# Print

Большую роль в Python играют отступы. Неправильно поставленный отступ фактически является ошибкой. Например, в следующем случае мы получим ошибку, хотя код будет практически аналогичен приведенному выше:

|  |  |
| --- | --- |
| 1  2 | print(2 + 3)      print("Hello") |

Поэтому стоит помещать новые инструкции сначала строк

# Input

Если функция print отвечает за вывод, то функция **input** отвечает за ввод информации. В качестве необязательного параметра эта функция принимает приглашение к вводу и возвращает введенную строку, которую мы можем сохранить в переменную:

|  |  |
| --- | --- |
| 1  2 | name = input("Введите имя: ")  print("Привет", name) |

# Типы данных

* **boolean** - логическое значение True или False
* **int** - представляет целое число, для хранения которого использует 4 байта в памяти компьютера.
* **float** - представляет число с плавающей точкой, для хранения которого используется 8 байт, например, 1.2 или 34.76
* **complex** - комплексные числа
* **str** - строки, например "hello". В Python 3.x строки представляют набор символов в кодировке Unicode
* **bytes** - последовательность чисел в диапазоне 0-255
* **byte array** - массив байтов, аналогичен bytes с тем отличием, что может изменяться
* **list** - список
* **tuple** - кортеж
* **set** - неупорядоченная коллекция уникальных объектов
* **frozen set** - то же самое, что и set, только не может изменяться (immutable)
* **dict** - словарь, где каждый элемент имеет ключ и значение

# Операции

* **//**

Целочисленное деление двух чисел:

|  |  |
| --- | --- |
| 1  2 | print(7 / 2)  # 3.5  print(7 // 2)  # 3 |

Данная операция возвращает целочисленный результат деления, отбрасывая дробную часть

* **\*\***

Возведение в степень:

|  |  |
| --- | --- |
| 1 | print(6 \*\* 2)  # Возводим число 6 в степень 2. Результат - 36 |

* **%**

Получение остатка от деления:

|  |  |
| --- | --- |
| 1 | print(7 % 2)  # Получение остатка от деления числа 7 на 2. Результат - 1 |

# Преобразование чисел и строк

Мы ожидаем, что "2" + 3 будет равно 5. Однако этот код сгенерирует исключение, так как первое число на самом деле представляет строку. И чтобы все заработало как надо, необходимо привести строку к числу с помощью функции int():

|  |  |
| --- | --- |
| 1  2  3  4 | first\_number = "2"  second\_number = 3  third\_number = int(first\_number) + second\_number  print(third\_number) # 5 |

В этот случае для округления результата мы можем использовать функцию **round()**:

|  |  |
| --- | --- |
| 1  2  3  4 | first\_number = 2.0001  second\_number = 0.1  third\_number = first\_number + second\_number  print(round(third\_number, 4))  # 2.1001 |

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 |

# Системы представления чисел

При обычном определении числовой переменной она получает значение в десятичной системе. Но кроме десятичной в Python мы можем использовать двоичную, восьмеричную и шестнадцатиричную системы.

Для определения числа в двоичной системе перед его значением ставится 0 и префикс **b**:

|  |  |
| --- | --- |
| 1 | x = 0b101   # 101 в десятичной системе равно 5 |

Для определения числа в восьмеричной системе перед его значением ставится 0 и префикс **o**:

|  |  |
| --- | --- |
| 1 | a = 0o11    # 11 в восьмеричной системе равно 9 |

Для определения числа в шестнадцатиричной системе перед его значением ставится 0 и префикс **x**:

|  |  |
| --- | --- |
| 1 | y = 0x0a        # a в десятичной системе равно 11 |

И с числами в других системах измерения также можно проводить арифметические операции:

|  |  |
| --- | --- |
| 1  2  3  4 | x = 0b101   # 5  y = 0x0a    # 10  z = x + y   # 15  print("{0} in binary {0:08b}   in hex {0:02x} in octal {0:02o}".format(z)) |

Для вывода числа в различных системах исчисления используются функция format, которая вызывается у строки. В эту строку передаются различные форматы. Для двоичной системы "{0:08b}", где число 8 указывает, сколько знаков должно быть в записи числа. Если знаков указано больше, чем требуется для числа, то ненужные позиции заполняются нулями. Для шестнадцатиричной системы применяется формат "{0:02x}". И здесь все аналогично - запись числа состоит из двух знаков, если один знак не нуден, то вместо него вставляется ноль. А для записи в восьмеричной системе испольуется формат "{0:02o}".

# Ескейп-последовательности

# Эскейп-последовательности

Кроме стандартных символов строки могут включать управляющие эскейп-последовательности, которые интерпретируются особым образом. Например, последовательность **\n** представляет перевод строки. Поэтому следующее выражение:

|  |  |
| --- | --- |
| 1 | print("Время пришло в гости отправится\nждет меня старинный друг") |

На консоль выведет две строки:

Время пришло в гости отправится

ждет меня старинный друг

Тоже самое касается и последовательности **\t**, которая добавляет табляцию.

Строки lower upper сравнение

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 |

Кроме того, существуют символы, которые вроде бы сложно использовать в строке. Например, кавычки. И чтобы отобразить кавычки (как двойные, так и одинарные) внутри строки, перед ними ставится слеш:

|  |  |
| --- | --- |
| 1 | print("Кафе \"Central Perk\"" |

# If elif else

Если необходимо ввести несколько альтернативных условий, то можно использовать дополнительные блоки **elif**, после которого идет блок инструкций.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | age = 18  if age >= 21:      print("Доступ разрешен")  elif age >= 18:      print("Доступ частично разрешен")  else:      print("Доступ запрещен") |

age = 18

if age >= 18:

    print("Больше 17")

    if age > 21:

        print("Больше 21")

    else:

        print("От 18 до 21")

# Циклы

Все инструкции, которые относятся к циклу while, располагаются на последующих строках и должны иметь отступ от начала строки.

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | choice = "y"    while choice.lower() == "y":      print("Привет")      choice = input("Для продолжения нажмите Y, а для выхода любую другую клавишу: ")  print("Работа программы завешена") |

#! Программа по вычислению факториала

number = int(input("Введите число: "))

factorial = 1

for i in range(1, number+1):

    factorial \*= i

print("Факториал числа", number, "равен", factorial)

Функция range имеет следующие формы:

* **range(stop)**: возвращает все целые числа от 0 до stop
* **range(start, stop)**: возвращает все целые числа в промежутке от start (включая) до stop (не включая). Выше в программе факториала использована именно эта форма.
* **range(start, stop, step)**: возвращает целые числа в промежутке от start (включая) до stop (не включая), которые увеличиваются на значение step

Примеры вызовов функции range:

|  |  |
| --- | --- |
| 1  2  3  4 | range(5)            # 0, 1, 2, 3, 4  range(1, 5)         # 1, 2, 3, 4  range(2, 10, 2)     # 2, 4, 6, 8  range(5, 0, -1)     # 5, 4, 3, 2, 1 |

Одни циклы внутри себя могут содержать другие циклы. Рассмотрим на примере вывода таблицы умножения:

|  |  |
| --- | --- |
| 1  2  3  4 | for i in range(1, 10):      for j in range(1, 10):          print(i \* j, end="\t")      print("\n") |

# Break continue

Оператор break может использоваться, если в цикле образуются условия, которые несовместимы с его дальнейшим выполнением. Рассмотрим следующий пример:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | #! Программа Обменный пункт    print("Для выхода нажмите Y")    while True:      data = input("Введите сумму для обмена: ")      if data.lower() == "y":          break  # выход из цикла      money = int(data)      cache = round(money / 56, 2)      print("К выдаче", cache, "долларов")    print("Работа обменного пункта завершена") |

# Функции

Определение функции начинается с выражения **def**, которое состоит из имени функции, набора скобок с параметрами и двоеточия. Параметры в скобках необязательны. А со следующей строки идет блок инструкций, которые выполняет функция. Все инструкции функции имеют отступы от начала строки.

Например, определение простейшей функции:

|  |  |
| --- | --- |
| 1  2 | def say\_hello():      print("Hello") |

### Значения по умолчанию

Некоторые параметры функции мы можем сделать необязательными, указав для них значения по умолчанию при определении функции. Например:

|  |  |
| --- | --- |
| 1  2  3  4  5 | def say\_hello(name="Tom"):      print("Hello,", name)    say\_hello()  say\_hello("Bob") |

### Именованные параметры

При вызове функции первое значение "Tom" передается первому параметру - параметру name, второе значение - число 22 передается второму параметру - age. И так далее по порядку. Использование именованных параметров позволяет переопределить порядок передачи:

|  |  |
| --- | --- |
| 1  2  3  4 | def display\_info(name, age):      print("Name:", name, "\t", "Age:", age)    display\_info(age=22, name="Tom") |

Именованные параметры предполагают указание имени параметра с присвоением ему значения при вызове функции.

### Return

 Python функция может возвращать сразу несколько значений:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | def create\_default\_user():      name = "Tom"      age = 33      return name, age      user\_name, user\_age = create\_default\_user()  print("Name:", user\_name, "\t Age:", user\_age) |

### Main

### Функция main

В программе может быть определено множество функций. И чтобы всех их упорядочить, хорошей практикой считается добавление специальной функции main, в которой потом уже вызываются другие функции:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | def main():      say\_hello("Tom")      usd\_rate = 56      money = 30000      result = exchange(usd\_rate, money)      print("К выдаче", result, "долларов")      def say\_hello(name):      print("Hello,", name)      def exchange(usd\_rate, money):      result = round(money/usd\_rate, 2)      return result    # Вызов функции main  main() |

# Variable scope

Если же мы хотим изменить в локальной функции глобальную переменную, а не определить локальную, то необходимо использовать ключевое слово **global**:

|  |  |
| --- | --- |
| 1  2  3  4 | def say\_bye():      global name      name = "Bob"      print("Good bye", name) |

# Модули

Соответственно модуль будет называться **account**. И определим в нем следующий код:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | def calculate\_income(rate, money, month):      if money <= 0:          return 0        for i in range(1, month+1):          money = round(money + money \* rate / 100 / 12, 2)      return money      def main():      rate = 10      money = 100000      period = 12        result = calculate\_income(rate, money, period)      print("Параметры счета:\n", "Сумма: ", money, "\n", "Ставка: ", rate, "\n",            "Период: ", period, "\n", "Сумма на счете в конце периода: ", result)    if \_\_name\_\_=="\_\_main\_\_":      main() |

Здесь определена функция calculate\_income, которая в качестве параметров получает процентную ставку вклада, сумму вклада и период, на который делается вклад, и высчитывает сумму, которая получится в конце данного периода. Кроме того, для тестирования функции определена главная функция main. И мы можем сразу запустить и протестировать код.

Теперь используем этот модуль в файле **hello.py**:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | #! Программа Банковский счет  import account      def main():      rate = int(input("Введите процентную ставку: "))      money = int(input("Введите сумму: "))      period = int(input("Введите период ведения счета в месяцах: "))        result = account.calculate\_income(rate, money, period)      print("Параметры счета:\n", "Сумма: ", money, "\n", "Ставка: ", rate, "\n",            "Период: ", period, "\n", "Сумма на счете в конце периода: ", result)    if \_\_name\_\_ == "\_\_main\_\_":      main() |

# Exception

Рассмотрим обработку исключения на примере преобразовании строки в число:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | try:      number = int(input("Введите число: "))      print("Введенное число:", number)  except:      print("Преобразование прошло неудачно")  print("Завершение программы") |

Вводим строку:

Введите число: hello

Преобразование прошло неудачно

Завершение программы

Если ситуация такова, что в программе могут быть сгенерированы различные типы исключений, то мы можем их обработать по отдельности, используя дополнительные выражения except:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | try:      number1 = int(input("Введите первое число: "))      number2 = int(input("Введите второе число: "))      print("Результат деления:", number1/number2)  except ValueError:      print("Преобразование прошло неудачно")  except ZeroDivisionError:      print("Попытка деления числа на ноль")  except Exception:      print("Общее исключение")  print("Завершение программы") |

### Блок finally

При обработке исключений также можно использовать необязательный блок **finally**. Отличительной особенностью этого блока является то, что он выполняется вне зависимости, было ли сгенерировано исключение:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | try:      number = int(input("Введите число: "))      print("Введенное число:", number)  except ValueError:      print("Не удалось преобразовать число")  finally:      print("Блок try завершил выполнение")  print("Завершение программы") |

### Получение информации об исключении

С помощью оператора **as** мы можем передать всю информацию об исключении в переменную, которую затем можно использовать в блоке except:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | try:      number = int(input("Введите число: "))      print("Введенное число:", number)  except ValueError as e:      print("Сведения об исключении", e)  print("Завершение программы") |

# Raise

### Генерация исключений

Иногда возникает необходимость вручную сгенерировать то или иное исключение. Для этого применяется оператор **raise**.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | try:      number1 = int(input("Введите первое число: "))      number2 = int(input("Введите второе число: "))      if number2 == 0:          raise Exception("Второе число не должно быть равно 0")      print("Результат деления двух чисел:", number1/number2)  except ValueError:      print("Введены некорректные данные")  except Exception as e:      print(e)  print("Завершение программы") |

# Lists

Конструктор list для создания списока может принимать другой список:

|  |  |
| --- | --- |
| 1  2 | numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9]  numbers2 = list(numbers) |

Если необходимо создать список, в котором повторяется одно и то же значение несколько раз, то можно использовать символ звездочки \*. Например, определим список из шести пятерок:

|  |  |
| --- | --- |
| 1  2 | numbers = [5] \* 6  # [5, 5, 5, 5, 5, 5]  print(numbers) |

Кроме того, если нам необходим последовательный список чисел, то для его создания удобно использовать функцию **range**, которая имеет три формы:

* range(end): создается набор чисел от 0 до числа end
* range(start, end): создается набор чисел от числа start до числа end
* range(start, end, step): создается набор чисел от числа start до числа end с шагом step

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | numbers = list(range(10))  print(numbers)      # [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]  numbers = list(range(2, 10))  print(numbers)      # [2, 3, 4, 5, 6, 7, 8, 9]  numbers = list(range(10, 2, -2))  print(numbers)      # [10, 8, 6, 4] |

### Перебор элементов

Для перебора элементов можно использовать как цикл for, так и цикл while.

Перебор с помощью цикла for:

|  |  |
| --- | --- |
| 1  2  3 | companies = ["Microsoft", "Google", "Oracle", "Apple"]  for item in companies:      print(item) |

Перебор с помощью цикла while:

|  |  |
| --- | --- |
| 1  2  3  4  5 | companies = ["Microsoft", "Google", "Oracle", "Apple"]  i = 0  while i < len(companies):      print(companies[i])      i += 1 |

Для перебора с помощью функции **len()** получаем длину списка. С помощью счетчика i выводит по элементу, пока значение счетчика не станет равно длине списка.

### Сравнение списков

Два списка считаются равными, если они содержат один и тот же набор элементов:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9]  numbers2 = list(range(1,10))  if numbers == numbers2:      print("numbers equal to numbers2")  else:      print("numbers is not equal to numbers2") |

### Методы и функции по работе со списками

Для управления элементами списки имеют целый ряд методов. Некоторые из них:

* **append(item)**: добавляет элемент item в конец списка
* **insert(index, item)**: добавляет элемент item в список по индексу index
* **remove(item)**: удаляет элемент item. Удаляется только первое вхождение элемента. Если элемент не найден, генерирует исключение ValueError
* **clear()**: удаление всех элементов из списка
* **index(item)**: возвращает индекс элемента item. Если элемент не найден, генерирует исключение ValueError
* **pop([item])**: удаляет и возвращает элемент по индексу index. Если индекс не передан, то просто удаляет последний элемент.
* **count(item)**: возвращает количество вхождений элемента item в список
* **sort([key])**: сортирует элементы. По умолчанию сортирует по возрастанию. Но с помощью параметра key мы можем передать функцию сортировки.
* **reverse()**: расставляет все элементы в списке в обратном порядке

Кроме того, Python предоставляет ряд встроенных функций для работы со списками:

* **len(list)**: возвращает длину списка
* **sorted(list, [key])**: возвращает отсортированный список
* **min(list)**: возвращает наименьший элемент списка
* **max(list)**: возвращает наибольший элемент списка

### Копирование списков

При копировании списков следует учитывать, что списки представляют изменяемый (mutable) тип, поэтому если обе переменных будут указывать на один и тот же список, то изменение одной переменной, затронет и другую переменную:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | users1 = ["Tom", "Bob", "Alice"]  users2 = users1  users2.append("Sam")  # users1 и users2 указывают на один и тот же список  print(users1)   # ["Tom", "Bob", "Alice", "Sam"]  print(users2)   # ["Tom", "Bob", "Alice", "Sam"] |

Это так называемое "поверхностное копирование" (shallow copy). И, как правило, такое поведение нежелательное. И чтобы происходило копирование элементов, но при этом переменные указывали на разные списки, необходимо выполнить глубокое копирование (deep copy). Для этого можно использовать метод **deepcopy()**, который определен во встроенном модуле **copy**:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | import copy    users1 = ["Tom", "Bob", "Alice"]  users2 = copy.deepcopy(users1)  users2.append("Sam")  # пееменные users1 и users2 указывают на разные списки  print(users1)   # ["Tom", "Bob", "Alice"]  print(users2)   # ["Tom", "Bob", "Alice", "Sam"] |

### Копирование части списка

Если необходимо скопировать не весь список, а только его какую-то определенную часть, то мы можем применять специальный синтаксис. который может принимать следующие формы:

* list[:end]: через параметр end передается индекс элемента, до которого нужно копировать список
* list[start:end]: параметр start указывает на индекс элемента, начиная с которого надо скопировать элементы
* list[start:end:step]: параметр step указывает на шаг, через который будут копироваться элементы из списка. По умолчанию этот параметр равен 1.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | users = ["Tom", "Bob", "Alice", "Sam", "Tim", "Bill"]    slice\_users1 = users[:3]   # с 0 по 3  print(slice\_users1)   # ["Tom", "Bob", "Alice"]    slice\_users2 = users[1:3]   # с 1 по 3  print(slice\_users2)   # ["Bob", "Alice"]    slice\_users3 = users[1:6:2]   # с 1 по 6 с шагом 2  print(slice\_users3)   # ["Bob", "Sam", "Bill"] |

# Кортежи

Для создания кортежа используются круглые скобки, в которые помещаются его значения, разделенные запятыми:

|  |  |
| --- | --- |
| 1  2 | user = ("Tom", 23)  print(user) |

Также для определения кортежа мы можем просто перечислить значения через запятую без применения скобок:

|  |  |
| --- | --- |
| 1  2 | user = "Tom", 23  print(user) |

Если вдруг кортеж состоит из одного элемента, то после единственного элемента кортежа необходимо поставить запятую:

|  |  |
| --- | --- |
| 1 | user = ("Tom",) |

Для создания кортежа из списка можно передать список в функцию **tuple()**, которая возвратит кортеж:

|  |  |
| --- | --- |
| 1  2  3 | users\_list = ["Tom", "Bob", "Kate"]  users\_tuple = tuple(users\_list)  print(users\_tuple)      # ("Tom", "Bob", "Kate") |

Для создания кортежа из списка можно передать список в функцию **tuple()**, которая возвратит кортеж:

|  |  |
| --- | --- |
| 1  2  3 | users\_list = ["Tom", "Bob", "Kate"]  users\_tuple = tuple(users\_list)  print(users\_tuple)      # ("Tom", "Bob", "Kate") |

Обращение к элементам в кортеже происходит также, как и в списке по индексу. Индексация начинается также с нуля при получении элементов с начала списка и с -1 при получении элементов с конца списка:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | users = ("Tom", "Bob", "Sam", "Kate")  print(users[0])     # Tom  print(users[2])     # Sam  print(users[-1])     # Kate    # получим часть кортежа со 2 элемента по 4  print(users[1:4])       # ("Bob", "Sam", "Kate") |

Но так как кортеж - неизменяемый тип (immutable), то мы не сможем изменить его элементы. То есть следующая запись работать не будет:

|  |  |
| --- | --- |
| 1 | users[1] = "Tim" |

При необходимости мы можем разложить кортеж на отдельные переменные:

|  |  |
| --- | --- |
| 1  2  3  4  5 | user = ("Tom", 22, False)  name, age, isMarried = user  print(name)             # Tom  print(age)              # 22  print(isMarried)        # False |

Особенно удобно использовать кортежи, когда необходимо возвратить из функции сразу несколько значений. Когда функция возвращает несколько значений, фактически она возвращает в кортеж:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | def get\_user():      name = "Tom"      age = 22      is\_married = False      return name, age, is\_married      user = get\_user()  print(user[0])              # Tom  print(user[1])              # 22  print(user[2])              # False |

# Словари

Определим пару словарей:

|  |  |
| --- | --- |
| 1  2  3 | users = {1: "Tom", 2: "Bob", 3: "Bill"}    elements = {"Au": "Золото", "Fe": "Железо", "H": "Водород", "O": "Кислород"} |

В словаре users в качестве ключей используются числа, а в качестве значений - строки. В словаре element в качестве ключей используются строки.

Но необязательно ключи и строки должны быть однотипными. Они могу представлять разные типы:

|  |  |
| --- | --- |
| 1 | objects = {1: "Tom", "2": True, 3: 100.6} |

Мы можем также вообще определить пустой словарь без элементов:

|  |  |
| --- | --- |
| 1 | objects = {} |

или так:

|  |  |
| --- | --- |
| 1 | objects = dict() |

### Преобразование из списка в словарь

Несмотря на то, что словарь и список - непохожие по структуре типы, но тем не менее существует возможности для отдельных видов списков преобразования их в словарь с помощью встроенной функции **dit()**. Для этого список должен хранить набор вложенных списков. Каждый вложенный список должен состоять из двух элементов - при конвертации в словарь первый элемент станет ключом, а второй - значением:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | users\_list = [      ["+111123455", "Tom"],      ["+384767557", "Bob"],      ["+958758767", "Alice"]  ]  users\_dict = dict(users\_list)  print(users\_dict)  # {"+111123455": "Tom", "+384767557": "Bob", "+958758767": "Alice"} |

### Получение и изменение элементов

Для доступа к элементам словаря необходимо использовать ключ:

|  |  |
| --- | --- |
| 1 | dictionary[ключ] |

Например, получим и изменим элементы в словаре:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | users = {      "+11111111": "Tom",      "+33333333": "Bob",      "+55555555": "Alice"  }    # получаем элемент с ключом "+11111111"  print(users["+11111111"])      # Tom    # установка значения элемента с ключом "+33333333"  users["+33333333"] = "Bob Smith"  print(users["+33333333"])      # Bob Smith |

Если при установки значения элемента с таким ключом в словаре не окажется, то произойдет его добавление:

|  |  |
| --- | --- |
| 1 | users["+4444444"] = "Sam" |

Но если мы попробуем получить значение с ключом, которого нет в словаре, то Python сгенерирует ошибку KeyError:

|  |  |
| --- | --- |
| 1 | user = users["+4444444"]    # KeyError |

И чтобы предупредить эту ситуацию перед обращением к элементу мы можем проверять наличие ключа в словаре с помощью выражения **ключ in словарь**. Если ключ имеется в словаре, то данное выражение возвращает True:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | key = "+4444444"  if key in users:      user = users[key]      print(user)  else:      print("Элемент не найден") |

Также для получения элементов можно использовать метод **get**, который имеет две формы:

* get(key): возвращает из словаря элемент с ключом key. Если элемента с таким ключом нет, то возвращает значение None
* get(key, default): возвращает из словаря элемент с ключом key. Если элемента с таким ключом нет, то возвращает значение по умолчанию default

|  |  |
| --- | --- |
| 1  2  3 | key = "+55555555"  user = users.get(key)  user = users.get(key, "Unknown user") |

### Удаление

Для удаления элемента по ключу применяется оператор **del**:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | users = {      "+11111111": "Tom",      "+33333333": "Bob",      "+55555555": "Alice"  }    del users["+55555555"]  print(users) |

Другой способ удаления представляет метод **pop()**. Он имеет две формы:

* pop(key): удаляет элемент по ключу key и возвращает удаленный элемент. Если элемент с данным ключом отсутствует, то генерируется исключение KeyError
* pop(key, default): удаляет элемент по ключу key и возвращает удаленный элемент. Если элемент с данным ключом отсутствует, то возвращается значение default

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | users = {      "+11111111": "Tom",      "+33333333": "Bob",      "+55555555": "Alice"  }  key = "+55555555"  user = users.pop(key)  print(user)    user = users.pop("+4444444", "Unknown user")  print(user) |

Если необходимо удалить все элементы, то в этом случае можно воспользоваться методом **clear():**

|  |  |
| --- | --- |
| 1 | users.clear() |

### Копирование и объединение словарей

Метод **copy()** копирует содержимое словаря, возвращая новый словарь:

|  |  |
| --- | --- |
| 1  2 | users = {"+1111111": "Tom","+3333333": "Bob","+5555555": "Alice"}  users2 = users.copy() |

Метод **update()** объединяет два словаря:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | users = {"+1111111": "Tom","+3333333": "Bob","+5555555": "Alice"}    users2 = {"+2222222": "Sam","+6666666": "Kate"}  users.update(users2)    print(users)    # {"+1111111": "Tom", "+3333333": "Bob", "+5555555": "Alice", "+2222222": "Sam", "+6666666": "Kate"}  print(users2)   # {"+2222222": "Sam", "+6666666": "Kate"} |

При этом словарь users2 остается без изменений. Изменяется словарь users, в который добавляются элементы другого словаря. Но если необходимо, чтобы оба исходных словаря были без изменений, а результатом объединения был какой-то третий словарь, то можно предварительно скопировать один словарь в другой:

|  |  |
| --- | --- |
| 1  2 | users3 = users.copy()  users3.update(users2) |

### Перебор словаря

Для перебора словаря можно воспользоваться циклом for:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | users = {      "+11111111": "Tom",      "+33333333": "Bob",      "+55555555": "Alice"  }  for key in users:      print(key, " - ", users[key]) |

При переборе элементов мы получаем ключ текущего элемента и по нему можем получить сам элемент.

Другой способ перебора элементов представляет использование метода **items()**:

|  |  |
| --- | --- |
| 1  2 | for key, value in users.items():      print(key, " - ", value) |

Метод items() возвращает набор кортежей. Каждый кортеж содержит ключ и значение элемента, которые при переборе мы тут же можем получить в переменные key и value.

Также существуют отдельно возможности перебора ключей и перебора значений. Для перебора ключей мы можем вызвать у словаря метод **keys()**:

|  |  |
| --- | --- |
| 1  2 | for key in users.keys():      print(key) |

Правда, этот способ перебора не имеет смысла, так как и без вызова метода keys() мы можем перебрать ключи, как было показано выше.

Для перебора только значений мы можем вызвать у словаря метод **values()**:

|  |  |
| --- | --- |
| 1  2 | for value in users.values():      print(value) |

### Комплексные словари

Кроме простейших объектов типа чисел и строк словари также могут хранить и более сложные объекты - те же списки, кортежи или другие словари:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | users = {      "Tom": {          "phone": "+971478745",          "email": "tom12@gmail.com"      },      "Bob": {          "phone": "+876390444",          "email": "bob@gmail.com",          "skype": "bob123"      }  } |

В данном случае значение каждого элемента словаря в свою очередь представляет отдельный словарь.

Для обращения к элементам вложенного словаря соответственно необходимо использовать два ключа:

|  |  |
| --- | --- |
| 1  2 | old\_email = users["Tom"]["email"]  users["Tom"]["email"] = "supertom@gmail.com" |

Но если мы попробуем получить значение по ключу, который отсутствует в словаре, Python сгенерирует исключение KeyError:

|  |  |
| --- | --- |
| 1 | tom\_skype = users["Tom"]["skype"]   # KeyError |

Чтобы избежать ошибки, можно проверять наличие ключа в словаре:

|  |  |
| --- | --- |
| 1  2  3  4  5 | key = "skype"  if key in users["Tom"]:      print(users["Tom"]["skype"])  else:      print("skype is not found") |

Во всем остальном работа с комплексными и вложенными словарями аналогична работе с обычными словарями.

# Множества

Множество (set) представляют еще один вид набора элементов. Для определения множества используются фигурные скобки, в которых перечисляются элементы:

|  |  |
| --- | --- |
| 1  2 | users = {"Tom","Bob","Alice", "Tom"}  print(users)    # {"Tom","Bob","Alice"} |

Обратите внимание, что несмотря на то, что функция print вывела один раз элемент "Tom", хотя в определении множества этот элемент содержится два раза. Все потому что множество содержит только **уникальные** значения.

Также для определения множества может применяться функция **set()**, в которую передается список или кортеж элементов:

|  |  |
| --- | --- |
| 1 | users3 = set(["Mike", "Bill", "Ted"]) |

Функцию set удобно применять для создания пустого множества:

|  |  |
| --- | --- |
| 1 | users = set() |

Для получения длины множества применяется встроенная функция **len()**:

|  |  |
| --- | --- |
| 1  2 | users = {"Tom","Bob","Alice"}  print(len(users)}   # 3 |

### Добавление элементов

Для добавления одиночного элемента вызывается метод **add()**:

|  |  |
| --- | --- |
| 1  2  3 | users = set()  users.add("Sam")  print(users) |

### Удаление элементов

Для удаления одного элемента вызывается метод **remove()**, в который передается удаляемый элемент. Но следует учитывать, что если такого элемента не окажется в множестве, то будет сгенерирована ошибка. Поэтому перед удалением следует проверять на наличие элемента с помощью оператора **in**:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | users = {"Tom", "Bob", "Alice"}    user = "Tom"  if user in users:      users.remove(user)  print(users)    # {"Bob", "Alice"} |

Также для удаления можно использовать метод **discard()**, который не будет генерировать исключения при отсутствии элемента:

|  |  |
| --- | --- |
| 1  2 | user = "Tim"  users.discard(user) |

Для удаления всех элементов вызывается метод **clear()**:

|  |  |
| --- | --- |
| 1 | users.clear() |

### Перебор множества

Для перебора элементов можно использовать цикл for:

|  |  |
| --- | --- |
| 1  2  3  4 | users = {"Tom","Bob","Alice"}    for user in users:      print(user) |

При переборе каждый элемент помещается в переменную user.

### Операции с множествами

С помощью метода **copy()** можно скопировать содержимое одного множества в другую переменную:

|  |  |
| --- | --- |
| 1  2 | users = {"Tom","Bob","Alice"}  users3 = users.copy() |

Метод **union()** объединяет два множества и возвращает новое множество:

|  |  |
| --- | --- |
| 1  2  3  4  5 | users = {"Tom","Bob","Alice"}  users2 = {"Sam","Kate", "Bob"}    users3 = users.union(users2)  print(users3)   # {"Bob", "Alice", "Sam", "Kate", "Tom"} |

Пересечение множеств позволяет получить только те элементы, которые есть одновременно в обоих множествах. Метод **intersection()**производит операцию пересечения множеств и возвращает новое множество:

|  |  |
| --- | --- |
| 1  2  3  4  5 | users = {"Tom","Bob","Alice"}  users2 = {"Sam","Kate", "Bob"}    users3 = users.intersection(users2)  print(users3)   # {"Bob"} |

Вместо метода intersection мы могли бы использовать операцию логического умножения:

|  |  |
| --- | --- |
| 1  2  3  4 | users = {"Tom","Bob","Alice"}  users2 = {"Sam","Kate", "Bob"}    print(users & users2)   # {"Bob"} |

Еще одна операция - разность множеств возвращает те элементы, которые есть в первом множестве, но отсутствуют во втором. Для получения разности множеств можно использовать метод **difference** или операцию вычитания:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | users = {"Tom","Bob","Alice"}  users2 = {"Sam","Kate", "Bob"}    users3 = users.difference(users2)  print(users3)           # {"Tom", "Alice"}  print(users - users2)   # {"Tom", "Alice"} |

### Отношения между множествами

Метод **issubset** позволяет выяснить, является ли текущее множество подмножеством (то есть частью) другого множества:

|  |  |
| --- | --- |
| 1  2  3  4  5 | users = {"Tom", "Bob", "Alice"}  superusers = {"Sam", "Tom", "Bob", "Alice", "Greg"}    print(users.issubset(superusers))   # True  print(superusers.issubset(users))   # False |

Метод **issuperset**, наоборот, возвращает True, если текущее множество является надмножеством (то есть содержит) для другого множества:

|  |  |
| --- | --- |
| 1  2  3  4  5 | users = {"Tom", "Bob", "Alice"}  superusers = {"Sam", "Tom", "Bob", "Alice", "Greg"}    print(users.issuperset(superusers))   # False  print(superusers.issuperset(users))   # True |

# Работа с файлами

* **r** (Read). Файл открывается для чтения. Если файл не найден, то генерируется исключение FileNotFoundError
* **w** (Write). Файл открывается для записи. Если файл отсутствует, то он создается. Если подобный файл уже есть, то он создается заново, и соответственно старые данные в нем стираются.
* **a** (Append). Файл открывается для дозаписи. Если файл отсутствует, то он создается. Если подобный файл уже есть, то данные записываются в его конец.
* **b** (Binary). Используется для работы с бинарными файлами. Применяется вместе с другими режимами - w или r.

После завершения работы с файлом его обязательно нужно закрыть методом close(). Данный метод освободит все связанные с файлом используемые ресурсы.

Например, откроем для записи текстовый файл "hello.txt":

|  |  |
| --- | --- |
| 1  2  3 | myfile = open("hello.txt", "w")    myfile.close() |

При открытии файла или в процессе работы с ним мы можем столкнуться с различными исключениями, например, к нему нет доступа и т.д. В этом случае программа выпадет в ошибку, а ее выполнение не дойдет до вызова метода close, и соответственно файл не будет закрыт.

В этом случае мы можем обрабатывать исключения:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | try:      somefile = open("hello.txt", "w")      try:          somefile.write("hello world")      except Exception as e:          print(e)      finally:          somefile.close()  except Exception as ex:      print(ex) |

В данном случае вся работа с файлом идет во вложенном блоке try. И если вдруг возникнет какое-либо исключение, то в любом случае в блоке finally файл будет закрыт.

Однако есть и более удобная конструкция - конструкция **with**:

|  |  |
| --- | --- |
| 1  2 | with open(file, mode) as file\_obj:      инструкции |

Эта конструкция определяет для открытого файла переменную file\_obj и выполняет набор инструкций. После их выполнения файл автоматически закрывается. Даже если при выполнении инструкций в блоке with возникнут какие-либо исключения, то файл все равно закрывается.

Так, перепишем предыдущий пример:

|  |  |
| --- | --- |
| 1  2 | with open("hello.txt", "w") as somefile:      somefile.write("hello world") |

### Запись в текстовый файл

Чтобы открыть текстовый файл на запись, необходимо применить режим w (перезапись) или a (дозапись). Затем для записи применяется метод **write(str)**, в который передается записываемая строка. Стоит отметить, что записывается именно строка, поэтому, если нужно записать числа, данные других типов, то их предварительно нужно конвертировать в строку.

Запишем некоторую информацию в файл "hello.txt":

|  |  |
| --- | --- |
| 1  2 | with open("hello.txt", "w") as file:      file.write("hello world") |

Если мы откроем папку, в которой находится текущий скрипт Python, то увидем там файл hello.txt. Этот файл можно открыть в любом текстовом редакторе и при желании изменить.

Теперь дозапишем в этот файл еще одну строку:

|  |  |
| --- | --- |
| 1  2 | with open("hello.txt", "a") as file:      file.write("\ngood bye, world") |

### Чтение файла

Для чтения файла он открывается с режимом r (Read), и затем мы можем считать его содержимое различными методами:

* **readline()**: считывает одну строку из файла
* **read()**: считывает все содержимое файла в одну строку
* **readlines()**: считывает все строки файла в список

Например, считаем выше записанный файл построчно:

|  |  |
| --- | --- |
| 1  2  3 | with open("hello.txt", "r") as file:      for line in file:          print(line, end="") |

Несмотря на то, что мы явно не применяем метод readline() для чтения каждой строки, но в при переборе файла этот метод автоматически вызывается для получения каждой новой строки. Поэтому в цикле вручную нет смысла вызывать метод readline. И поскольку строки разделяются символом перевода строки "\n", то чтобы исключить излишнего переноса на другую строку в функцию print передается значение end="".

Теперь явным образом вызовем метод readline() для чтения отдельных строк:

|  |  |
| --- | --- |
| 1  2  3  4  5 | with open("hello.txt", "r") as file:      str1 = file.readline()      print(str1, end="")      str2 = file.readline()      print(str2) |

Консольный вывод:

hello world

good bye, world

Метод readline можно использовать для построчного считывания файла в цикле while:

|  |  |
| --- | --- |
| 1  2  3  4  5 | with open("hello.txt", "r") as file:      line = file.readline()      while line:          print(line, end="")          line = file.readline() |

Если файл небольшой, то его можно разом считать с помощью метода **read()**:

|  |  |
| --- | --- |
| 1  2  3 | with open("hello.txt", "r") as file:      content = file.read()      print(content) |

И также применим метод **readlines()** для считывания всего файла в список строк:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | with open("hello.txt", "r") as file:      contents = file.readlines()      str1 = contents[0]      str2 = contents[1]      print(str1, end="")      print(str2) |

При чтении файла мы можем столкнуться с тем, что его кодировка не совпадает с ASCII. В этом случае мы явным образом можем указать кодировку с помощью паоаметра **encoding**:

|  |  |
| --- | --- |
| 1  2  3 | filename = "hello.txt"  with open(filename, encoding="utf8") as file:      text = file.read() |

### CSV-файл

Рассмотрим работу модуля на примере:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | import csv    FILENAME = "users.csv"    users = [      ["Tom", 28],      ["Alice", 23],      ["Bob", 34]  ]    with open(FILENAME, "w", newline="") as file:      writer = csv.writer(file)      writer.writerows(users)      with open(FILENAME, "a", newline="") as file:      user = ["Sam", 31]      writer = csv.writer(file)      writer.writerow(user) |

В файл записывается двухмерный список - фактически таблица, где каждая строка представляет одного пользователя. А каждый пользователь содержит два поля - имя и возраст. То есть фактически таблица из трех строк и двух столбцов.

При открытии файла на запись в качестве третьего параметра указывается значение newline="" - пустая строка позволяет корректно считывать строки из файла вне зависимости от операционной системы.

Для записи нам надо получить объект **writer**, который возвращается функцией csv.writer(file). В эту функцию передается открытый файл. А собственно запись производится с помощью метода writer.writerows(users) Этот метод принимает набор строк. В нашем случае это двухмерный список.

Если необходимо добавить одну запись, которая представляет собой одномерный список, например, ["Sam", 31], то в этом случае можно вызвать метод **writer.writerow(user)**

В итоге после выполнения скрипта в той же папке окажется файл users.csv, который будет иметь следующее содержимое:

|  |  |
| --- | --- |
| 1  2  3  4 | Tom,28  Alice,23  Bob,34  Sam,31 |

Для чтения из файла нам наоборот нужно создать объект **reader**:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | import csv    FILENAME = "users.csv"    with open(FILENAME, "r", newline="") as file:      reader = csv.reader(file)      for row in reader:          print(row[0], " - ", row[1]) |

При получении объекта reader мы можем в цикле перебрать все его строки:

Tom - 28

Alice - 23

Bob - 34

Sam - 31

### Работа со словарями

В примере выше каждая запись или строка представляла собой отдельный список, например, ["Sam", 31]. Но кроме того, модуль csv имеет специальные дополнительные возможности для работы со словарями. В частности, функция **csv.DictWriter()** возвращает объект writer, который позволяет записывать в файл. А функция **csv.DictReader()** возвращает объект reader для чтения из файла. Например:

|  |  |
| --- | --- |
| 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 | import csv    FILENAME = "users.csv"    users = [      {"name": "Tom", "age": 28},      {"name": "Alice", "age": 23},      {"name": "Bob", "age": 34}  ]    with open(FILENAME, "w", newline="") as file:      columns = ["name", "age"]      writer = csv.DictWriter(file, fieldnames=columns)      writer.writeheader()        # запись нескольких строк      writer.writerows(users)        user = {"name" : "Sam", "age": 41}      # запись одной строки      writer.writerow(user)    with open(FILENAME, "r", newline="") as file:      reader = csv.DictReader(file)      for row in reader:          print(row["name"], "-", row["age"]) |

Запись строк также производится с помощью методов writerow() и writerows(). Но теперь каждая строка представляет собой отдельный словарь, и кроме того, производится запись и заголовков столбцов с помощью метода **writeheader()**, а в метод cvs.DictWriter в качестве второго параметра передается набор столбцов.

Бинарные файлы

Бинарные файлы в отличие от текстовых хранят информацию в виде набора байт. Для работы с ними в Python необходим встроенный модуль **pickle**. Этот модуль предоставляет два метода:

* **dump(obj, file)**: записывает объект obj в бинарный файл file
* **load(file)**: считывает данные из бинарного файла в объект

При открытии бинарного файла на чтение или запись также надо учитывать, что нам нужно применять режим "b" в дополнение к режиму записи ("w") или чтения ("r"). Допустим, надо надо сохранить два объекта:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | import pickle    FILENAME = "user.dat"    name = "Tom"  age = 19    with open(FILENAME, "wb") as file:      pickle.dump(name, file)      pickle.dump(age, file)    with open(FILENAME, "rb") as file:      name = pickle.load(file)      age = pickle.load(file)      print("Имя:", name, "\tВозраст:", age) |

С помощью функции dump последовательно записываются два объекта. Поэтому при чтении файла также последовательно посредством функции load мы можем считать эти объекты. Консольный вывод программы:

Имя: Tom Возраст: 28

Подобным образом мы можем сохранять и извлекать из файла наборы объектов:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | import pickle    FILENAME = "users.dat"    users = [      ["Tom", 28, True],      ["Alice", 23, False],      ["Bob", 34, False]  ]    with open(FILENAME, "wb") as file:      pickle.dump(users, file)      with open(FILENAME, "rb") as file:      users\_from\_file = pickle.load(file)      for user in users\_from\_file:          print("Имя:", user[0], "\tВозраст:", user[1], "\tЖенат(замужем):", user[2]) |

В зависимости от того, какой объект мы записывали функцией dump, тот же объект будет возвращен функцией load при считывании файла.

Консольный вывод:

Имя: Tom Возраст: 28 Женат(замужем): True

Имя: Alice Возраст: 23 Женат(замужем): False

Имя: Bob Возраст: 34 Женат(замужем): False

# OS Module

Ряд возможностей по работе с каталогами и файлами предоставляет встроенный модуль **os**. Хотя он содержит много функций, рассмотрим только основные из них:

* **mkdir()**: создает новую папку
* **rmdir()**: удаляет папку
* **rename()**: переименовывает файл
* **remove()**: удаляет файл

### Создание и удаление папки

Для создания папки применяется функция **mkdir()**, в которую передается путь к создаваемой папке:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | import os    # путь относительно текущего скрипта  os.mkdir("hello")  # абсолютный путь  os.mkdir("[c://somedir](file:///C:\somedir)")  os.mkdir("[c://somedir/hello](file:///C:\somedir\hello)") |

Для удаления папки используется функция **rmdir()**, в которую передается путь к удаляемой папке:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | import os    # путь относительно текущего скрипта  os.rmdir("hello")  # абсолютный путь  os.rmdir("[c://somedir/hello](file:///C:\somedir\hello)") |

### Переименование файла

Для переименования вызывается функция **rename(source, target)**, первый параметр которой - путь к исходному файлу, а второй - новое имя файла. В качестве путей могут использоваться как абсолютные, так и относительные. Например, пусть в папке *C://SomeDir/*располагается файл *somefile.txt*. Переименуем его в файл "hello.txt":

|  |  |
| --- | --- |
| 1  2  3 | import os    os.rename("[C://SomeDir/somefile.txt](file:///C:\SomeDir\somefile.txt)", "[C://SomeDir/hello.txt](file:///C:\SomeDir\hello.txt)") |

### Удаление файла

Для удаления вызывается функция **remove()**, в которую передается путь к файлу:

|  |  |
| --- | --- |
| 1  2  3 | import os    os.remove("[C://SomeDir/hello.txt](file:///C:\SomeDir\hello.txt)") |

### Существование файла

Если мы попытаемся открыть файл, который не существует, то Python выбросит исключение FileNotFoundError. Для отлова исключения мы можем использовать конструкцию try...except. Однако можно уже до открытия файла проверить, существует ли он или нет с помощью метода **os.path.exists(path)**. В этот метод передается путь, который необходимо проверить:

|  |  |
| --- | --- |
| 1  2  3  4  5 | filename = input("Введите путь к файлу: ")  if os.path.exists(filename):      print("Указанный файл существует")  else:      print("Файл не существует") |

# Строки

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.

### 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
* string[start:end]: извлекается последовательность символов начиная с индекса start по индекс end
* string[start:end:step]: извлекается последовательность символов начиная с индекса start по индекс end 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 символа  sub\_string1 = string[:5]  print(sub\_string1)      # hello    # со 2 до 5 символа  sub\_string2 = string[2:5]  print(sub\_string2)      # llo    # со 2 по 9 символ через один символ  sub\_string3 = string[2:9:2]  print(sub\_string3)      # lowr |

### 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 |

### Перебор строки

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

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

# Методы строк

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

* **isalpha(str)**: возвращает True, если строка состоит только из алфавитных символов
* **islower(str)**: возвращает True, если строка состоит только из символов в нижнем регистре
* **isuppper(str)**: возвращает True, если все символы строки в верхнем регистре
* **isdigit(str)**: возвращает True, если все символы строки - цифры
* **isnumeric(str)**: возвращает True, если строка представляет собой число
* **startwith(str)**: возвращает True, если строка начинается с подстроки str
* **endwith(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

# Форматирование

### 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 |

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 |

Для вывода процентов лучше воспользоваться кодом "%":

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

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

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

### Форматирование без метода format

В принципе для форматирования значений мы можем обойтись и без метода format, применяя следующий синтаксис:

|  |  |
| --- | --- |
| 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 |

# 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, stop): 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  print(number)  number = random.randrange(2, 10)  # значение в диапазоне 2, 3, 4, 5, 6, 7, 8, 9, 10  print(number)  number = random.randrange(2, 10, 2)  # значение в диапазоне 2, 4, 6, 8, 10  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)     # 1  random\_number = random.choice(numbers)  print(random\_number) |

# Math

* **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:

|  |  |
| --- | --- |
|  | 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)    # возведение в квадрат  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 |

# 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 станы 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(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("%f", number)  print(formatted)    # 12345,678900    formatted = locale.format("%.2f", number)  print(formatted)    # 12345,68    formatted = locale.format("%d", number)  print(formatted)    # 12345    formatted = locale.format("%e", number)  print(formatted)    # 1,234568e+04    money = 234.678  formatted = locale.currency(money)  print(formatted)    # 234,68 € |

# 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 |

# Clasess

|  |
| --- |
| class Person:      name = "Tom"        def display\_info(self):          print("Привет, меня зовут", self.name)    person1 = Person()  person1.display\_info()         # Привет, меня зовут Tom    person2 = Person()  person2.name = "Sam"  person2.display\_info()         # Привет, меня зовут Sam |

Класс Person определяет атрибут name, который хранит имя человека, и метод display\_info, с помощью которого выводится информация о человеке.

При определении методов любого класса следует учитывать, что все они должны принимать в качестве первого параметра ссылку на текущий объект, который согласно условностям называется **self** (в ряде языков программирования есть своего рода аналог - ключевое слово this). Через эту ссылку внутри класса мы можем обратиться к методам или атрибутам этого же класса. В частности, через выражение self.name можно получить имя пользователя.

Однако мы можем явным образом определить в классах конструктор с помощью специального метода, который называется **\_\_init()**. К примеру, изменим класс Person, добавив в него конструктор:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | class Person:        # конструктор      def \_\_init\_\_(self, name):          self.name = name  # устанавливаем имя        def display\_info(self):          print("Привет, меня зовут", self.name)      person1 = Person("Tom")  person1.display\_info()         # Привет, меня зовут Tom  person2 = Person("Sam")  person2.display\_info()         # Привет, меня зовут Sam |

В файле *classes.py* определим два класса:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | class Person:        # конструктор      def \_\_init\_\_(self, name):          self.name = name  # устанавливаем имя        def display\_info(self):          print("Привет, меня зовут", self.name)      class Auto:      def \_\_init\_\_(self, name):          self.name = name        def move(self, speed):          print(self.name, "едет со скоростью", speed, "км/ч") |

В дополнение к классу Person здесь также определен класс Auto, который представляет машину и который имеет метод move и атрибут name. Подключим эти классы и используем их в скрипте *main.py*:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | from classes import Person, Auto    tom = Person("Tom")  tom.display\_info()    bmw = Auto("BMW")  bmw.move(65) |

# Инкапсуляция

По умолчанию атрибуты в классах являются общедоступными, а это значит, что из любого места программы мы можем получить атрибут объекта и изменить его. Например:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | class Person:      def \_\_init\_\_(self, name, age):          self.name = name    # устанавливаем имя          self.age = age      # устанавливаем возраст        def display\_info(self):          print("Имя:", self.name, "\tВозраст:", self.age)      tom = Person("Tom", 23)  tom.name = "Человек-паук"       # изменяем атрибут name  tom.age = -129                  # изменяем атрибут age  tom.display\_info()              # Имя: Человек-паук     Возраст: -129 |

Но в данном случае мы можем, к примеру, присвоить возрасту или имени человека некорректное значение, например, указать отрицательный возраст. Подобное поведение нежелательно, поэтому встает вопрос о контроле за доступом к атрибутам объекта.

С данной проблемой тесно связано понятие инкапсуляции. **Инкапсуляция** является фундаментальной концепцией объектно-ориентированного программирования. Она предотвращает прямой доступ к атрибутам объект из вызывающего кода.

Касательно инкапсуляции непосредственно в языке программирования Python скрыть атрибуты класса можно сделав их приватными или закрытыми и ограничив доступ к ним через специальные методы, которые еще называются **свойствами**.

Изменим выше определенный класс, определив в нем свойства:

|  |  |
| --- | --- |
| 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 | class Person:      def \_\_init\_\_(self, name, age):          self.\_\_name = name    # устанавливаем имя          self.\_\_age = age      # устанавливаем возраст        def set\_age(self, age):          if age in range(1, 100):              self.\_\_age = age          else:              print("Недопустимый возраст")        def get\_age(self):          return self.\_\_age        def get\_name(self):          return self.\_\_name        def display\_info(self):          print("Имя:", self.\_\_name, "\tВозраст:", self.\_\_age)    tom = Person("Tom", 23)    tom.\_\_age = 43              # Атрибут age не изменится  tom.display\_info()          # Имя: Tom  Возраст: 23  tom.set\_age(-3486)          # Недопустимый возраст  tom.set\_age(25)  tom.display\_info()          # Имя: Tom  Возраст: 25 |

Для создания приватного атрибута в начале его наименования ставится двойной прочерк: self.\_\_name. К такому атрибуту мы сможем обратиться только из того же класса. Но не сможем обратиться вне этого класса. Например, присвоение значения этому атрибуту ничего не даст:

|  |  |
| --- | --- |
| 1 | tom.\_\_age = 43 |

А попытка получить его значение приведет к ошибке выполнения:

|  |  |
| --- | --- |
| 1 | print(tom.\_\_age) |

Однако все же нам может потребоваться устанавливать возраст пользователя из вне. Для этого создаются свойства. Используя одно свойство, мы можем получить значение атрибута:

|  |  |
| --- | --- |
| 1  2 | def get\_age(self):      return self.\_\_age |

Данный метод еще часто называют геттер или аксессор.

Для изменения возраста определено другое свойство:

|  |  |
| --- | --- |
| 1  2  3  4  5 | def set\_age(self, value):      if value in range(1, 100):          self.\_\_age = value      else:          print("Недопустимый возраст") |

Здесь мы уже можем решить в зависимости от условий, надо ли переустанавливать возраст. Данный метод еще называют сеттер или мьютейтор (mutator).

### Аннотации свойств

Выше мы рассмотрели, как создавать свойства. Но Python имеет также еще один - более элегантный способ определения свойств. Этот способ предполагает использование аннотаций, которые предваряются символом @.

Для создания свойства-геттера над свойством ставится аннотация **@property**.

Для создания свойства-сеттера над свойством устанавливается аннотация **имя\_свойства\_геттера.setter**.

Перепишем класс Person с использованием аннотаций:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | class Person:      def \_\_init\_\_(self, name, age):          self.\_\_name = name    # устанавливаем имя          self.\_\_age = age      # устанавливаем возраст        @property      def age(self):          return self.\_\_age        @age.setter      def age(self, age):          if age in range(1, 100):              self.\_\_age = age          else:              print("Недопустимый возраст")        @property      def name(self):          return self.\_\_name        def display\_info(self): |

Наследование

class Person:

    def \_\_init\_\_(self, name, age):

        self.\_\_name = name  # устанавливаем имя

        self.\_\_age = age  # устанавливаем возраст

    @property

    def age(self):

        return self.\_\_age

    @age.setter

    def age(self, age):

        if age in range(1, 100):

            self.\_\_age = age

        else:

            print("Недопустимый возраст")

    @property

    def name(self):

        return self.\_\_name

    def display\_info(self):

        print("Имя:", self.\_\_name, "\tВозраст:", self.\_\_age)

class Employee(Person):

    def details(self, company):

        # print(self.\_\_name, "работает в компании", company) # так нельзя, self.\_\_name - приватный атрибут

        print(self.name, "работает в компании", company)

tom = Employee("Tom", 23)

tom.details("Google")

tom.age = 33

tom.display\_info()

# Полиморфизм

**Полиморфизм** является еще одним базовым аспектом объектно-ориентированного программирования и предполагает способность к изменению функционала, унаследованного от базового класса.

Например, пусть у нас будет следующая иерархия классов:

|  |  |
| --- | --- |
| 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  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51 | class Person:      def \_\_init\_\_(self, name, age):          self.\_\_name = name  # устанавливаем имя          self.\_\_age = age  # устанавливаем возраст        @property      def name(self):          return self.\_\_name        @property      def age(self):          return self.\_\_age        @age.setter      def age(self, age):          if age in range(1, 100):              self.\_\_age = age          else:              print("Недопустимый возраст")        def display\_info(self):          print("Имя:", self.\_\_name, "\tВозраст:", self.\_\_age)      class Employee(Person):      # определение конструктора      def \_\_init\_\_(self, name, age, company):          Person.\_\_init\_\_(self, name, age)          self.company = company        # переопределение метода display\_info      def display\_info(self):          Person.display\_info(self)          print("Компания:", self.company)      class Student(Person):      # определение конструктора      def \_\_init\_\_(self, name, age, university):          Person.\_\_init\_\_(self, name, age)          self.university = university        # переопределение метода display\_info      def display\_info(self):          print("Студент", self.name, "учится в университете", self.university)    people = [Person("Tom", 23), Student("Bob", 19, "Harvard"), Employee("Sam", 35, "Google")]    for person in people:      person.display\_info()      print() |

### Проверка типа объекта

При работе с объектами бывает необходимо в зависимости от их типа выполнить те или иные операции. И с помощью встроенной функции **isinstance()** мы можем проверить тип объекта. Эта функция принимает два параметра:

|  |  |
| --- | --- |
| 1 | isinstance(object, type) |

Первый параметр представляет объект, а второй - тип, на принадлежность к которому выполняется проверка. Если объект представляет указанный тип, то функция возвращает True. Например, возьмем выше описанную иерархию классов:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | for person in people:      if isinstance(person, Student):          print(person.university)      elif isinstance(person, Employee):          print(person.company)      else:          print(person.name)      print() |

# Строковое представление объекта

class Person:

    def \_\_init\_\_(self, name, age):

        self.\_\_name = name  # устанавливаем имя

        self.\_\_age = age  # устанавливаем возраст

    @property

    def name(self):

        return self.\_\_name

    @property

    def age(self):

        return self.\_\_age

    @age.setter

    def age(self, age):

        if age in range(1, 100):

            self.\_\_age = age

        else:

            print("Недопустимый возраст")

    def display\_info(self):

        print(self.\_\_str\_\_())

    def \_\_str\_\_(self):

        return "Имя: {} \t Возраст: {}".format(self.\_\_name, self.\_\_age)

tom = Person("Tom", 23)

print(tom)

# Работа с датами и временем

Основной функционал для работы с датами и временем сосредоточен в модуле **datetime** в виде следующих классов:

* **date**
* **time**
* **datetime**

### Класс date

Для работы с датами воспользуемся классом **date**, который определен в модуле datetime. Для создания объекта date мы можем использовать конструктор date, который последовательно принимает три параметра: год, месяц и день.

|  |  |
| --- | --- |
| 1 | date(year, month, day) |

Например, создадим какую-либо дату:

|  |  |
| --- | --- |
| 1  2  3  4 | import datetime    yesterday = datetime.date(2017,5, 2)  print(yesterday)      # 2017-05-02 |

Если необходимо получить текущую дату, то можно воспользоваться методом **today()**:

|  |  |
| --- | --- |
| 1  2  3  4  5 | from datetime import date    today = date.today()  print(today)      # 2017-05-03  print("{}.{}.{}".format(today.day, today.month, today.year))      # 2.5.2017 |

С помощью свойств day, month, year можно получить соответственно день, месяц и год

### Класс time

За работу с временем отвечает класс **time**. Используя его конструктор, можно создать объект времени:

|  |  |
| --- | --- |
| 1 | time([hour] [, min] [, sec] [, microsec]) |

Конструктор последовательно принимает часы, минуты, секунды и микросекунды. Все параметры необязательные, и если мы какой-то параметр не передадим, то соответствующее значение будет инициализироваться нулем.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | from datetime import time    current\_time = time()  print(current\_time)     # 00:00:00    current\_time = time(16, 25)  print(current\_time)     # 16:25:00    current\_time = time(16, 25, 45)  print(current\_time)     # 16:25:45 |

### Класс datetime

Класс **datetime** из одноименного модуля объединяет возможности работы с датой и временем. Для создания объекта datetime можно использовать следующий конструктор:

|  |  |
| --- | --- |
| 1 | datetime(year, month, day [, hour] [, min] [, sec] [, microsec]) |

Первые три параметра, представляющие год, месяц и день, являются обязательными. Остальные необязательные, и если мы не укажем для них значения, то по умолчанию они инициализируются нулем.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | from datetime import datetime    deadline = datetime(2017, 5, 10)  print(deadline)     # 2017-05-10 00:00:00    deadline = datetime(2017, 5, 10, 4, 30)  print(deadline)     # 2017-05-10 04:30:00 |

Для получения текущих даты и времени можно вызвать метод **now()**:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | from datetime import datetime    now = datetime.now()  print(now)     # 2017-05-03 11:18:56.239443    print("{}.{}.{}  {}:{}".format(now.day, now.month, now.year, now.hour, now.minute))  # 3.5.2017  11:21    print(now.date())  print(now.time()) |

С помощью свойств day, month, year, hour, minute, second можно получить отдельные значения даты и времени. А через методы date()и time() можно получить отдельно дату и время соответственно.

### Преобразование из строки в дату

Из функциональности класса datetime следует отметить метод **strptime()**, который позволяет распарсить строку и преобразовать ее в дату. Этот метод принимает два параметра:

|  |  |
| --- | --- |
| 1 | strptime(str, format) |

Первый параметр str представляет строковое определение даты и времени, а второй параметр - формат, который определяет, как различные части даты и времени расположены в этой строке.

Для определения формата мы можем использовать следующие коды:

* **%d**: день месяца в виде числа
* **%m**: порядковый номер месяца
* **%y**: год в виде 2-х чисел
* **%Y**: год в виде 4-х чисел
* **%H**: час в 24-х часовом формате
* **%M**: минута
* **%S**: секунда

Применим различные форматы:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | from datetime import datetime  deadline = datetime.strptime("22/05/2017", "%d/%m/%Y")  print(deadline)     # 2017-05-22 00:00:00    deadline = datetime.strptime("22/05/2017 12:30", "%d/%m/%Y %H:%M")  print(deadline)     # 2017-05-22 12:30:00    deadline = datetime.strptime("05-22-2017 12:30", "%m-%d-%Y %H:%M")  print(deadline)     # 2017-05-22 12:30:00 |

# Операции с датами

### Фоматирование дат и времени

Для форматирования объектов date и time в обоих этих классах предусмотрен метод **strftime(format)**. Этот метод принимает только один параметр, указывающий на формат, в который нужно преобразовать дату или время.

Для определения формата мы можем использовать один из следующих кодов форматирования:

* **%a**: аббревиатура дня недели. Например, Wed - от слова Wednesday (по умолчанию используются английские наименования)
* **%A**: день недели полностью, например, Wednesday
* **%b**: аббревиатура названия месяца. Например, Oct (сокращение от October)
* **%B**: название месяца полностью, например, October
* **%d**: день месяца, дополненный нулем, например, 01
* **%m**: номер месяца, дополненный нулем, например, 05
* **%y**: год в виде 2-х чисел
* **%Y**: год в виде 4-х чисел
* **%H**: час в 24-х часовом формате, например, 13
* **%I**: час в 12-ти часовом формате, например, 01
* **%M**: минута
* **%S**: секунда
* **%f**: микросекунда
* **%p**: указатель AM/PM
* **%c**: дата и время, отформатированные под текущую локаль
* **%x**: дата, отформатированная под текущую локаль
* **%X**: время, форматированное под текущую локаль

Используем различные форматы:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | from datetime import datetime  now = datetime.now()  print(now.strftime("%Y-%m-%d"))             # 2017-05-03  print(now.strftime("%d/%m/%Y"))             # 03/05/2017  print(now.strftime("%d/%m/%y"))             # 03/05/17  print(now.strftime("%d %B %Y (%A)"))        # 03 May 2017 (Wednesday)  print(now.strftime("%d/%m/%y %I:%M"))       # 03/05/17 01:36 |

При выводе названий месяцев и дней недели по умолчанию используются английские наименования. Если мы хотим использовать текущую локаль, но то мы можем ее предварительно установить с помощью модуля locale:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | from datetime import datetime  import locale  locale.setlocale(locale.LC\_ALL, "")    now = datetime.now()  print(now.strftime("%d %B %Y (%A)"))        # 03 Май 2017 (среда) |

### Сложение и вычитани дат и времени

Нередко при работе с датами возникает необходимость добавить к какой-либо дате определенный промежуток времени или, наоборот, вычесть некоторый период. И специально для таких операций в модуле datetime определен класс **timedelta**. Фактически этот класс определяет некоторый период времени.

Для определения промежутка времени можно использовать конструктор timedelta:

|  |  |
| --- | --- |
| 1 | timedelta([days] [, seconds] [, microseconds] [, milliseconds] [, minutes] [, hours] [, weeks]) |

В конструктор мы последовательно передаем дни, секунды, микросекунды, миллисекунды, минуты, часы и недели.

Определим несколько периодов:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | from datetime import timedelta    three\_hours = timedelta(hours=3)  print(three\_hours)       # 3:00:00  three\_hours\_thirty\_minutes = timedelta(hours=3, minutes=30)  # 3:30:00    two\_days = timedelta(2)  # 2 days, 0:00:00    two\_days\_three\_hours\_thirty\_minutes = timedelta(days=2, hours=3, minutes=30)  # 2 days, 3:30:00 |

Используя timedelta, мы можем складывать или вычитать даты. Например, получим дату, которая будет через два дня:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | from datetime import timedelta, datetime    now = datetime.now()  print(now)                      # 2017-05-03 17:46:44.558754  two\_days = timedelta(2)  in\_two\_days = now + two\_days  print(in\_two\_days)              # 2017-05-05 17:46:44.558754 |

Или узнаем, сколько было времени 10 часов 15 минут назад, то есть фактически нам надо вычесть из текущего времени 10 часов и 15 минут:

|  |  |
| --- | --- |
| 1  2  3  4  5 | from datetime import timedelta, datetime    now = datetime.now()  till\_ten\_hours\_fifteen\_minutes = now - timedelta(hours=10, minutes=15)  print(till\_ten\_hours\_fifteen\_minutes) |

### Свойства timedelta

Класс timedelta имеет несколько свойств, с помощью которых мы можем получить временной промежуток:

* **days**: возвращает количество дней
* **seconds**: возвращает количество секунд
* **microseconds**: возвращает количество микросекунд

Кроме того, метод **total\_seconds()** возвращает общее количество секунд, куда входят и дни, и собственно секунды, и микросекунды.

Например, узнаем какой временной период между двумя датами:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | from datetime import timedelta, datetime    now = datetime.now()  twenty\_two\_may = datetime(2017, 5, 22)  period = twenty\_two\_may - now  print("{} дней  {} секунд   {} микросекунд".format(period.days, period.seconds, period.microseconds))  # 18 дней  17537 секунд   72765 микросекунд    print("Всего: {} секунд".format(period.total\_seconds()))  # Всего: 1572737.072765 секунд |

### Сравнение дат

Также как и строки и числа, даты можно сравнивать с помощью стандартных операторов сравнения:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | from datetime import datetime    now = datetime.now()  deadline = datetime(2017, 5, 22)  if now > deadline:      print("Срок сдачи проекта прошел")  elif now.day == deadline.day and now.month == deadline.month and now.year == deadline.year:      print("Срок сдачи проекта сегодня")  else:      period = deadline - now      print("Осталось {} дней".format(period.days)) |

# Window

Многие программы на сегодняшний день используют графический интерфейс, который более интуитивен и удобен для пользователя, чем консоль. И с помощью языка программирования Python также можно создавать графические программы. Для этого в Python по умолчанию применяется специальный тулкит - набор компонентов, который называется **tkinter**.

Тулкит tkinter доступен в виде отдельного встроенного модуля, который содержит все необходимые графические компоненты - кнопки, текстовые поля и т.д.

Базовым моментом в построении графических программ является создание окна. Затем в окно добавляются все остальные компоненты графического интерфейса. Поэтому создадим вначале простейшее окно. Для этого определим следующий скрипт:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | from tkinter import \*    root = Tk()  root.title("Графическая программа на Python")  root.geometry("400x300")    root.mainloop() |

Для создания графического окна применяется конструктор **Tk()**, который определен в модуле tkinter. Создаваемое окно присваивается переменной root, и через эту переменную мы можем управлять атрибутами окна. В частности, с помощью метода **title()**можно установить заголовок окна.

С помощью метода **geometry()** - размер окна. Для установки размера в метод geometry() передается строка в формате "Ширина x Высота". Если при создании окна приложения метод geometry() не вызывается, то окно занимает то пространство, которое необходимо для размещения внутреннего содержимого.

Для отображения окна надо вызвать у него метод **mainloop()**, который запускает цикл обработки событий окна для взаимодействия с пользователем.

В результате при запуске скрипта мы увидим такое пустое окошко:

### Начальная позиция окна

По умолчанию окно позиционируется в верхний левый угол экрана. Но мы можем изменить его положение, передав нужные значения в метод geometry():

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | from tkinter import \*    root = Tk()  root.title("Графическая программа на Python")  root.geometry("400x300+300+250")    root.mainloop() |

Теперь строка в методе geometry имеет следующий формат: "Ширина x Высота + координатаX + координатаY". То есть при запуске окно будет находиться на 300 пикселей вправо и на 250 пикселей вниз от верхнего левого угла экрана.