тип bool: True и False
Алгебра логики над bool

Итак, базово в Питоне есть тип bool, представлен значениями *True* и *False.* Всё, как у людей: Продолжение Сишной идеологии "Всё что не ноль - единица", логические операции *and* и *or*

>>> True
True
>>> 3 == 4
False
>>> True and False
False
>>> True or False
True
>>> False is False # один и тот же объект
True

>>> 1 < 5 or 5 > 3
True
>>> 1 < 5 and 5 > 3
True

Вообще любые объекты интерпретируются, как логические. Объекты делятся на Пустые и Непустые

>>> not 0
True
>>> not -123456
False
>>> not "asdfghj"
False
>>> not []
True
>>> not [1, 2, 3]
False

Кроме того, У нас есть логика возврата объектов из логических выражений в соответствии с ленивой логикой (н-р, выражение “Пустой объект and Непустой объект” вернёт тот самый Пустой объект, потому что на нём ленивая логика однозначно определяет ноль выражения. Или, н-р, “ПО1 or ПО2” вернёт второй Пустой Объект, тк ленивая логика на нём определила результат {очевидно, нулевой} выражения)

```
>>> 1 and 4
4
>>> [] and 4
[]
>>> "asdfg" and 1.234
1.234
>>> "asdfg" or 1.234
'asdfg'
>>> "" or 12345
12345
```

На этой логике построено Ленивое программирование

```
>>> False and 1/0
False # ошибки нет, тк до проблемного выражения мы даже не дошли
>>> False or 1/0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero

>>> 123456 or 1/0
123456

>>> a, b = 1, 2
>>> c = b and a/b or 100500
>>> c
0.5
>>> a, b = 1, 0
>>> c = b and a/b or 100500
>>> c
100500
```

Сравнения

```
>>> 3 <= 6
True
>>> type(13 < 6)
<class 'bool'>
>>>
```

Операция сравнения связывания - *is.* Мы про неё говорили в прошлом конспекте, она проверяет именно не равность объектов, а то, один и тот же ли он

>>> a = b = [1, 2, 3]
>>> a == b
True
>>> a is b
True

>>> a, b = [1, 2, 3], [1, 2, 3]
>>> a == b
True
>>> a is b
False
>>> id(a), id(b)
(139715253706304, 139715253703808)

Синтаксический сахар питона - удобная запись логических выражений

>>> a is not b # вот так
True
>>> not(a is b) # вместо вот так
True

in - операция проверки наличия в 'множестве'

>>> b
[1, 2, 3]
>>> 2 in b
True
>>> 12 not in b
True

Питончик, как мы выяснили, умеет в многоместные операции. Отсюда логично вытекают многоместные сравнения

>>> a, b, c, d = 1, 2, 3, 4
>>> a < b < c < d
True
>>> # раскрывается, как (a < b) and (b < c) and (c < d)
>>> # Поэтому...
>>>
>>> a != b < d > c
True

Переходим к условным операторам. Логика базовая: последовательная проверка, попадает в первое верное. Конструкция языка ***if-elif-else***

```
>>> a, b,  = 1, 2
>>>
>>> if a < b:
...     print("<")
... elif a > b:
...     print(">")
... else:
...     print("==")
...
<
```

Кроме условного оператора в питончике есть оператор match-case. Справедливости ради, match-case питона это волшебная штука сильно умнее сишного матчкейса. Там это просто селектор, а здесь - оператор сопоставления объекта из match объектам из case, причём со своим уникальным синтаксисом

В примитивном случае может работать, как селектор

```
>>> a = 123
>>>
>>> match a // 10:
...     case 12:
...         print("12")
...     case 1:
...         print("1")
...     case var:
...         print(var)
...
12
```

А на деле может обрабатывать многоэлементные конструкции (н-р, подаёшь ему список параметров, а он проверяет их по значениям, типам, просто под локальные переменные оператора подгоняет), и после этого можно внутри написать большой код обработки

Примерчик: нам задаётся строка s. Мы разбиваем её на слова (метод ***split***) и этот список слов подаём на вход match-case.
У нас три условия:
1. Если первое слово в списке “text”, а дальше любое количество любых слов (помним с прошлой лекции, что такой переменной со звёздочкой при присваивании обозначается возможность запихнуть любое количество значений, и нулевое тоже), то напечатать текст
2. Если первое слово “info”, потом одно из трёх слов “x”, ”y”, ”xy”, которые мы в этом локальном коде будем обозначать переменной data, то сделать свой особенный вывод *про конструкцию форматных строк, а это именно они, будет в будущих конспектах*
3. Если вывод любой другой (в переменную _ (да, это переменная с таким названием, почему бы и нет, собственно) запишется просто весь список, он же один объект, и тут одна переменная на него), то вон вывести Пупупу.

```
match s.split():
    case ['say', *text]:
        print(*text)
    case ['info', 'x'|'y'|'xy' as data]:
        print(f'{x}' * (data == 'x') +\
              f'{x} {y}' * (data == 'xy') +\
              f'{y}' * (data == 'y'))
    case _:
        print(“Pupupu…”)
```

Давай про более простые вещи - Операторы цикла, это база, как и везде, сюда же ключевые слова ***break, continue***

Ровно как и в Си, ввод значений можно засунуть в условие цикла, он будет читать, возвращать прочитанное и его проверять. Единственное, для ввода в условии цикла нужно явно указывать, что мы вот прям хотим эту операцию провести сначала, а потому уже результат этот проверять, поэтому пишем синтаксис Паскалевского присваивания. Объяснение этому есть чуть более умное, но его вспоминать и писать сюда мне лень.

*К слову, знакомимся здесь с функцией ввода ***input().*** У неё тоже есть свои базовыепараметры по умолчанию, н-р, можно дать ему в аргументы строку, которую input выведет перед тем, как запросить ввод, не надо будет лишний принт писать, он уже как бы предвстроенный есть*

```
>>> while (a := input()) != "":
...     print("@@@", a)
...
qwerty
@@@ qwerty
sdftg
@@@ sdftg
sedrftgh
```

@@@ sedrftgh

```
>>> # Поиск первого
>>> res = False
>>> while (a := input()) != "":
...     if a == "QQ":
...         res = True
...         break
...     print(res)
asdfgh
werty
sdfgh
```

Красота Питона: клауза else для while. Она срабатывает, когда цикл завершился корректно (то есть по условию while вышел), но при этом внутри тела цикла не выполнено какое-либо условие выхода (Логично. что если никаких условий выхода внутри нет, нечему выполняться, else будет отрабатывать)

```
>>> while (a := input()) != "":
...     if a == "QQ":
...         print("YESSSS QQ")
...         break
... else:
...     print("no QQ")
...
asdfghj
wertyu
xcvbnjm
no QQ

>>> while (a := input()) != "":
...     if a == "QQ":
...         print("YESSSS QQ")
...         break
... else:
...     print("no QQ")
...
asdfghj
QQ
YESSSS QQ

>>> while (a := input()) != "":
... else:
...     print("no QQ")
...
asdfghj
xcvbnjm
no QQ
```