Контроль хода программы

До сих пор, весь код выполнялся последовательно - все строки скрипта выполнялись в том порядке, в котором они записаны в файле. В этом разделе рассматриваются возможности Python в управлении ходом программы:

- ответвления в ходе программы с помощью конструкции `if/elif/else`
- повторение действий в цикле с помощью конструкций `for` и `while`
- обработка ошибок с помощью конструкции `try/except`

if/elif/else

Конструкция `if/elif/else` позволяет делать ответвления в ходе программы. Программа уходит в ветку при выполнении определенного условия.

В этой конструкции только if является обязательным, elif и else опциональны:

- Проверка if всегда идет первой.
- После оператора if должно быть какое-то условие: если это условие выполняется (возвращает True), то действия в блоке if выполняются.
- С помощью elif можно сделать несколько разветвлений, то есть, проверять входящие данные на разные условия.
- Блок elif это тот же if, но только следующая проверка. Грубо говоря, это «а если …»
- Блоков elif может быть много.
- Блок else выполняется в том случае, если ни одно из условий if или elif не было истинным.

Пример конструкции:

```
In [1]: a = 9

In [2]: if a == 10:
   ...:     print('a равно 10')
   ...: elif a < 10:
   ...:     print('a меньше 10')
   ...: else:
   ...:     print('a больше 10')
   ...:
a меньше 10
```

Условия

Конструкция if построена на условиях: после if и elif всегда пишется условие. Блоки if/elif выполняются только когда условие возвращает True, поэтому первое с чем надо разобраться - это что является истинным, а что ложным в Python.

True и False

В Python, кроме очевидных значений True и False, всем остальным объектам также соответствует ложное или истинное значение:

- истинное значение:
  - любое ненулевое число
  - любая непустая строка – любой непустой объект

- ложное значение:
  - 0
  - None
  - пустая строка
  - пустой объект

Например, так как пустой список это ложное значение, проверить, пустой ли список, можно таким образом:

```
In [12]: list_to_test = [1, 2, 3]

In [13]: if list_to_test:
   ....:     print("В списке есть объекты")
   ....:
В списке есть объекты
```

Тот же результат можно было бы получить несколько иначе:

```
In [14]: if len(list_to_test) != 0: ....:
       print("В списке есть объекты")
   ....:
В списке есть объекты
```

Операторы сравнения

Операторы сравнения, которые могут использоваться в условиях:

```
In [3]: 5 > 6 Out[3]: False

In [4]: 5 > 2 Out[4]: True

In [5]: 5 < 2 Out[5]: False

In [6]: 5 == 2 Out[6]: False

In [7]: 5 == 5 Out[7]: True

In [8]: 5 >= 5
Out[8]: True

In [9]: 5 <= 10
Out[9]: True

In [10]: 8 != 10
Out[10]: True
```

Примечание: обратите внимание, что равенство проверяется двойным ==.

Пример использования операторов сравнения:

```
In [1]: a = 9

In [2]: if a == 10:
   ...:     print('a равно 10')
   ...: elif a < 10:
   ...:     print('a меньше 10')
   ...: else:
   ...:     print('a больше 10')
   ...:
a меньше 10
```

Оператор in

Оператор `in` позволяет выполнять проверку на наличие элемента в последовательности (например, элемента в списке или подстроки в строке):

```
In [8]: 'Fast' in 'FastEthernet'
Out[8]: True

In [9]: 'Gigabit' in 'FastEthernet'
Out[9]: False

In [10]: vlan = [10, 20, 30, 40]

In [11]: 10 in vlan Out[11]: True

In [12]: 50 in vlan
Out[12]: False
```

При использовании со словарями условие **in** выполняет проверку по ключам словаря:

```
In [15]: r1 = {
   ....: 'IOS': '15.4',
   ....: 'IP': '10.255.0.1',
   ....: 'hostname': 'london_r1',
   ....: 'location': '21 New Globe Walk',
   ....: 'model': '4451',
   ....: 'vendor': 'Cisco'}

In [16]: 'IOS' in r1
Out[16]: True

In [17]: '4451' in r1
Out[17]: False
```

Операторы and, or, not

В условиях могут также использоваться **логические операторы** `and`, `or`, `not`:

```
In [15]: r1 = {
   ....: 'IOS': '15.4',
   ....: 'IP': '10.255.0.1',
....: 'hostname': 'london_r1',
   ....: 'location': '21 New Globe Walk',
   ....: 'model': '4451',
   ....: 'vendor': 'Cisco'}

In [18]: vlan = [10, 20, 30, 40]

In [19]: 'IOS' in r1 and 10 in vlan
Out[19]: True

In [20]: '4451' in r1 and 10 in vlan
Out[20]: False

In [21]: '4451' in r1 or 10 in vlan
Out[21]: True

In [22]: not '4451' in r1
Out[22]: True

In [23]: '4451' not in r1
Out[23]: True
```

Оператор and

В Python оператор `and` возвращает не булево значение, а значение одного из операндов.

Если оба операнда являются истиной, результатом выражения будет последнее значение:

```
In [24]: 'string1' and 'string2'
Out[24]: 'string2'

In [25]: 'string1' and 'string2' and 'string3'
Out[25]: 'string3'
```

Если один из операторов является ложью, результатом выражения будет первое ложное значение:

```
In [26]: '' and 'string1'
Out[26]: ''

In [27]: '' and [] and 'string1'
Out[27]: ''
```

Оператор or

Оператор `or`, как и оператор and, возвращает значение одного из операндов.

При оценке операндов возвращается первый истинный операнд:

```
In [28]: '' or 'string1'
Out[28]: 'string1'

In [29]: '' or [] or 'string1'
Out[29]: 'string1'

In [30]: 'string1' or 'string2'
Out[30]: 'string1'
```

Если все значения являются ложными, возвращается последнее значение:

```
In [31]: '' or [] or {}
Out[31]: {}
```

Важная особенность работы оператора `or` - операнды, которые находятся после истинного, не вычисляются:

```
In [33]: '' or sorted([44,1,67])
Out[33]: [1, 44, 67]

In [34]: '' or 'string1' or sorted([44,1,67])
Out[34]: 'string1'
```

Пример использования конструкции if/elif/else

Пример скрипта check_password.py, который проверяет длину пароля и есть ли в пароле имя пользователя:

```
# -*- coding: utf-8 -*-

username = input('Введите имя пользователя: ')
password = input('Введите пароль: ')

if len(password) < 8: print('Пароль
    слишком короткий')
elif username in password:
print('Пароль содержит имя пользователя')

else:
    print('Пароль для пользователя {} установлен'.format(username))
```

Проверка скрипта:

```
$ python check_password.py
Введите имя пользователя: nata
Введите пароль: nata1234
Пароль содержит имя пользователя
```

```
$ python check_password.py
Введите имя пользователя: nata
Введите пароль: 123nata123
Пароль содержит имя пользователя

$ python check_password.py
Введите имя пользователя: nata
Введите пароль: 1234
Пароль слишком короткий

$ python check_password.py
Введите имя пользователя: nata
Введите пароль: 123456789
Пароль для пользователя nata установлен
```

Тернарное выражение (Ternary expression)

Иногда удобнее использовать тернарный оператор, нежели развернутую форму:

```
s = [1, 2, 3, 4]
result = True if len(s) > 5 else False
```

Этим лучше не злоупотреблять, но в простых выражениях такая запись может быть полезной. **for**

Очень часто одно и то же действие надо выполнить для набора однотипных данных. Например, преобразовать все строки в списке в верхний регистр. Для выполнения таких действий в Python используется цикл `for`.

Цикл for перебирает по одному элементы указанной последовательности и выполняет действия, которые указаны в блоке for, для каждого элемента.

Примеры последовательностей элементов, по которым может проходиться цикл for:

- строка
- список
- словарь
- *Функция range*
- любой *Итерируемый объект*

Пример преобразования строк в списке в верхний регистр без цикла for:

```
In [1]: words = ['list', 'dict', 'tuple']

In [2]: upper_words = []

In [3]: words[0]
Out[3]: 'list'

In [4]: words[0].upper() # преобразование слова в верхний регистр
Out[4]: 'LIST'

In [5]: upper_words.append(words[0].upper()) # преобразование и добавление в
   ...: новый список

In [6]: upper_words
Out[6]: ['LIST']

In [7]: upper_words.append(words[1].upper())

In [8]: upper_words.append(words[2].upper())

In [9]: upper_words
Out[9]: ['LIST', 'DICT', 'TUPLE']
```

У данного решения есть несколько нюансов:

- одно и то же действие надо повторять несколько раз
- код привязан к определенному количеству элементов в списке words Те же действия с циклом for:

```
In [10]: words = ['list', 'dict', 'tuple']

In [11]: upper_words = []
```

```
In [12]: for word in words:
    ...:     upper_words.append(word.upper())
    ...:

In [13]: upper_words
Out[13]: ['LIST', 'DICT', 'TUPLE']
```

Выражение `for word in words: upper_words.append(word.upper())` означает «для каждого слова в списке words выполнить действия в блоке for». При этом word это имя переменной, которое каждую итерацию цикла ссылается на разные значения.

Примечание: Проект pythontutor может очень помочь в понимании циклов. Там есть специальная визуализация кода, которая позволяет увидеть, что происходит на каждом этапе выполнения кода, что особенно полезно на

первых порах с циклами. На сайте pythontutor можно загружать свой код, но для примера, по этой ссылке можно посмотреть пример выше.

Цикл for может работать с любой последовательностью элементов. Например, выше использовался список и цикл перебирал элементы списка. Аналогичным образом for работает с кортежами.

При работе со строками цикл for перебирает символы строки, например:

```
In [1]: for letter in 'Test string':
   ...:     print(letter)
   ...:
T
e
s
t

s
t
r
i
n
g
```

Примечание: В цикле используется переменная с именем **letter**. Хотя имя может быть любое, удобно, когда имя подсказывает, через какие объекты проходит цикл.

Иногда в цикле необходимо использовать последовательность чисел. В этом случае, лучше всего использовать функцию *Функция range*

Пример цикла `for` с функцией `range()`:

```
In [2]: for i in range(10):
   ...:     print('interface FastEthernet0/{}'.format(i))
   ...:
interface FastEthernet0/0
interface FastEthernet0/1
interface FastEthernet0/2
interface FastEthernet0/3
interface FastEthernet0/4
interface FastEthernet0/5
interface FastEthernet0/6
interface FastEthernet0/7
interface FastEthernet0/8
interface FastEthernet0/9
```

В этом цикле используется `range(10)`. Функция range генерирует числа в диапазоне от нуля до указанного числа (в данном примере - до 10), не включая его.

В этом примере цикл проходит по списку VLANов, поэтому переменную можно назвать vlan:

```
In [3]: vlans = [10, 20, 30, 40, 100]
In [4]: for vlan in vlans:
   ...:     print('vlan {}'.format(vlan)) ...:
      print(' name VLAN_{}'.format(vlan)) ...:
vlan 10 name VLAN_10
vlan 20 name
 VLAN_20
vlan 30 name
 VLAN_30
vlan 40 name
 VLAN_40
vlan 100
 name VLAN_100
```

Когда цикл идет по словарю, то фактически он проходится по ключам:

```
In [34]: r1 = {
    ...:     'ios': '15.4',
    ...:     'ip': '10.255.0.1',
    ...:     'hostname': 'london_r1',
    ...:     'location': '21 New Globe Walk',
    ...:     'model': '4451',
```

```
    ...:     'vendor': 'Cisco'}
    ...:

In [35]: for k in r1:
    ...:     print(k)
    ...:
ios ip
hostname
location
model
vendor
```

Если необходимо выводить пары ключ-значение в цикле, можно делать так:

```
In [36]: for key in r1:
    ...:     print(key + ' => ' + r1[key])
    ...:
ios => 15.4 ip => 10.255.0.1
hostname => london_r1
location => 21 New Globe
Walk model => 4451 vendor =>
Cisco
```

Или воспользоваться методом items, который позволяет проходится в цикле сразу по паре ключ-значение:

```
In [37]: for key, value in r1.items():
    ...:     print(key + ' => ' + value)
    ...:
ios => 15.4 ip => 10.255.0.1
hostname => london_r1
location => 21 New Globe
Walk model => 4451 vendor =>
Cisco
```

Метод items возвращает специальный объект view, который отображает пары ключ-значение:

```
In [38]: r1.items()
Out[38]: dict_items([('ios', '15.4'), ('ip', '10.255.0.1'), ('hostname', 'london_
↪r1'), ('location', '21 New Globe Walk'), ('model', '4451'), ('vendor', 'Cisco
↪')])
```

Вложенные for

Циклы for можно вкладывать друг в друга.

В этом примере в списке commands хранятся команды, которые надо выполнить для каждого из интерфейсов в списке fast_int:

```
In [7]: commands = ['switchport mode access', 'spanning-tree portfast',
 'spanning→tree bpduguard enable']

In [8]: fast_int = ['0/1','0/3','0/4','0/7','0/9','0/10','0/11']


In [9]: for intf in fast_int:
   ...:     print('interface FastEthernet
   {}'.format(intf)) ...:for command in commands: ...:
      print(' {}'.format(command))
   ...:
interface FastEthernet 0/1
 switchport mode access
 spanning-tree portfast
 spanning-tree bpduguard enable
interface FastEthernet 0/3
 switchport mode access
 spanning-tree portfast spanning-
 tree bpduguard enable
interface FastEthernet 0/4
 switchport mode access
 spanning-tree portfast
 spanning- tree bpduguard enable
...
```

Первый цикл for проходится по интерфейсам в списке fast_int, а второй по командам в списке commands.

Совмещение for и if

Рассмотрим пример совмещения for и if.

Файл generate_access_port_config.py:

```
access_template = ['switchport mode access',
                   'switchport access vlan',
                   'spanning-tree portfast',
                   'spanning-tree bpduguard enable']

fast_int = {'access': { '0/12':10,
             '0/14':11,
             '0/16':17,
             '0/17':150}}

for intf, vlan in fast_int['access'].items():
    print('interface FastEthernet' + intf)
    for command in access_template:
        if command.endswith('access vlan'):
        print(' {} {}'.format(command, vlan))
    else:
        print(' {}'.format(command))
```

Комментарии к коду:

- В первом цикле for перебираются ключи и значения во вложенном словаре fast_int[„access“]
- Текущий ключ, на данный момент цикла, хранится в переменной intf
- Текущее значение, на данный момент цикла, хранится в переменной vlan
- Выводится строка interface FastEthernet с добавлением к ней номера интерфейса
- Во втором цикле for перебираются команды из списка access_template
- Так как к команде switchport access vlan надо добавить номер VLAN:
  - внутри второго цикла for проверяются команды
  - если команда заканчивается на access vlan
    - выводится команда, и к ней добавляется номер VLAN
  - во всех остальных случаях просто выводится команда

Результат выполнения скрипта:

```
$ python
generate_access_port_config.py
interface FastEthernet0/12 switchport
mode access switchport access vlan 10
spanning-tree portfast spanning-tree
bpduguard enable
interface FastEthernet0/14
 switchport mode access
 switchport access vlan 11
 spanning-tree portfast
 spanning-tree bpduguard enable
```

```
interface FastEthernet0/16
 switchport mode access
 switchport access vlan 17
```

```
 spanning-tree portfast
 spanning-tree bpduguard enable
interface FastEthernet0/17
 switchport mode access
 switchport access vlan 150
 spanning-tree portfast
 spanning-tree bpduguard enable
```

while

Цикл while - это еще одна разновидность цикла в Python.

В цикле while, как и в выражении if, надо писать условие. Если условие истинно, выполняются действия внутри блока while. При этом, в отличие от if, после выполнения кода в блоке, while возвращается в начало цикла.

При использовании циклов while необходимо обращать внимание на то, будет ли достигнуто такое состояние, при котором условие цикла будет ложным.

Рассмотрим простой пример:

```
In [1]: a = 5

In [2]: while a > 0:
   ...:     print(a)
   ...:     a -= 1 # Эта запись равнозначна a = a - 1
   ...:
5
4
3
2
1
```

Сначала создается переменная a со значением 5.

Затем, в цикле while указано условие a > 0. То есть, пока значение a больше 0, будут выполняться действия в теле цикла. В данном случае будет выводиться значение переменной a.

Кроме того, в теле цикла при каждом прохождении значение a становится на единицу меньше.

Примечание: Запись `a -= 1` может быть немного необычной. Python позволяет использовать такой формат вместо `a = a - 1`.

Аналогичным образом можно писать: `a += 1`, `a *= 2`, `a /= 2`.
Так как значение a уменьшается, цикл не будет бесконечным, и в какой-то момент выражение a > 0 станет ложным.

Следующий пример построен на основе примера про пароль из раздела о конструкции if *Пример использования конструкции if/elif/else*. В том примере приходилось заново запускать скрипт, если пароль не соответствовал требованиям.

С помощью цикла while можно сделать так, что скрипт сам будет запрашивать пароль заново, если он не соответствует требованиям.

Файл check_password_with_while.py:

```
# -*- coding: utf-8 -*-

username = input('Введите имя пользователя: ' )
password = input('Введите пароль: ' )

password_correct = False

while not password_correct:
    if len(password) < 8:
        print('Пароль слишком короткий\n') password
        = input('Введите пароль еще раз: ' )
    elif username in password:
        print('Пароль содержит имя пользователя\n')
        password = input('Введите пароль еще раз: ' )
    else:
        print('Пароль для пользователя {} установлен'.format( username ))
        password_correct = True
```

В этом случае цикл while полезен, так как он возвращает скрипт снова в начало проверок, позволяет снова набрать пароль, но при этом не требует перезапуска самого скрипта.

Теперь скрипт отрабатывает так:

```
$ python check_password_with_while.py
Введите имя пользователя: nata
Введите пароль: nata
Пароль слишком короткий
```

(continues on next page)

```
Введите пароль еще раз: natanata
Пароль содержит имя пользователя

Введите пароль еще раз: 123345345345
Пароль для пользователя nata установлен
```

break, continue, pass

В Python есть несколько операторов, которые позволяют менять поведение циклов по умолчанию.

Оператор break

Оператор break позволяет досрочно прервать цикл:

- break прерывает текущий цикл и продолжает выполнение следующих выражений
- если используется несколько вложенных циклов, break прерывает внутренний цикл и продолжает выполнять выражения, следующие за блоком * break может использоваться в циклах for и while

Пример с циклом for:

```
In [1]: for num in range(10):
   ...:     if num < 7:
   ...:         print(num)
   ...:     else: ...:
       break
   ...:
0
1
2
3
4
5
6
```

Пример с циклом while:

```
In [2]: i = 0
In [3]: while i < 10:
   ...:     if i == 5:
   ...:         break
   ...:     else:
   ...:         print(i)
   ...:         i += 1
0
1
2
3
4
```

Использование break в примере с запросом пароля (файл check_password_with_while_break.py):

```
username = input('Введите имя пользователя: ' )
password = input('Введите пароль: ' )

while True:
    if len(password) < 8: print('Пароль
        слишком короткий\n')
    elif username in password:
        print('Пароль содержит имя пользователя\n')
    else:
        print('Пароль для пользователя {} установлен'.format(username))
        # завершает цикл while
        break
    password = input('Введите пароль еще раз: ')
```

Теперь можно не повторять строку `password = input('Введите пароль еще раз: ')` в каждом ответвлении, достаточно перенести ее в конец цикла.

И, как только будет введен правильный пароль, break выведет программу из цикла while.

Оператор continue

Оператор continue возвращает управление в начало цикла. То есть, continue позволяет «перепрыгнуть» оставшиеся выражения в цикле и перейти к следующей итерации.

Пример с циклом for:

```
In [4]: for num in range(5):
   ...:     if num == 3:
   ...:     continue
   ...:     else:
   ...:         print(num)
...:
0
1
2
4
```

Пример с циклом while:

```
In [5]: i = 0
In [6]: while i < 6:
   ....:     i += 1
   ....:     if i == 3:
   ....:         print("Пропускаем 3")
   ....:         continue
   ....:     print("Это никто не увидит") ....:
      else:
   ....:         print("Текущее значение: ", i)
   ....:
Текущее значение: 1
Текущее значение: 2
Пропускаем 3
Текущее значение: 4
Текущее значение: 5
Текущее значение: 6
```

Использование continue в примере с запросом пароля (файл check_password_with_while_continue.py):

```
username = input('Введите имя пользователя: ')
password = input('Введите пароль: ')

password_correct = False

while not password_correct:
    if len(password) < 8: print('Пароль
        слишком короткий\n')
    elif username in password:
        print('Пароль содержит имя пользователя\n')
    else:
        print('Пароль для пользователя {}
        установлен'.format(username)) password_correct = True continue
    password = input('Введите пароль еще раз: ')
```

Тут выход из цикла выполнятся с помощью проверки флага password_correct. Когда был вве-

ден правильный пароль, флаг выставляется равным True, и с помощью continue выполняется переход в начало цикла, перескочив последнюю строку с запросом пароля.

Результат выполнения будет таким:

```
$ python check_password_with_while_continue.py
Введите имя пользователя: nata
Введите пароль: nata12
Пароль слишком короткий

Введите пароль еще раз: natalksdjflsdjf
Пароль содержит имя пользователя

Введите пароль еще раз: asdfsujljhdflaskjdfh
Пароль для пользователя nata установлен
```

Оператор pass

Оператор `pass` ничего не делает. Фактически, это такая заглушка для объектов.

Например, `pass` может помочь в ситуации, когда нужно прописать структуру скрипта. Его можно ставить в циклах, функциях, классах. И это не будет влиять на исполнение кода.

Пример использования pass:

```
In [6]: for num in range(5):
   ....:     if num < 3:
   ....:         pass
   ....:     else: ....:
       print(num)
   ....:
3
4
```

for/else, while/else

В циклах for и while опционально может использоваться блок else.

for/else

В цикле for:

- блок else выполняется в том случае, если цикл завершил итерацию списка
- но else **не выполняется**, если в цикле был выполнен break

Пример цикла for с else (блок else выполняется после завершения цикла for):

```
In [1]: for num in
   range(5): ....:
      print(num) ....:
   else:
   ....:     print("Числа закончились")
   ....:
0
1
2
3
4
Числа закончились
```

Пример цикла for с else и break в цикле (из-за break блок else не выполняется):

```
In [2]: for num in range(5):
   ....:     if num == 3:
   ....:         break
   ....:     else: ....:
      print(num)
   ....: else:
   ....:     print("Числа закончились")
   ....:
0
1
2
```

Пример цикла for с else и continue в цикле (continue не влияет на блок else):

```
In [3]: for num in range(5):
   ....:     if num == 3:
   ....:     continue ....:
      else: ....:
      print(num)
   ....: else:
   ....:     print("Числа закончились")
   ....:
0
1
2
4
Числа закончились
```

while/else

В цикле while:

- блок else выполняется в том случае, если цикл завершил итерацию списка
- но else **не выполняется**, если в цикле был выполнен break

Пример цикла while с else (блок else выполняется после завершения цикла while):

```
In [4]: i = 0
In [5]: while i < 5:
   ....:     print(i)
   ....:     i += 1
   ....: else:
   ....:     print("Конец")
   ....:
0
1
2
3
4
Конец
```

Пример цикла while с else и break в цикле (из-за break блок else не выполняется):

```
In [6]: i = 0

In [7]: while i < 5:
   ....:     if i == 3:
   ....:         break
   ....:     else:
   ....:         print(i)
   ....:         i += 1
   ....: else:
   ....:     print("Конец")
   ....:
0
1
2
```

Работа с исключениями try/except/else/finally

try/except

Если вы повторяли примеры, которые использовались ранее, то наверняка были ситуации, когда выскакивала ошибка. Скорее всего, это была ошибка синтаксиса, когда не хватало, например, двоеточия.

Как правило, Python довольно понятно реагирует на подобные ошибки, и их можно исправить.

Тем не менее, даже если код синтаксически написан правильно, могут возникать ошибки. В Python эти ошибки называются **исключения (exceptions)**.

Примеры исключений:

```
In [1]: 2/0
---------------------------------------------------
ZeroDivisionError: division by zero

In [2]: 'test' + 2 ------------------------------------------------------
TypeError: must be str, not int
```

В данном случае возникло два исключения: **ZeroDivisionError** и **TypeError**.

Чаще всего можно предсказать, какого рода исключения возникнут во время исполнения программы.

Например, если программа на вход ожидает два числа, а на выходе выдает их сумму, а пользователь ввел вместо одного из чисел строку, появится ошибка TypeError, как в примере выше.

Python позволяет работать с исключениями. Их можно перехватывать и выполнять определенные действия в том случае, если возникло исключение.

Примечание: Когда в программе возникает исключение, она сразу завершает работу.

Для работы с исключениями используется конструкция `try/except`:

```
In [3]: try:
   ...:     2/0
   ...: except ZeroDivisionError:
   ...:     print("You can't divide by zero")
   ...:
You can't divide by zero
```

Конструкция try работает таким образом:

- сначала выполняются выражения, которые записаны в блоке try
- если при выполнения блока try не возникло никаких исключений, блок except пропускается, и выполняется дальнейший код
- если во время выполнения блока try в каком-то месте возникло исключение, оставшаяся часть блока try пропускается
  - если в блоке except указано исключение, которое возникло, выполняется код в блоке except
  - если исключение, которое возникло, не указано в блоке except, выполнение программы прерывается и выдается ошибка

Обратите внимание, что строка `Cool!` в блоке try не выводится:

```
In [4]: try:
   ...:     print("Let's divide some numbers")
   ...:     2/0
   ...:     print('Cool!')
   ...: except ZeroDivisionError:
   ...:     print("You can't divide by zero")
   ...:
Let's divide some numbers
You can't divide by zero
```

В конструкции try/except может быть много except, если нужны разные действия в зависимости от типа ошибки.

Например, скрипт divide.py делит два числа введенных пользователем:

```
# -*- coding: utf-8 -*-

try:
    a = input("Введите первое число: ")
    b = input("Введите второе число: ")
    print("Результат: ", int(a)/int(b))
except ValueError:
    print("Пожалуйста, вводите только числа")
except ZeroDivisionError:
    print("На ноль делить нельзя")
```

Примеры выполнения скрипта:

```
$ python divide.py
Введите первое число: 3
Введите второе число: 1
Результат: 3

$ python divide.py
Введите первое число: 5
Введите второе число: 0
На ноль делить нельзя

$ python divide.py
Введите первое число: qewr
Введите второе число: 3
Пожалуйста, вводите только числа
```

В данном случае исключение **ValueError** возникает, когда пользователь ввел строку вместо числа, во время перевода строки в число.

Исключение ZeroDivisionError возникает в случае, если второе число было равным 0.

Если нет необходимости выводить различные сообщения на ошибки ValueError и ZeroDivisionError, можно сделать так (файл divide_ver2.py):

```
# -*- coding: utf-8 -*-

try:
    a = input("Введите первое число: ") b = input("Введите второе число: ")
print("Результат: ", int(a)/int(b)) except (ValueError, ZeroDivisionError):
print("Что-то пошло не так...")
```

Проверка:

```
$ python divide_ver2.py
Введите первое число: wer
Введите второе число: 4 Что-то пошло не так...

$ python divide_ver2.py
```

```
Введите первое число: 5
Введите второе число: 0 Что-то пошло не так...
```

Примечание: В блоке except можно не указывать конкретное исключение или исключения.

В таком случае будут перехватываться все исключения.

Это делать не рекомендуется!

---

try/except/else

В конструкции try/except есть опциональный блок else. Он выполняется в том случае, если не было исключения.

Например, если необходимо выполнять в дальнейшем какие-то операции с данными, которые ввел пользователь, можно записать их в блоке else (файл divide_ver3.py):

```
# -*- coding: utf-8 -*-

try:
    a = input("Введите первое число: ")
    b = input("Введите второе число: ")
    result = int(a)/int(b)
except (ValueError, ZeroDivisionError):
    print("Что-то пошло не так...")
else:
    print("Результат в квадрате: ", result**2)
```

Пример выполнения:

```
$ python divide_ver3.py
Введите первое число: 10
Введите второе число: 2
Результат в квадрате: 25

$ python divide_ver3.py
Введите первое число: werq
Введите второе число: 3 Что-то пошло не так...
```

try/except/finally

Блок finally - это еще один опциональный блок в конструкции try. Он выполняется **всегда**, независимо от того, было ли исключение или нет.

Сюда ставятся действия, которые надо выполнить в любом случае. Например, это может быть закрытие файла.

Файл divide_ver4.py с блоком finally:

```
# -*- coding: utf-8 -*-

try:
    a = input("Введите первое число: ")
```

(continues on next page)
(продолжение с предыдущей страницы)

```
    b = input("Введите второе число: ")
    result = int(a)/int(b)
except (ValueError, ZeroDivisionError):
    print("Что-то пошло не так...")
else: print("Результат в квадрате: ",
    result**2)
finally:
    print("Вот и сказочке конец, а кто слушал - молодец.")
```

Проверка:

```
$ python divide_ver4.py
Введите первое число: 10
Введите второе число: 2
Результат в квадрате: 25
Вот и сказочке конец, а кто слушал - молодец.

$ python divide_ver4.py
Введите первое число: qwerewr
Введите второе число: 3 Что-
то пошло не так...
Вот и сказочке конец, а кто слушал - молодец.

$ python divide_ver4.py
Введите первое число: 4
Введите второе число: 0 Что-
то пошло не так...
Вот и сказочке конец, а кто слушал - молодец.
```

Когда использовать исключения

Как правило, один и тот же код можно написать и с использованием исключений, и без них.

Например, этот вариант кода:

```
while True:
    a = input("Введите число: ") b =
    input("Введите второе число: ")
    try:
        result = int(a)/int(b)
    except ValueError:
        print("Поддерживаются только числа")
    except ZeroDivisionError:
```

(continues on next page)

(продолжение с предыдущей страницы)

```
        print("На ноль делить нельзя")
    else:
        print(result)
        break
```

Можно переписать таким образом без try/except (файл try_except_divide.py):

```
while True:
    a = input("Введите число: ") b =
    input("Введите второе число: ") if
    a.isdigit() and b.isdigit():
        if int(b) == 0:
            print("На ноль делить нельзя")
        else:
            print(int(a)/int(b))
            break
    else:
        print("Поддерживаются только числа")
```

Но далеко не всегда аналогичный вариант без использования исключений будет простым и понятным.

Важно в каждой конкретной ситуации оценивать, какой вариант кода более понятный, компактный и универсальный - с исключениями или без.

Если вы раньше использовали какой-то другой язык программирования, есть вероятность, что в нём использование исключений считалось плохим тоном. В Python это не так. Чтобы немного больше разобраться с этим вопросом, посмотрите ссылки на дополнительные материалы в конце этого раздела.

Дополнительные материалы

Документация:

- Compound statements (if, while, for, try)
- break, continue
- Errors and Exceptions
- Built-in Exceptions

Статьи:

- Write Cleaner Python: Use Exceptions
- Robust exception handling
- Python Exception Handling Techniques

Stack Overflow:

- Why does python use „else“ after for and while loops?
- Is it a good practice to use try-except-else in Python?

Задания

Задание 6.1

Список mac содержит MAC-адреса в формате XXXX:XXXX:XXXX. Однако, в оборудовании cisco MAC-адреса используются в формате XXXX.XXXX.XXXX.

Создать скрипт, который преобразует MAC-адреса в формат cisco и добавляет их в новый список mac_cisco

Ограничение: Все задания надо выполнять используя только пройденные темы.

```
mac = ['aabb:cc80:7000', 'aabb:dd80:7340', 'aabb:ee80:7000', 'aabb:ff80:7000']
```

Задание 6.2

1. Запросить у пользователя ввод IP-адреса в формате 10.0.1.1
2. Определить тип IP-адреса.
3. В зависимости от типа адреса, вывести на стандартный поток вывода:
    - „unicast“ - если первый байт в диапазоне 1-223
    - „multicast“ - если первый байт в диапазоне 224-239
    - „local broadcast“ - если IP-адрес равен 255.255.255.255
    - „unassigned“ - если IP-адрес равен 0.0.0.0
    - „unused“ - во всех остальных случаях

Ограничение: Все задания надо выполнять используя только пройденные темы.

Задание 6.2a

Сделать копию скрипта задания 6.2.

Добавить проверку введенного IP-адреса. Адрес считается корректно заданным, если он:

- состоит из 4 чисел разделенных точкой,
- каждое число в диапазоне от 0 до 255.

Если адрес задан неправильно, выводить сообщение: „Неправильный IP-адрес“

Ограничение: Все задания выполнять используя только пройденные темы.

Задание 6.2b

Сделать копию скрипта задания 6.2a.

Дополнить скрипт: Если адрес был введен неправильно, запросить адрес снова.

Ограничение: Все задания надо выполнять используя только пройденные темы.

Задание 6.3

В скрипте сделан генератор конфигурации для access-портов.

Сделать аналогичный генератор конфигурации для портов trunk.

В транках ситуация усложняется тем, что VLANов может быть много, и надо понимать, что с ним делать.

Поэтому в соответствии каждому порту стоит список и первый (нулевой) элемент списка указывает как воспринимать номера VLAN, которые идут дальше:

- add - VLANы надо будет добавить (команда switchport trunk allowed vlan add 10,20)
- del - VLANы надо удалить из списка разрешенных (команда switchport trunk allowed vlan remove 17)
- only - на интерфейсе должны остаться разрешенными только указанные VLANы (команда switchport trunk allowed vlan 11,30)

Задача для портов 0/1, 0/2, 0/4:

- сгенерировать конфигурацию на основе шаблона trunk_template
- с учетом ключевых слов add, del, only

Код не должен привязываться к конкретным номерам портов. То есть, если в словаре trunk будут другие номера интерфейсов, код должен работать.

Ограничение: Все задания выполнять используя только пройденные темы.

```
access_template = [
    'switchport mode access', 'switchport access vlan',
    'spanning-tree portfast', 'spanning-tree bpduguard enable'
]

trunk_template = [
    'switchport trunk encapsulation dot1q', 'switchport mode trunk',
    'switchport trunk allowed vlan'
]

access = {
    '0/12': '10',
    '0/14': '11',
    '0/16': '17',
    '0/17': '150'
}
trunk = {
        '0/1': ['add', '10', '20'],
        '0/2': ['only', '11', '30'],
        '0/4': ['del', '17']
    }

for intf, vlan in access.items():
    print('interface FastEthernet' + intf)
    for command in access_template:
        if command.endswith('access vlan'):
            print(' {} {}'.format(command, vlan))
        else:
            print(' {}'.format(command))
```