нескольких значений в строку. В нашем примере в строке было два спецификатора, поэтому написали кортеж из двух переменных.

Следует отметить, что данный способ форматирования используется не так часто: для совместимости со старыми версиями языка, для защиты от SQL-инъекций, для логирования при помощи модуля logging.

Форматирование строк: метод `.format()`

Гораздо чаще в коде вы встретите форматирование при помощи специального метода строк `.format()`:

```
greeting = 'Уважаемый, {}! Поздравляем с {} годом!'.format(name, year)
```

Получили тот же самый результат. Здесь в строке необходимо прописать плейсхолдеры в виде фигурных скобок. Две подстановки — два плейсхолдера. На самом деле синтаксис форматирования строк таким способом даёт очень много возможностей: можно задавать выравнивание, количество выводимых знаков для чисел и дополнять нулём. Настоятельно рекомендуем вернуться к этой теме позже. Пока приведём более продвинутый пример:

```
name, year, month, money = 'Борис', 2021, 3, 1789.47689
mes = '{2}! Сегодня {1} месяц {0} года.'.format(year, month, name)
mes_2 = '{2:^15}! Сегодня {1:02d} месяц {0} года.'.format(year, month, name)
mes_3 = '{name:>15}! На счете {money:.2f}'.format(name=name, money=money)
print(mes)
# Борис! Сегодня 3 месяц 2021 года.
print(mes_2)
#      Борис     ! Сегодня 03 месяц 2021 года.
print(mes_3)
#           Борис! На счете 1789.48
```

При формировании первого сообщения использовали принудительную подстановку по позициям аргументов (как обычно, нумерация с нуля).

Во втором сообщении дополнительно задали ширину поля для вывода имени `15` символов и выравнивание по середине. Для месяца задали тип — целое число, ширину — два разряда и попросили дополнять нулём — получили `03`.

В третьем сообщении использовали именованные аргументы — это очень удобно, когда строка состоит из множества подстановок. Имя выровняли по правому краю, а для вещественного числа ограничили число знаков после запятой двумя — `Python` всё автоматически округлил.

Форматирование строк: `f-строки`

Если вы используете версию Python 3.6 или новее, лучше всего применять для форматирования f-строки. Это самый компактный и читабельный способ, к тому же лучший в плане производительности:

```
greeting = f'Уважаемый, {name}! Поздравляем с {year} годом!'
```

Оригинальный пример стал ещё проще. В чём отличие? Перед строкой пишем литеру `f`. Теперь переменные можно размещать прямо внутри плейсхолдеров. Продвинутый пример теперь будет выглядеть так:

```
mes_2 = f'{name:^15}! Сегодня {month:02d} месяц {year} года.'
mes_3 = f'{name:>15}! На счете {money:.2f}'
```

Фактически мы просто убрали «лишнее» — `.format()`.

Работа со строками в Python: полезные методы

Разделяй: метод `.split()`

В конце урока познакомимся с одной из сильных сторон Python — возможностями при работе со строками.

Первая задача: дан адрес `"https://geekbrains.ru/teacher/lessons/79615"`. Нужно его распарсить — придать определённую структуру, понять, что есть что.

Обычно при парсинге начинают с разделения исходной строки на части по определённой схеме. В нашем примере очевидно, что надо делить по слешу:

```
url = 'https://geekbrains.ru/teacher/lessons/79615'
url_parts = url.split('/')
print(url_parts)
# ['https:', '', 'geekbrains.ru', 'teacher', 'lessons', '79615']
```

Что произошло? Метод `.split()` «распилил» строку по символу `/` и вернул нам список. Дальше можно придавать элементам этого списка определённое значение: первый элемент — протокол передачи плюс двоеточие, второй элемент — пустышка, третий элемент — доменное имя, остальные элементы — элементы адреса ресурса в рамках домена.

*Можно записать решение более красиво (но сложно!):

```
_t_protocol, _, domain, *resource_address = url.split('/')
t_protocol = _t_protocol[:-1]
print(t_protocol, domain, resource_address)
# https geekbrains.ru ['teacher', 'lessons', '79615']
```

Для понимания этого кода нужны все полученные на курсе знания. Временные переменные в `Python`-разработке мы обычно обозначаем префиксом «_». Если значение переменной нам принципиально не нужно, можем вообще обозначить ее «_» — в нашем случае это второй элемент списка (пустышка). Как вы уже догадались, мы использовали параллельное присваивание, но более продвинутую версию — последнюю переменную записали с префиксом `*` (распаковка). По сути мы говорим: «Всё остальное нужно в виде списка поместить в эту переменную». Зачем? Если справа от знака присваивания может быть больше элементов, чем слева. Попробуйте убрать * — что получилось? Ошибка, ведь слева `4` переменные, а справа список из `6` элементов. Важно: переменная `resource_address` в этом примере всегда будет списком. Для тех, кто любит вникать, ещё одна версия:

```
_t_protocol, _, domain, *resource_address = url.split('/')[:3]
```

Тренируем интуицию — будет ли ошибка (ведь слева 4 переменные, а справа список из 3 элементов)? Если нет, то что будет в переменной `resource_address`?

Соединяй: метод `.join()`

С этим методом мы уже познакомились. Главное помнить, что он корректно склеивает последовательности, состоящие только из строк:

```
raw_url = ['https:', '', 'geekbrains.ru', 'teacher', 'lessons', '79615']
url = '/'.join(raw_url)
print(url)
# https://geekbrains.ru/teacher/lessons/79615
raw_url_2 = ['https:', '', 'geekbrains.ru', 'teacher', 'lessons', 79615]
url_2 = '/'.join(raw_url_2)
print(url_2)
# ...TypeError: sequence item 5: expected str instance, int found
```

Надеемся, что вы разобрались, почему во второй раз выскочила ошибка.

Всё вверх или вниз: методы `.upper()` и `.lower()`

Тут всё очень просто — вся строка переводится в верхний или нижний регистр:

```
msg = 'Товаров в корзине: 5'
print(msg.upper())  # ТОВАРОВ В КОРЗИНЕ: 5
print(msg.lower())  # товаров в корзине: 5
```

Правим регистр: методы `.title()` и `.capitalize()`

Тут тоже всё очень просто — либо у всех слов первая буква становится заглавной, либо только первая:

```
msg = 'тОВАров в коРЗИНЕ: 5. стоимость: 4789,5 руб.'
print(msg.title())  # Товаров В Корзине: 5. Стоимость: 4789,5 Руб.
print(msg.capitalize())  # Товаров в корзине: 5. стоимость: 4789,5 руб.
```

Реверс строк

Для реверса строк используем срез с шагом -1:

```
msg = 'а роза упала на лапу Азора'
print(msg[::-1])  # арозА упал ан алапу азор а
```

*Совет: попробуйте сделать реверс через функцию `reversed()`. Сразу не получится и наверняка придется использовать `.join()`. Поэтому используйте её для реверса, только если в алгоритме нужен будет генератор (о них пойдёт речь на одном из следующих уроков).

Практическое задание

1. Выяснить тип результата выражений:
   - 15 * 3
   - 15 / 3
   - 15 // 2
   - 15 ** 2

2. Дан список:

```
['в', '5', 'часов', '17', 'минут', 'температура', 'воздуха',
'была', '+5', 'градусов']
```

   Необходимо его обработать — обособить каждое целое число (вещественные не трогаем) кавычками (добавить кавычку до и кавычку после элемента списка, являющегося числом) и дополнить нулём до двух целочисленных разрядов:

```
['в', '"', '05', '"', 'часов', '"', '17', '"', 'минут',
'температура', 'воздуха', 'была', '"', '+05', '"', 'градусов']
```

   Сформировать из обработанного списка строку:

```
в "05" часов "17" минут температура воздуха была "+05" градусов
```

   Подумать, какое условие записать, чтобы выявить числа среди элементов списка? Как модифицировать это условие для чисел со знаком?
   Примечание: если обособление чисел кавычками не будет получаться - можете вернуться к его реализации позже. Главное: дополнить числа до двух разрядов нулём!

3. *(вместо задачи 2) Решить задачу 2 не создавая новый список (как говорят, in place). Эта задача намного серьёзнее, чем может сначала показаться.

4. Дан список, содержащий искажённые данные с должностями и именами сотрудников:

```
['инженер-конструктор Игорь', 'главный бухгалтер МАРИНА',
'токарь высшего разряда нИКОЛАй', 'директор аэлита']
```

   Известно, что имя сотрудника всегда в конце строки. Сформировать из этих имён и вывести на экран фразы вида: `'Привет, Игорь!'` Подумать, как получить имена сотрудников из элементов списка, как привести их к корректному виду. Можно ли при этом не создавать новый список?

5. Создать вручную список, содержащий цены на товары (10–20 товаров), например:

```
[57.8, 46.51, 97, ...]
```

A. Вывести на экран эти цены через запятую в одну строку, цена должна отображаться в виде <r> руб <kk> коп (например «5 руб 04 коп»). Подумать, как из цены получить рубли и копейки, как добавить нули, если, например, получилось 7 копеек или 0 копеек (должно быть 07 коп или 00 коп).
B. Вывести цены, отсортированные по возрастанию, новый список не создавать (доказать, что объект списка после сортировки остался тот же).
C. Создать новый список, содержащий те же цены, но отсортированные по убыванию.
D. Вывести цены пяти самых дорогих товаров. Сможете ли вывести цены этих товаров по возрастанию, написав минимум кода?

Задачи со * предназначены для продвинутых учеников, которым мало сделать обычное задание.

Дополнительные материалы

1. Лутц Марк. Изучаем Python.
2. Встроенные типы Python.

Используемая литература

Для подготовки данного методического пособия были использованы следующие ресурсы:

1. https://realpython.com/python-string-formatting/.
2. https://docs.python.org/3/.