Шпаргалка по Python

49 карточек: короткий пример кода и подпись, где обычно ошибаются. Свежая версия, фильтр и песочница — по QR.

koddo.ru/cheatsheets/python

Основы Python

Первые конструкции, с которых начинается любой скрипт: вывод, переменные, ввод.

Переменные и присваивание

```
price = 199
qty = 3
total = price * qty       # 597
total += 100              # 697
price, qty = qty, price   # swap: price=3, qty=199
```

*Имя появляется в момент первого присваивания, объявлять тип не нужно. total += 100 — то же, что total = total + 100; так же работают -=, *= и /=.*

print: вывод и разделители

```
print('total', 597)        # total 597
print(1, 2, 3, sep=', ')   # 1, 2, 3
print('loading', end='... ')
print('done')              # loading... done
```

print сам ставит пробел между значениями и перенос строки в конце — sep и end это меняют. Для отладки печатай подпись рядом со значением: print(«total», total).

f-строки: значения внутри текста

```
name = 'Anna'
price = 199
print(f'{name}: {price} RUB')   # Anna: 199 RUB
print(f'{price * 0.9:.2f}')     # 179.10
```

Буква f перед кавычками включает подстановку — внутри {} работает любое выражение. Формат :.2f оставляет два знака после точки.

Комментарии

```
# comment until the end of the line
rate = 0.13  # tax rate

# temporarily disable a line:
# rate = 0.2
```

Всё после # интерпретатор пропускает — до конца строки. Хороший комментарий объясняет «почему», а не пересказывает код.

Типы и числа в Python

Числа, строки, булевы значения и приведение одного к другому.

Базовые типы и type()

```
age = 21          # int
price = 19.9      # float
name = 'Anna'     # str
active = True     # bool
type(price)       # <class 'float'>
```

Четыре типа закрывают почти все задачи новичка: int, float, str, bool. type(x) показывает, с чем ты имеешь дело — первый шаг, когда значение ведёт себя странно.

Преобразование типов

```
int('42')       # 42
float('19.9')   # 19.9
str(597)        # '597'
int('12a')      # ValueError
```

int(), float() и str() создают новое значение нужного типа. Строка с мусором внутри падает с ValueError — оборачивай вызов в try/except, если данные приходят извне.

Деление: /, // и %

```
7 / 2      # 3.5
7 // 2     # 3
7 % 2      # 1
597 % 10   # 7, the last digit
```

/ всегда возвращает float, // отбрасывает дробную часть, % даёт остаток. Пара // и % — стандартный способ разобрать число на цифры или минуты на часы.

Степень и корень: **

```
2 ** 10     # 1024
25 ** 0.5   # 5.0, the square root
10 ** -2    # 0.01
```

*** возводит в степень, дробный показатель даёт корень: x ** 0.5 — квадратный. Заметь: результат корня — float, даже когда он «ровный».*

round и abs

```
round(2.678)      # 3
round(2.678, 2)   # 2.68
abs(-15)          # 15
round(2.5)        # 2, not 3
```

round(x, n) округляет до n знаков, abs убирает знак. Подвох: round(2.5) даёт 2 — Python округляет половинки к чётному соседу, а не вверх.

Строки в Python

Форматирование, разбор и склейка — то, что нужно почти в каждой задаче.

Строка: индексы и срезы

```
word = 'python'
word[0]      # 'p'
word[-1]     # 'n'
word[:3]     # 'pyt'
word[::-1]   # 'nohtyp'
```

Строка индексируется как список: с нуля, минус — с конца, срезы те же. Изменить символ по индексу нельзя — строки неизменяемы, собирай новую.

Регистр и пробелы: upper, lower, strip

```
raw = '  Anna@Mail.RU  '
raw.strip()           # 'Anna@Mail.RU'
raw.strip().lower()   # 'anna@mail.ru'
'python'.upper()      # 'PYTHON'
```

strip срезает пробелы и переносы по краям, lower/upper меняют регистр. Методы не трогают исходную строку — результат нужно сохранить в переменную.

split и join: строка ↔ список

```
line = 'python,sql,go'
parts = line.split(',')    # ['python', 'sql', 'go']
text = ' / '.join(parts)   # 'python / sql / go'
```

split режет строку по разделителю и отдаёт список, join собирает список обратно в строку. Разделитель у join стоит слева: '-'.join(parts), а не parts.join('-').

replace: замена в строке

```
phone = '8-999-123-45-67'
phone.replace('-', ' ')     # '8 999 123 45 67'
phone.replace('-', '', 2)   # '8999123-45-67'
```

replace(old, new) меняет все вхождения и возвращает новую строку; третий аргумент ограничивает число замен. Удалить символ — заменить его на пустую строку ''.

Поиск в строке: in, find, count

```
email = 'anna@mail.ru'
'@' in email      # True
email.find('@')   # 4
email.find('#')   # -1
email.count('a')  # 3
```

Просто проверить наличие — оператор in, нужна позиция — find, количество — count. find возвращает -1, а не ошибку, когда ничего не нашёл — не забудь это проверить.

Перебор строки по символам

```
word = 'koddo'
count = 0
for char in word:
    if char in 'aeiou':
        count += 1
print(count)   # 2
```

for идёт по строке посимвольно — отдельный индекс не нужен. Паттерн «счётчик + условие в цикле» закрывает большинство задач на подсчёт символов.

Условия в Python

Ветвление и логические операторы, включая проверки на пустоту и None.

if / elif / else

```
score = 78
if score >= 90:
    grade = 'A'
elif score >= 70:
    grade = 'B'
else:
    grade = 'C'
# grade == 'B'
```

Ветки проверяются сверху вниз, выполняется первая истинная — остальные пропускаются. Веток elif может быть сколько угодно, else не обязателен.

Логика: and, or, not

```
age = 25
has_pass = False
age >= 18 and has_pass   # False
age >= 18 or has_pass    # True
not has_pass             # True
0 <= age <= 120          # True
```

and требует оба условия, or — хотя бы одно, not переворачивает значение. Двойное сравнение 0 <= age <= 120 работает без and — питоновская фишка.

in: проверка вхождения

```
cart = ['coffee', 'tea']
'tea' in cart           # True
'juice' not in cart     # True
'od' in 'koddo'         # True
5 in {1: 'a', 5: 'b'}   # True, checks keys
```

in работает со списками, строками, множествами и словарями. У словаря проверяются ключи, а не значения — для значений есть in dict.values().

Тернарное выражение

```
qty = 7
label = 'many' if qty >= 5 else 'few'   # 'many'
discount = 0.1 if qty >= 10 else 0      # 0
```

Однострочный if для выбора значения: сначала результат, потом условие. Для действий и нескольких веток бери обычный if — читается лучше.

Циклы в Python

Перебор коллекций, счётчики и досрочный выход из цикла.

for: цикл по коллекции

```
prices = [349, 99, 199]
total = 0
for price in prices:
    total += price
print(total)   # 647
```

for перебирает элементы напрямую — индексы для этого не нужны. Не меняй список, по которому идёшь: удаление элементов внутри цикла пропускает соседние.

for и range: цикл по числам

```
for i in range(3):
    print(i)   # 0, 1, 2

for i in range(2, 10, 2):
    print(i)   # 2, 4, 6, 8
```

range(stop) считает от нуля, правая граница не входит. Полная форма range(start, stop, step) задаёт начало и шаг.

while: цикл по условию

```
balance = 1000
months = 0
while balance > 0:
    balance -= 350
    months += 1
print(months)   # 3
```

while крутится, пока условие истинно — бери его, когда число шагов заранее неизвестно. Что-то внутри обязано менять условие, иначе цикл вечный.

break и continue

```
for price in [99, 199, -5, 349]:
    if price < 0:
        break      # stop the whole loop
    if price > 150:
        continue   # skip to the next item
    print(price)   # 99
```

break выходит из цикла целиком, continue пропускает только текущую итерацию. Во вложенных циклах break прерывает лишь внутренний.

enumerate: индекс + элемент

```
steps = ['clone', 'install', 'run']
for i, step in enumerate(steps, start=1):
    print(i, step)
# 1 clone
# 2 install
# 3 run
```

enumerate отдаёт пару «номер, элемент» — это питоничнее, чем range(len(items)). start задаёт, с какого числа нумеровать.

Вложенные циклы

```
for row in range(1, 3):
    for col in range(1, 4):
        print(row * col, end=' ')
    print()
# 1 2 3
# 2 4 6
```

Внутренний цикл прокручивается целиком на каждом шаге внешнего — итераций получается rows × cols. Больше двух уровней — сигнал вынести внутренний цикл в функцию.

Списки в Python

Срезы, сортировка, агрегаты и включения — самая рабочая часть языка.

Список: создание, индексы, len

```
cart = ['coffee', 'tea', 'juice']
len(cart)   # 3
cart[0]     # 'coffee'
cart[-1]    # 'juice'
cart[3]     # IndexError
```

Индексы идут с нуля, минус считает с конца. Последний элемент — cart[-1], а вот cart[len(cart)] всегда падает с IndexError.

Добавить и удалить: append, insert, remove, pop

```
cart = ['coffee', 'tea']
cart.append('juice')      # ['coffee', 'tea', 'juice']
cart.insert(0, 'water')   # ['water', 'coffee', 'tea', 'juice']
cart.remove('tea')        # by value
last = cart.pop()         # 'juice', and it is removed
```

append добавляет в конец, insert — по индексу, remove ищет по значению, pop снимает по индексу и возвращает элемент. remove убирает только первое совпадение.

Срезы: часть списка и разворот

```
items = [10, 20, 30, 40, 50]
items[1:3]    # [20, 30]
items[:2]     # [10, 20]
items[-2:]    # [40, 50]
items[::-1]   # [50, 40, 30, 20, 10]
```

Срез [a:b] берёт элементы с индекса a до b, не включая b, минус считает с конца; работает и со строками. [::-1] возвращает перевёрнутую копию, исходный список не меняется.

Сортировка: sorted, key и reverse

```
prices = [349, 99, 199]
sorted(prices)                # [99, 199, 349]
sorted(prices, reverse=True)  # [349, 199, 99]

words = ['banana', 'fig', 'apple']
sorted(words, key=len)        # ['fig', 'apple', 'banana']
```

sorted возвращает новый список, а items.sort() сортирует на месте и возвращает None — результат sort присваивать нельзя. key задаёт, по чему сравнивать элементы.

sum, min, max по списку

```
prices = [349, 99, 199]
sum(prices)               # 647
min(prices)               # 99
max(prices)               # 349
sum(prices) / len(prices) # 215.66...
```

Готовые агрегаты вместо цикла со счётчиком; среднее — sum / len. На пустом списке min и max падают с ValueError — проверь длину заранее.

Списковое включение

```
prices = [349, 99, 199]
doubled = [p * 2 for p in prices]       # [698, 198, 398]
cheap = [p for p in prices if p < 200]  # [99, 199]
```

Включение строит новый список из старого в одну строку: слева выражение, справа цикл и фильтр. Если читать стало трудно — разверни в обычный for.

Словари в Python

Ключи и значения: безопасный доступ, обход и подсчёт частот.

Словарь: доступ к значению без KeyError

```
stock = {'coffee': 12, 'tea': 5}
stock['coffee']         # 12
stock.get('juice')      # None
stock.get('juice', 0)   # 0
```

Квадратные скобки падают с KeyError, когда ключа нет. get возвращает None или значение по умолчанию вторым аргументом — удобно для счётчиков.

Словарь: добавить, изменить, удалить

```
stock = {'coffee': 12}
stock['tea'] = 5              # add
stock['coffee'] = 10          # change
del stock['tea']
qty = stock.pop('juice', 0)   # 0, no KeyError
```

Присваивание по новому ключу добавляет пару, по существующему — перезаписывает. del падает на отсутствующем ключе, pop со вторым аргументом — нет.

Перебор словаря: keys, values, items

```
stock = {'coffee': 12, 'tea': 5}
for name in stock:      # keys
    print(name)
for name, qty in stock.items():
    print(name, qty)    # coffee 12, tea 5
```

for по словарю идёт по ключам; пары «ключ, значение» даёт items(). Только значения — values().

Ключ в словаре: in

```
stock = {'coffee': 12, 'tea': 5}
'coffee' in stock   # True
12 in stock         # False, values do not count
if 'tea' in stock:
    stock['tea'] += 1
```

in проверяет только ключи — значение 12 словарь «не видит». Паттерн if key in dict страхует доступ по скобкам от KeyError.

Словарное включение

```
prices = {'coffee': 199, 'tea': 99}
sale = {n: p // 2 for n, p in prices.items()}
# {'coffee': 99, 'tea': 49}
cheap = {n: p for n, p in prices.items() if p < 150}
# {'tea': 99}
```

Как списковое включение, но с парой ключ: значение слева. Удобно, например, перевернуть словарь: {v: k for k, v in d.items()}.

Кортежи и множества в Python

Неизменяемые кортежи и множества: распаковка, дубликаты, операции над наборами.

Кортеж и распаковка

```
point = (10, 20)
x, y = point                    # x=10, y=20
lat, lon = 55.7, 37.6           # tuple without parens
first, *rest = [1, 2, 3, 4]     # 1, [2, 3, 4]
```

Кортеж — неизменяемый список; распаковка раскладывает его по переменным за один шаг. Звёздочка собирает «всё остальное» в список.

set: уникальные значения

```
emails = ['a@m.ru', 'b@m.ru', 'a@m.ru']
unique = set(emails)          # {'a@m.ru', 'b@m.ru'}
len(unique)                   # 2
deduped = list(set(emails))   # back to a list
```

set хранит каждое значение один раз — set(список) мгновенно убирает дубли. Подвох: множество не помнит порядок; нужен порядок — бери dict.fromkeys(items).

Операции множеств: | & -

```
py_devs = {'anna', 'oleg', 'ivan'}
js_devs = {'oleg', 'maria'}
py_devs & js_devs   # {'oleg'}, both
py_devs | js_devs   # everyone
py_devs - js_devs   # {'anna', 'ivan'}, python only
```

& — пересечение, | — объединение, минус — разница. Это в разы быстрее и короче, чем сверять два списка вложенными циклами.

Что выбрать: список, кортеж, множество, словарь

```
cart = ['tea', 'tea']    # list: order + duplicates
point = (55.7, 37.6)     # tuple: fixed pair
tags = {'sale', 'new'}   # set: unique, fast lookup
stock = {'tea': 5}       # dict: value by key
```

Порядок и повторы — список; «замороженная» запись — кортеж; уникальность и быстрый in — множество; поиск по ключу — словарь. Ошибся — конвертация в один вызов: list(), set(), tuple().

Функции в Python

Объявление, аргументы, возврат значений и короткие lambda.

Функция: def и return

```
def total_price(price, qty):
    return price * qty

result = total_price(199, 3)   # 597
```

return отдаёт значение и сразу выходит из функции. Без return она вернёт None: если тесты видят None — скорее всего, ты напечатал результат через print вместо return.

Параметры по умолчанию

```
def price_with_tax(price, rate=0.13):
    return round(price * (1 + rate), 2)

price_with_tax(199)        # 224.87
price_with_tax(199, 0.2)   # 238.8
```

Параметр со значением по умолчанию можно не передавать. Не ставь дефолтом список или словарь — он создаётся один раз и живёт между вызовами; используй None и создавай внутри.

Несколько возвращаемых значений

```
def min_max(items):
    return min(items), max(items)

low, high = min_max([349, 99, 199])
# low=99, high=349
```

return a, b возвращает кортеж, который распаковывают в переменные на месте вызова. Забыл распаковать — получишь один кортеж, а не два значения.

lambda для key

```
products = [('coffee', 349), ('tea', 99)]
sorted(products, key=lambda item: item[1])
# [('tea', 99), ('coffee', 349)]
max(products, key=lambda item: item[1])
# ('coffee', 349)
```

lambda — безымянная функция в одну строку, чаще всего живёт в key у sorted, min и max. Логика сложнее одного выражения — выноси в обычную def.

Ошибки и отладка в Python

Что означает трейсбек, как поймать исключение и чем отличаются частые ошибки.

Как читать traceback

```
Traceback (most recent call last):
  File "main.py", line 4, in <module>
    print(total(199, '3'))
  File "main.py", line 2, in total
    return price + qty
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

Читай снизу вверх: последняя строка — тип ошибки и причина, строки выше — путь до места падения, твой код обычно в нижней из File-строк. Тип плюс текст ошибки — готовый запрос для поиска по этому справочнику.

Частые ошибки: что значат

```
199 + '3'     # TypeError: mixing int and str
items[10]     # IndexError: index out of range
stock['x']    # KeyError: no such key
prnit('hi')   # NameError: name is not defined
int('12a')    # ValueError: bad value for int()
```

TypeError — смешал типы (чинится через int()/str()); IndexError и KeyError — обращение к тому, чего нет; NameError — опечатка в имени. Название ошибки говорит, куда смотреть, точную строку показывает traceback.

try/except: перехват ошибки

```
raw = 'abc'
try:
    value = int(raw)
except ValueError:
    value = 0

print(value)   # 0
```

Код в try выполняется до первой ошибки, except подхватывает её и идёт по запасному пути. Указывай конкретный тип: голый except прячет настоящие баги.

raise: выбросить ошибку

```
def set_price(price):
    if price < 0:
        raise ValueError('price must be >= 0')
    return price

set_price(-5)   # ValueError: price must be >= 0
```

raise останавливает функцию и отдаёт ошибку наверх — вызывающий код решит, ловить её или падать. Плохие входные данные лучше отклонить сразу, чем вернуть «тихий» None.

Отладка print'ом

```
def apply_discount(price, percent):
    print('price:', price, 'percent:', percent)
    result = price * (1 - percent / 100)
    print('result:', result)
    return result

apply_discount(200, 15)   # result: 170.0
```

Печатай подпись и значение на входе функции и перед return — сразу видно, где данные разошлись с ожиданием. Убери отладочные print перед сабмитом: лишний вывод ломает сравнение в тестах.