Программа Python.
2021-2022 год.

Продолжаем изучать действия со строками.

Мы можем получить любой символ строки по индексу символа, обратившись к строке напрямую.

```
m = "metro"[2]
print(m) # t
```

Мы можем получить любой символ строки по индексу символа, обратившись к строке через переменную, в которой находится строка.

```
m = "metro"
b = m[2]

print(b) # t
```

Правило: можно по индексу получать символ не только если обратиться к переменной, в котором находится объект - строка, но и можно к объекту строка обратиться напрямую и получить любой символ по индексу.

При этом тоже нужно понимать, что "metro" это набор символов. И если с этим набором мы работаем в разных переменных, то связи с переменными нет. Изменения в одной переменной не повлияют на результат в другой переменной.

```
m = "metro"
a = "metro"[2]

print(a) # t
```

Теперь, если поменять строку внутри переменной a:

```
a = "merro"[2]

print(m) # metro - значение переменной не изменилось
print(a) # r - значение переменной изменилось
```

Еще один пример:

```
m = 'metro'
print(m[2]) # t

m = "metro"[2][0]
print(m) # t
```

В этом примере надо понимать, что мы уже обращаемся к одной букве t, как к объекту. А у объекта строка с одним символом есть только один индекс - 0! Если мы попробуем обратиться ко второму индексу объекта строка с одним символом (t), то получим ошибку.

```
m = "metro"[2][2]
print(m)
```

builtins.IndexError: string index out of range

встроенные.IndexError: строковый индекс вне диапазона

Ошибки нужно переводить - тогда становится более понятно!

Можно производить склейку операций над строкой через точку.

```
m = 'metro'
print(m) - metro

m = m.replace('t', 'm')
print(m) - memro
m = m.replace('t', 'm')[2:]
print(m) - mro
```

Но если у нас replace('t', 'm') и replace('t', 'm')[2:] это отдельные операции, то операции можно склеивать в строке через запятую.

```
m = 'metro'
m = m.replace('t', 'm')[2:].replace('m', 't')
print(m)
```

Результат: tro

```
m = 'metro'
```

```
m = m.replace('t', 'm')[2:].replace('m', 't').find('r')
print(m)
```

Результат: 1

Есть специальные служебные символы для работы со строками. К примеру, символ - \n - символ перевода на следующую строку.

```
m = "met\nro"
print(m)

#   met
    ro
```

При этом символ **\n** для Python это один символ!
Проверим наше утверждение с помощью функции len()

```
m = "\n"
print(len(m)) # 1
```

Заменим \ на букву n

```
m = 'nn'
print(len(m)) # 2 - Уже в строке 2 символа!
```

Если возникнет необходимость напечатать \n реально на экране, не используя как служебный символ, то необходимо произвести экранирование этого символа.

```
m = "\\n"
print(m) # \n
```

Еще пример:

```
m = 'me't'ro'
print(m)
```

Syntax Error: invalid syntax: C:\Users\packard bell tm87\Desktop\untitled-2.py, line 1, pos 9
m = 'me't'ro'

```
m = 'me\'t\'ro'
print(m) - me't'ro
```

Условные операторы.

В языке Python мы познакомились с такими сущностями:

- Переменные **(имя и ссылка на значение в памяти)**
- Функции и методы **abc(...параметры..)**
- Операции **(* / - +)**
- Операторы **(=)**

Есть еще сущности - Условные операторы:

Зачем они нужны?

Для того, чтобы выполнять строки кода программы в определенном порядке, а не последовательно от начала до конца - строка за строкой, каждый раз при запуске программы.
Условные операторы это один из способов задать выполнение программы в своем определенном порядке. Далее мы рассмотрим еще способы (циклы и функции), которые тоже помогают задавать свой алгоритм выполнения команд в программе.

Допустим, есть программа, которая печатает YES и NO.

```
print ('YES')
print ('NO')

#   YES
    NO
```

Как мне переписать код программы, чтобы программа печатала при одном условии только YES или при другом условии только NO?

```
a = 5
if a>0:
 print ('YES')

# YES
```

Если переписать значение нашей переменной a:

```
a = -1
if a > 0:
    print('YES')

# нет результата! Функция print() не сработала!
```

Чтобы команда print() сработала, необходимо поместить команду print() внутрь условного оператора if. Вложенность инструкций (команд) внутри конструкции if в Python никак специально не выделяется (скобки, кавычки и т.д.). Для этого необходимо отступить вправо (принято 4 пробела, хотя можно и 1). Но пробел нужен обязательно. Так же необходимо двоеточие на строке, где указывается условие if. Двоеточие говорит о том, что условие закончилось, и на следующей строке внутри if будут находиться команды, которые необходимо будет выполнять (конечно, если условие истинно True).

Почему 4 пробела?
Это принятый стиль программирования (для Python).
PEP 8 — документ, описывающий соглашение о том, как писать код на языке Python. Документ PEP 8 создан на основе рекомендаций создателя языка Гвидо Ван Россума. Ключевая идея Гвидо такова: код читается намного больше раз, чем пишется. Собственно, рекомендации о стиле написания кода направлены на то, чтобы улучшить читаемость кода и сделать его согласованным между большим числом проектов. В идеале, если весь код будет написан в едином стиле, то любой сможет легко его прочесть.

С этим документом можно ознакомиться в интернете.

А пока запоминаем: 4 пробела и двоеточие!

Теперь наша программа напечатает YES, только если выполняется условие a > 0. Если условие не будет выполняться, то строки кода внутри оператора if не будут выполняться потому, что будут, пропущены при выполнении всей программы. Просто программа не войдет внутрь оператора if и продолжит выполнять команды после блока if с новой строки.
А так как на новой строке, после условия if у нас в данной программе нет ничего, то мы не увидим слова YES и вообще

ничего не увидим. Чтобы понять, что программа прошла мимо условия if и продолжила свое выполнение ниже условия, добавим в самом конце строчку кода: **print('Конец программы').**

```
a = -1

if a > 0:
    print('YES')

print('Конец программы')
```

 # Конец программы (если a = -1)

print('Конец программы') - выполнилась только эта строка кода!

Теперь мы увидели, что даже если программа не зашла в условный оператор if, то она продолжает дальше выполнять строки кода уже после конструкции if.

Создадим программу, которая печатает при определенном условии только YES или только NO.

```
a = 2
if a>0:
 print ('YES')

print ('NO')
```

```
YES
NO
```

 # программа напечатает сначала YES, потом NO. Это неудобно, так как логично, что нам надо напечатать в зависимости от условий в нашей программе только YES или только NO. Поэтому такой вариант синтаксиса оператора if нам не совсем подходит. Перейдем ко второму варианту синтаксиса оператора if:

```
a = 2

if a > 0:
    print('YES')
else:
```

```
    print('NO')
```

YES

У оператора if есть необязательное продолжение (else - иначе)
А что значит else в нашей программе? Это очень важно! Это не только когда a<0! Это еще и когда a = 0! Очень важный момент, часто разработчики забывают и получают ошибки в логике программы.

a = -1 (переменная a явно меньше 0)

```
if a > 0:
    print('YES')
else:
    print('NO')
```

NO

a = 0 (переменная a не меньше 0! переменная a = 0!)

```
if a > 0:
    print('YES')
else:
    print('NO')
```

NO

Клавиша «TAB» на клавиатуре вставляет свой отступ в текстовых редакторах, он ничего общего с пробелами не имеет. Но в среде разработки WING IDE нажатие клавиши «TAB» действительно поставит нужное количество пробелов для написания кода в языке Python.

Но нужно обязательно учитывать тот факт, что Python не позволяет использовать в одной строке табуляцию и пробелы! Поэтому, к примеру, если мы копируем какой-то текст в программе из одного места и вставляем в WING, то при использовании TAB может возникнуть ошибка.

Теперь мы можем с помощью оператора if обрабатывать два условия. А как быть, если нам надо обработать отдельно три условия или более?

Можно, к примеру, создать вложенный оператор if:

```
a = -10
if a>0:
  print ('YES')
else:
  if a<0:
    print ('NO')
  else:
    print ('OK') # эта строка напечатается, если a = 0.
```

Но такой вложенный синтаксис оператора if удобен, если у нас проверяется три, четыре (макс) условия. Потому, что если нам надо проверить более 4-х условий, то такая конструкция становится не очень удобной для чтения (для восприятия глазу).

Поэтому придумали еще один вид синтаксиса оператора if: Добавили возможность ввести в оператор if дополнительный синтаксис elif (объединили else и if). При этом elif мы можем вставлять в оператор if сколько угодно раз. Else тоже остается, как условие, которое сработает, если никакие из специально запрограммированных условий не сработают.

```
a = -20
if a>0:
  print ('YES')
elif a<0:
  print ('NO')
else:
  print ('OK')

# Вопрос может возникнуть: А что если написать следующим образом:

a = 10
if a>0:
  print ('YES')
if a<0:
  print ('NO')
if a==0:
  print ('OK')
```

Проверяем подряд несколько условий и выводим соответствующий текст.
В принципе по логике программа выполняет одно и то же.

Но есть все же серьезные отличия: Первое это скорость выполнения программы. В первом случае с elif мы проверяем только одно условие и все. Если одно условие сработало, то программа выполняет функцию print() и останавливается. Во втором случае (без elif) программа будет обязана выполнять всегда три проверки подряд (весь код программы), даже если сработало первое условие.

Второе отличие, когда программа должна выполнять весь код программы (**без elif)**, продемонстрирует один пример:

```
# без elif

a = 10
if a>0:
  print ('YES')
  a = 0
if a<0:
  print ('NO')
if a==0:
  print ('OK')
```

```
YES
OK
```

Здесь мы видим, что программа без elif выполнит еще одно условие (a = 0), хотя нам по идее этого не нужно, мы же зашли в условие a>0 выполнили print() и этого достаточно. Но из-за того, что мы переписали значение переменной a = 0, то программа вынуждена выполнить и второе условие a==0.

Перепишем нашу программу с elif:

```
# с elif

a = 10
if a>0:
  print ('YES')
  a = 0
```

```
elif a<0:
  print ('NO')
else:
  print ('OK')
```

YES

Но в случае использования elif такого мы не наблюдаем. Выполнилось одно условие и дальше другие условия не выполняются. Выполняться print() будет только в одном условии.

```
a = 10

if a > 0:
    print('YES')
    a = 0
elif a < 0:
    print('NO')
else:
    print('OK')

print(a) - хотя переменная a конечно поменяла значение!
```

YES
0

Вот поэтому и создан был elif. Во-первых, ускоряет выполнение программы (нет лишних проверок условий). Во-вторых, иногда опасно проверять лишние условия (не будут выполняться условия, которые по сценарию программы в данный момент не нужно выполнять).