Cheatography

Python Cheat Sheet

by brianLane via cheatography.com/35375/cs/11124/

Строки

Строка не мутабельный тип

Наличие подстроки в строке

```
string = 'Python'
'Py' in string                  True
```

Слайсинг

```
string = 'I like Python!'       14 characters

string[0]                       'I'
string[:1]                      'I'
string[0:12]                    'I like Pytho'
string[:-5]                     'I like Py'
string[:14]                     'I like Python!'
string[:]                       'I like Python!'
string[2:]                      'like Python!'
```

Репликация

```
"Alice" * 3                     "AliceAliceAlice"
```

Проход по строке в цикле

```
for i in "Python":
```

Raw Strings

You can place an r before the beginning quotation mark of a string to make it a raw string. A raw string completely ignores all escape characters and prints any backslash that appears in the string:

```
>>>print(r'That is Carol\'s cat.')
That is Carol\'s cat.
```

Multiline Strings with triple Quotes

Строки (cont)

```
>>>print('''Dear Alice,

I love you!

Sincerely,
Bob''')

Dear Alice,

I love you!

Sincerely,
Bob
```

Списки

List is reference type

```
a = [1, 2]
b = a
a.append(3)
print(b)                        [1, 2, 3]

def add(some_list):
  some_list.append("end")
add(a)
print(a)                        [1, 2, 3, "end"]

import copy
copy.copy(a)
copy.deepcopy(a)                если список содержит вложенные списки
```

Создание пустого списка

```
a = []
a = list()
```

Список из кортежа

```
list(('cat', 'dog', 5))         ['cat', 'dog', 5]
```

Репликация

```
[5, 1] * 3                      [5, 1, 5, 1, 5, 1]
```

Конкатенация

Списки (cont)

```
a = [1, 2, 3]
b = [3, 4, 5]
```

a.extend(b)	[1, 2, 3, 3, 4, 5]
a += b	[1, 2, 3, 3, 4, 5]

Сортировка

```
a = [3, 2, 6, 9, 8]
```

a.sort()	[2, 3, 6, 8, 9]
a.sort(reverse=True)	[9, 8, 6, 3, 2]

```
a = ['a', 'z', 'A', 'Z']
```

a.sort()	['A', 'Z', 'a', 'z']
a.sort(key=str.lower)	['A', 'a', 'Z', 'z']

This causes the sort() function to treat all the items in the list as if they were lowercase without actually changing the values in the list.

Отрицательный индекс

```
a = [5, 11]
```

a[-1]	11

Кол-во элементов списка

```
a = [3, 3, 3, 3]
```

len(a)	4

Наличие элемента в списке

```
a = [22, 13]
```

13 in a	True

Удаление элемента списка

```
a = ["cat", "dog", "cat"]
```

a.remove("cat")	["dog", "cat"]
del a[1]	["dog"]

Извлечение значений списка

```
a = [3, 2, 1]
x, y, z = a
```

print(x, y, z)	3 2 1

Узнать индекс элемента

```
a = [3, 5, 2, 5]
```

a.index(5)	1

Добавление элемента

```
a = [1, 2]
```

a.append("end")	[1, 2, "end"]
a.insert(1, 5)	[1, 5, 2, "end"]

Tuples

Кортеж не мутабельный, в отличие от списка

You can use tuples to convey to anyone reading your code that you don't intend for that sequence of values to change. If you need an ordered sequence of values that never changes, use a tuple. A second bene t of using tuples instead of lists is that, because they are immutable and their contents don't change, Python can implement some optimizations that make code using tuples slightly faster than code using lists.

Создание кортежа из одного элемента

Tuples (cont)

```
a = (1,)
type(a)                    <class 'tuple'>

a = (1)
type(a)                    <class 'int'>
```

Кортеж из списка

```
a = tuple([1, 2, 3])
```

Слайсинг

```
a = (1, 2, 3, 4, 5)
a[0:3]                     (1, 2, 3)
```

Multiple-assignment trick

```
>>> a = (1, 2, 3)
>>> x, y, z = a
>>> y
2
```

Dictionaries

Ключи должны быть иммутабельными типами

Создание пустого словаря

```
a = {}
a = dict()
```

Наличие элемента в словаре

```
a = {"one":1, "two":2}

>>>"one" in a
True

>>>3 not in a.values()
True

>>>a.setdefault("three", 3)
3
```

- если в **a** есть ключ "three", то возвращает значение по этому ключу, иначе создает пару {"three":3} и возвращает 3

Извлечение элементов словаря

Dictionaries (cont)

```
a = {'one':1, 'two':2}

>>>a.get("three", 0)
0
```

- безопасное извлечение элемента: если такого ключа нет, то вернется значение второго аргумента get

```
>>>a.items()
dict_items([('two', 2), ('one', 1)])

for k, v in a.items():
  ...

>>>a.values()
dict_values([2, 1])

>>>a.keys()
dict_keys(['one', 'two'])

>>>list(a.keys())
['one', 'two']
```

In Python 2, the keys method returns a list. But in Python 3, it returns a view object. This gives the developer the ability to update the dictionary and the view will automatically update too.

Pretty Printing

```
import pprint

a = {'one':1, 'two':2}
pprint.pprint(a)
```

If you want to obtain the pretti ed text as a string value instead of displaying it on the screen, call pprint.pformat() instead:

```
print(pprint.pformat(a))
```

Conditional & Loop Statements

```
if
elif
else
```

>, >=, ==, !=, or, and, not

```
if x not in my_list:
  ...
```

0, 0.0, (), [], "", None = False

break	выход из цикла
continue	пропустить итерацию

Conditional & Loop Statements (cont)

for i in range(5): print(i)	0, 1, 2, 3, 4
for i in range(5, 10): print(i)	5,6,7,8,9
for i in range(5, 10, 2): print(i)	5, 7, 9
for i in range(5, -1, -1): print(i)	5, 4, 3, 2, 1, 0

Regular Expressions

Модуль регэкспов

import re

-

Regular Expressions (cont)

\d - stands for a digit character

333-333-4444:
\d\d\d-\d\d\d-\d\d\d\d
or:
\d{3}-\d{3}-\d{4}

Passing a string value representing your regular expression to **re.compile()** returns a Regex pattern object (or simply, a Regex object). To create a Regex object that matches the phone number pattern, enter:

>>>phoneNumRegex = re.compile(r'\d{3}-\d{3}-\d{4}')
- by putting an **r** before the first quote of the string value, you can mark the string as a raw string, which does not escape characters.

>>>mo = phoneNumRegex.search('My number is 415-555-4242')
>>>mo.group()
'415-555-4242'

A Regex object's **search()** method searches the string it is passed for any matches to the regex. The **search()** method will return **None** if the regex pattern is not found in the string. If the pattern is found, the **search()** method returns a **Match object**. **Match objects** have a **group()** method that will return the actual matched text from the searched string.

Grouping with Parentheses

Regular Expressions (cont)

Say you want to separate the area code from the rest of the phone number. Adding parentheses will create groups in the regex: (\d\d\d)-(\d\d\d-\d\d\d\d). Then you can use the group() match object method to grab the matching text from just one group.

>>>phoneNumRegex = re.compile(r'(\d{3})-(\d{3}-\d{4})')
>>>mo = phoneNumRegex.search('My number is 431-559-2243')
>>>mo.group(1)
'431'
>>>mo.group(2)
'559-2243'
>>>mo.group(0) or **>>>mo.group()**
'431-559-2243'
>>>mo.groups()
('431', '559-2243')

>>>areaCode, mainNumber = mo.groups()
>>>areaCode
'431'

Parentheses have a special meaning in regular expressions, but what do you do if you need to match a parenthesis in your text? In this case, you need to escape the (and) characters with a back-slash:

>>>>>>phoneNumRegex = re.compile(r'(\(\d{3}\)) (\d{3}-\d{4})')
>>>mo = phoneNumRegex.search('My phone number is (415) 555-4242.')
>>>mo.group(1)
'(415)'

Matching Multiple Groups with the Pipe

Regular Expressions (cont)

The | character is called a pipe. You can use it anywhere you want to match one of many expressions. For example, the regular expression r'Batman|Tina Fey' will match either 'Batman' or 'Tina Fey'.
When both Batman and Tina Fey occur in the searched string, the first occurrence of matching text will be returned as the Match object:

>>>heroRegex = re.compile(r'Batman|Tina Fey')
>>>mo1 = heroRegex.search('Batman and Tina Fey.')
>>>mo1.group()
'Batman'

>>>mo2 = heroRegex.search('Tina Fey and Batman.')
>>>mo2.group()
'Tina Fey'

You can also use the pipe to match one of several patterns as part of your regex. For example, say you wanted to match any of the strings 'Batman', 'Batmobile', 'Batcopter', and 'Batbat'. Since all these strings start with Bat, it would be nice if you could specify that prefix only once:

>>>batRegex = re.compile(r'Bat(man|mobile|copter|bat)')
>>>mo = batRegex.search('Batmobile lost a wheel')
>>>mo.group()
'Batmobile'
>>>mo.group(1)
'mobile'

If you need to match an actual pipe character, escape it with a back-slash, like \|.

.findall()

Regular Expressions (cont)

```
>>>heroRegex = re.compile(r'Batman|Tina Fey')
>>>heroRegex.findall('Batman and Tina Fey.')
['Batman', 'Tina Fey']
```

In addition to the **search()** method, **Regex objects** also have a **findall()** method. While **search()** will return a **Match object** of the first matched text in the searched string, the **findall()** method will return the strings of every match in the searched string.

```
>>>phoneNumRegex = re.compile(r'\d\d\d-\d\d\d-\d\d\d\d')
>>>mo = phoneNumRegex.search('Cell: 415-555-9999 Work: 212-555-0000')
>>>mo.group()
'415-555-9999'
```

On the other hand, **findall()** will not return a **Match object** but a list of strings — *as long as there are no groups in the regular expression*

```
>>>phoneNumRegex = re.compile(r'\d{3}-\d{3}-\d{4}') # has no groups
>>>phoneNumRegex.findall('Cell: 415-555-9999 Work: 212-555-0000')
['415-555-9999', '212-555-0000']
```

If there are groups in the regular expression, then **findall()** will return a list of tuples. Each tuple represents a found match, and its items are the matched strings for each group in the regex. Notice that the regular expression being compiled now has groups in parentheses:

```
>>>phoneNumRegex = re.compile(r'(\d\d\d)-(\d\d\d)-(\d\d\d\d)') # has groups
>>>phoneNumRegex.findall('Cell: 415-555-9999 Work: 212-555-0000')
[('415', '555', '1122'), ('212', '555', '0000')]
```

Optional Matching with the Question Mark

Regular Expressions (cont)

Sometimes there is a pattern that you want to match only optionally. That is, the regex should find a match whether or not that bit of text is there. The ? character flags the group that precedes it as an optional part of the pattern:

```
>>>batRegex = re.compile(r'Bat(wo)?man')
>>>mo1 = batRegex.search('The Adventures of Batman')
>>>mo1.group()
'Batman'
>>>mo2 = batRegex.search('The Adventures of Batwoman')
>>>mo2.group()
'Batwoman'
```

or:

```
>>>phoneRegex = re.compile(r'(\d\d\d-)?\d\d\d-\d\d\d\d')
>>>mo1 = phoneRegex.search('My number is 415-555-4242')
>>>mo1.group()
'415-555-4242'
>>>mo2 = phoneRegex.search('My number is 555-4242')
>>>mo2.group()
'555-4242'
```

You can think of the ? as saying, “Match zero or one of the group preceding this question mark.”
If you need to match an actual question mark character, escape it with \?.

Matching Zero or More with the Star

The * (called the star or asterisk) means “match zero or more”—the group that precedes the star can occur any number of times in the text. It can be completely absent or repeated over and over again:

```
>>>batRegex = re.compile(r'Bat(wo)*man')
>>>mo1 = batRegex.search('The Adventures of Batman')
>>>mo1.group()
'Batman'
>>>mo2 = batRegex.search('The Adventures of Batwoman')
>>>mo2.group()
'Batwoman'
>>>mo3 = batRegex.search('The Adventures of Batwowowowoman')
>>>mo3.group()
'Batwowowowoman'
```

Matching One or More with the Plus

Regular Expressions (cont)

While * means “match zero or more”, the + (or plus) means “match one or more.” Unlike the star, which does not require its group to appear in the matched string, the group preceding a plus must appear at least once. It is not optional:

```
>>>batRegex = re.compile(r'Bat(wo)+man')
>>>mo1 = batRegex.search('The Adventures of Batwoman')
>>>mo1.group()
'Batwoman'
>>>mo2 = batRegex.search('The Adventures of Batwowowowoman')
>>>mo2.group()
'Batwowowowoman'
>>>mo3 = batRegex.search('The Adventures of Batman')
>>>mo3 == None
True
```

If you need to match an actual plus sign character, prefix the plus sign with a backslash to escape it: \+.

Matching Specific Repetitions with Curly Brackets

If you have a group that you want to repeat a specific number of times, follow the group in your regex with a number in curly brackets. For example, the regex (Ha){3} will match the string 'HaHaHa', but it will not match 'HaHa', since the latter has only two repeats of the (Ha) group. Instead of one number, you can specify a range by writing a minimum, a comma, and a maximum in between the curly brackets. For example, the regex (Ha){3,5} will match 'HaHaHa', 'HaHaHaHa', and 'HaHaHaHaHa'. You can also leave out the first or second number in the curly brackets to leave the minimum or maximum unbounded. For example, (Ha){3,} will match three or more instances of the (Ha) group, while (Ha){,5} will match zero to five instances. Curly brackets can help make your regular expressions shorter.

Greedy and Nongreedy Matching

Regular Expressions (cont)

Since **(Ha){3,5}** can match three, four, or five instances of **Ha** in the string **'HaHaHaHaHa'**, you may wonder why the **Match object**’s call to **group()** returns **'HaHaHaHaHa'** instead of the shorter possibilities. After all, **'HaHaHa'** and **'HaHaHaHa'** are also valid matches of the regular expression **(Ha){3,5}**.

Python’s regular expressions are greedy by default, which means that in ambiguous situations they will match the longest string possible. The non-greedy version of the curly brackets, which matches the shortest string pos- sible, has the closing curly bracket followed by a question mark.

```
>>>greedyHaRegex = re.compile(r'(Ha){3,5}')
>>>mo1 = greedyHaRegex.search('HaHaHaHaHa')
>>>mo1.group()
'HaHaHaHaHa'

>>>nongreedyHaRegex = re.compile(r'(Ha){3,5}?')
>>>mo2 = nongreedyHaRegex.search('HaHaHaHaHa')
>>>mo2.group()
'HaHaHa'
```

Note that the question mark can have two meanings in regular expres- sions: declaring a nongreedy match or flagging an optional group. These meanings are entirely unrelated.

Character Classes

Regular Expressions (cont)

\d is shorthand for the regular expression (0|1|2|3|4|5|6|7|8|9)

\d
- any numeric digit from 0 to 9

\D
- any character that is **not** a numeric digit from 0 to 9

\w
- any letter, numeric digit, or the underscore character.

\W
- any character that is **not** a letter, numeric digit, or the underscore character.

\s
- any space, tab, or newline character.

\S
- any character that is **not** a space, tab, or newline.

[0-5]
- this character class will match only the numbers 0 to 5; this is much shorter than typing (0|1|2|3|4|5).

>>>xmasRegex = re.compile(r'\d+\s\w+')
>>>xmasRegex.findall('12 drummers, 110 pipers, 1 lord')
['12 drummers', '110 pipers', '1 lord']

Making Your Own Character Classes

Regular Expressions (cont)

There are times when you want to match a set of characters but the shorthand character classes (\d, \w, \s, and so on) are too broad. You can define your own character class using square brackets.

>>>vowelRegex = re.compile(r'[aeiouAEIOU]')
>>>vowelRegex.findall('RoboCop eats baby food. BABY FOOD.')
['o', 'o', 'o', 'e', 'a', 'a', 'o', 'o', 'A', 'O', 'O']

You can also include ranges of letters or numbers by using a hyphen. For example, the character class **[a-zA-Z0-9]** will match all lowercase letters, uppercase letters, and numbers.

Note that inside the square brackets, the normal regular expression symbols are not interpreted as such. This means you do not need to escape the ., *, ?, or () characters with a preceding backslash. For example, the character class **[0-5.]** will match digits 0 to 5 and a period. You do not need to write it as **[0-5\.]**.

By placing a caret character (^) just after the character class's opening bracket, you can make a negative character class. A negative character class will match all the characters that are not in the character class.

>>>consonantRegex = re.compile(r'[^aeiouAEIOU]')
>>>consonantRegex.findall('RoboCop eats baby food. BABY FOOD.')
['R', 'b', 'c', 'p', ' ', 't', 's', ' ', 'b', 'b', 'y', ' ', 'f', 'd', '.', ' ', 'B', 'B', 'Y', ' ', 'F', 'D', '.']

The Caret and Dollar Sign Characters

Regular Expressions (cont)

r'^Hello' - matches strings that begin with 'Hello', like 'Hello bla' but not 'bla Hello'

r'\d$' - matches strings that end with a numeric character from 0 to 9, like 'bla3' but not '3bla'

r'^\d+$' - matches strings that completely consist of numbers, like '123456' but not '123abc456'

"Carrots cost dollars" - the caret comes first and the dollar sign comes last.

The Wildcard Character

The . (or dot) character in a regular expression is called a wildcard and will match any character except for a newline.

>>>atRegex = re.compile(r'.at')
>>>atRegex.findall('The cat in the hat sat on the flat mat.')
['cat', 'hat', 'sat', 'lat', 'mat']

To match an actual dot, escape the dot with a backslash: \.

Matching Everything with Dot-Star

.* - stands for anything
Dot character means “any single character except the newline”
The star character means “zero or more of the preceding character”

The dot-star uses *greedy* mode: It will always try to match as much text as possible. To match any and all text in a *nongreedy* fashion, use the dot, star, and question mark (.*?):

>>>nongreedyRegex = re.compile(r'<.*?>')
>>>mo = nongreedyRegex.search('<To serve man> for dinner.>')
>>>mo.group()
'<To serve man>'

>>>greedyRegex = re.compile(r'<.*>')
>>>mo = greedyRegex.search('<To serve man> for dinner.>')
mo.group()
'<To serve man> for dinner.>'

Matching Newlines with the Dot Character

Regular Expressions (cont)

The dot-star will match everything except a newline. By passing **re.DOTALL** as the second argument to **re.compile()**, you can make the dot character match all characters, including the newline character.

>>>newlineRegex = re.compile('.*', re.DOTALL)
>>>newlineRegex.search('Bla-bla.\nBla-bla.').group()
'Bla-bla.\nBla-bla.'

Case-Insensitive Matching

re.IGNORECASE or **re.I** as a second argument to **re.compile()**

Substituting Strings with the sub() Method

>>>namesRegex = re.compile(r'Agent \w+')
>>>namesRegex.sub('CENSORED', 'Agent Alice gave the secret documents to Agent Bob.')
'CENSORED gave the secret documents to CENSORED.'

Sometimes you may need to use the matched text itself as part of the substitution. In the first argument to **sub()**, you can type **\1**, **\2**, **\3**, and so on, to mean “Enter the text of group 1, 2, 3, and so on, in the substitution.”

>>>agentNamesRegex = re.compile(r'Agent (\w)\w*')
>>>agentNamesRegex.sub(r'\1**', 'Agent Alice told Agent Carol that Agent Eve knew Agent Bob was a double agent.')**
A**** told C**** that E**** knew B**** was a double agent.'

Managing Complex Regexes

Regular expressions are fine if the text pattern you need to match is simple. But matching complicated text patterns might require long, convoluted regular expressions. You can mitigate this by telling the **re.compile()** function to ignore whitespace and comments inside the regular expression string. This “verbose mode” can be enabled by passing the variable **re.VERBOSE** as the second argument to **re.compile()**:

Regular Expressions (cont)

```
phoneRegex = re.compile(r'''(
    (\d{3}|\(\d{3}\))?          # area code
    (\s|-|\.)?                  # separator
    (\s*(ext|x|ext.)\s*\d{2,5})?  # extension
    \d{4}                       # last 4 digits
)''', re.VERBOSE)
```

Combining re.IGNORECASE, re.DOTALL, and re.VERBOSE

```
>>>someRegexValue = re.compile('foo', re.IGNORECASE |
re.DOTALL | re.VERBOSE)
```

String Methods

Длина строки

len(**string**)

Заглавные, строчные

string = 'aLpHa4'

>>>string.upper()
'ALPHA4'
>>>string.isupper()
True

>>>string.lower()
'alpha4'
>>>string.islower()
True

>>>string = **'4'** or **'Hello'**
>>>string.islower() or **isupper()**
False

The upper() and lower() methods are helpful if you need to make a case-insensitive comparison. The strings 'great' and 'GREat' are not equal to each other. But in the following small program, it does not matter whether the user types Great, GREAT, or grEAT, because the string is first converted to lowercase.

Все методы типа String в виде списка строк

dir(**string**)

String Methods (cont)

Справка по методу

help(**string**.capitalize)

Уникальный ID объекта

id(**string**)

The isX String Methods

.isalpha()
returns True if the string consists only of letters and is not blank.
'hello' - True
'hello123' - False

.isalnum()
returns True if the string consists only of letters and numbers and is not blank.
'hello123', 'hello', '123' - True

.isdecimal()
returns True if the string consists only of numeric characters and is not blank.
'123' - True

.isspace()
returns True if the string consists only of spaces, tabs, and new- lines and is not blank.
' '- True

.istitle()
returns True if the string consists only of words that begin with an uppercase letter followed by only lowercase letters.
'This Is Title Case'
'This Is Title Case 123'- True

'This Is not Title Case'
'This Is NOT Title Case Either' - False

Содержит начало/конец?

>>>'Hello world!'.startswith('Hello')
True
>>>'Hello world!'.endswith('world!')
True

Объединение/разбиение строк

String Methods (cont)

```
>>>' | '.join(['cats', 'rats', 'bats'])
'cats | rats | bats'

>>>'My name is Clark'.split()
['My', 'name', 'is', 'Clark']
>>>'MyABCnameABCisABCClark'.split('ABC')
['My', 'name', 'is', 'Clark']
>>>'My name is Clark'.split('m')
['My na', 'e is Clark']
```

A common use of split() is to split a multiline string along the newline characters:

```
>>>str = "'Dear Alice,
bla-bla
Sincerely,
Bob"'
>>>str.split('\n')
['Dear Alice,', 'bla-bla', 'Sincerely,', 'Bob']
```

Выравнивание текста

```
>>>'Hello'.rjust(10)
'      Hello'
>>>'Hello'.rjust(10, '*')
'*****Hello'

>>>'Hello'.ljust(10)
'Hello      '

>>>'Hello'.center(15)
'     Hello     '
>>>'Hello'.center(15, '=')
'=====Hello====='
```

Удаление пробелов

```
>>>'    Hello World    '.strip()
'Hello World'
>>>'    Hello World    '.lstrip()
'Hello World    '
>>'    Hello World    '.rstrip()
'    Hello World'

>>'SpamSpamBaconSpamEggsSpamSpam'.strip('ampS')
'BaconSpamEggs'
```

Passing strip() the argument 'ampS' will tell it to strip occurences of a, m, p, and capital S from the ends of the string stored in spam. The order of the characters in the string passed to strip() does not matter: strip('ampS') will do the same thing as strip('mapS') or strip('Spam').

Доступ к буферу обмена

String Methods (cont)

The pyperclip module has copy() and paste() functions that can send text to and receive text from your computer's clipboard. Sending the output of your program to the clipboard will make it easy to paste it to an email, word processor, or some other software. Pyperclip does not come with Python. To install it, follow the directions for installing third-party modules in Appendix A.

```
>>>import pyperclip
>>>pyperclip.copy('Hello world')
>>>pyperclip.paste()
'Hello world'
```

Of course, if something outside of your program changes the clipboard contents, the paste() function will return it.

String Formatting

%i, %f, %s - integer, float, string

var = "cookies" "I like %s" % var	"I like cookies"
"I like %s and %s" % (var, var)	"I like cookies and cookies"
"%.2f" % 1.237	"1.24"
"%(a)s %(a)s" % {"a":"cat"}	"cat cat"
"{1}, {0}, {2}".format("a", "b", "c")	"b, a, c"
xy = {"x":2, "y":5} "{x}, {y}".format(**xy)	"2, 5"

bla

Python 2.x code
a = raw_input("Enter a number: ")

Возвращает строку с введеными данными
Python 2.x actually has a built-in called input as well; however it tries to execute what is entered as a Python expression whereas raw_input returns a string.

Python 3.x code
a = input("Enter a number: ")

Working with Files

Модуль

import os
os.path - module inside the os module
os.path documentation

/ - macOS, Linux; \ - Windows

If you want your programs to work on all operating systems:

>>>os.path.join('usr', 'bin', 'spam')
'usr/bin/spam' (macOS, Linux)
'usr\\bin\\spam' (Windows, \\- escaped backslash)

The Current Working Directory

>>>os.getcwd()
'/Users/Admin/Documents'

Change Directory:
>>>os.chdir('../Movies')

Make Directory:
>>>os.makedirs('./python')

Handling Absolute and Relative Paths

Working with Files (cont)

>>>os.path.abspath('./Anki')
'/Users/Admin/Documents/Anki'

>>>os.path.isabs('..')
False

>>>os.path.relpath('/Users', '/Library/Audio')
'../../Users'

basename - все что после последнего слеша:
>>>os.path.basename('/Users/Admin')
'Admin'
>>>os.path.basename('/Users/Admin/')
''

dirname - до последнего слеша:
>>>os.path.dirname('/Users/Admin')
'Users'
>>>os.path.dirname('/Users/Admin/')
'/Users/Admin'

>>>os.path.split('/home/rob/script.py')
('/home/rob', 'script.py')

>>>os.sep or **>>>os.path.sep**
'/'
>>>'/home/rob/script.py'.split(os.sep)
['', 'home', 'rob', 'script.py']
- on OS X and Linux systems, there will be a blank string at the start of the returned list

Finding File Sizes and Folder Contents

>>>os.path.getsize('/home/rob/text.txt')
149
- размер файла в байтах (для папок не работает)

>>>os.listdir('/Users/Admin/a/')
['.DS_Store', 'c', 'dir-tree.list', 'file.txt', 'jupyter']

>>>os.path.exists('/Users')
True
>>>os.path.isdir('/Users')
True
>>>os.path.isfile('/Users')
False

Open File

Working with Files (cont)

Plaintext files contain only basic text characters and do not include font, size, or color information. Your programs can easily read the contents of plaintext files and treat them as an ordinary string value.

Binary files are files such as word processing documents, PDFs, images, spreadsheets, and executable programs. If you open a binary file in Notepad or TextEdit, it will look like scrambled nonsense.

Open file in *read mode*:
>>>helloFile = open('./hello.txt', 'r')
or
>>>helloFile = open('./hello.txt')
- open() returns a File object. A File object represents a file on your computer; it is simply another type of value in Python, much like the lists and dictionaries. You can call methods on the File object if you want read or write to the file.

Read File

>>>helloContent = helloFile.read()
>>>helloContent
'Hello world!'

>>>sonnetFile = open('sonnet29.txt')
>>>sonnetFile.readlines()
['first line\n', 'second line.']

Note that each of the string values ends with a newline character, \n , except for the last line of the file.

Write File

Working with Files (cont)

Write mode will overwrite the existing file and start from scratch. Append mode will append text to the end of the existing file.

If the filename passed to open() does not exist, both write and append mode will create a new, blank file. After reading or writing a file, call the close() method before opening the file again.

>>>baconFile = open('bacon.txt', 'w')
>>>baconFile.write('Hello world!\n')
13
>>>baconFile.close()
>>>baconFile = open('bacon.txt', 'a')
>>>baconFile.write('Bacon is not a vegetable.')
25
>>>baconFile.close()
>>>baconFile = open('bacon.txt')
>>>content = baconFile.read()
>>>baconFile.close()
>>>print(content)
Hello world!
Bacon is not a vegetable.

Note that the write() method does not automatically add a newline character to the end of the string like the print() function does. You will have to add this character yourself.

Saving Variables with the shelve Module

Working with Files (cont)

The shelve module will let you add Save and Open features to your program. For example, if you ran a program and entered some configuration settings, you could save those settings to a shelf file and then have the program load them the next time it is run.

```
>>>import shelve
>>>shelfFile = shelve.open('mydata')
>>>cats = ['Zophie', 'Pooka']
>>>shelfFile['cats'] = cats
>>>shelfFile.close()
```

After running the previous code on Windows, you will see three new files in the current working directory: mydata.bak, mydata.dat, and mydata.dir. On OS X, only a single mydata.db file will be created. These binary files contain the data you stored in your shelf.

Shelf values don't have to be opened in read or write mode—they can do both once opened.

```
>>>shelfFile = shelve.open('mydata')
>>>type(shelfFile)
<class 'shelve.DbfilenameShelf'>
>>>shelfFile['cats']
['Zophie', 'Pooka']
>>>shelfFile.close()
```

Just like dictionaries, shelf values have keys() and values() methods that will return list-like values of the keys and values in the shelf. Since these methods return list-like values instead of true lists, you should pass them to the list() function to get them in list form:

```
>>>shelfFile = shelve.open('mydata')
>>>list(shelfFile.keys())
['cats']
>>>list(shelfFile.values())
[['Zophie', 'Pooka']]
>>>shelfFile.close()
```

Plaintext is useful for creating files that you'll read in a text editor such as Notepad or TextEdit, but if you want to save data from your Python programs, use the shelve module.

Saving Variables with the pprint.pformat() function

Working with Files (cont)

The pprint.pformat() function returns a string formatted as syntactically correct Python code.

```
>>>import pprint
>>>cats = [{'name': 'Zophie', 'desc': 'chubby'}, {'name': 'Pooka',
'desc': 'fluffy'}]
>>>fileObj = open('myCats.py', 'w')
>>>fileObj.write('cats = ' + pprint.pformat(cats) + '\n')
83
>>>fileObj.close()
```

The modules that an import statement imports are themselves just Python scripts. When the string from pprint.pformat() is saved to a .py file, the file is a module that can be imported just like any other.

```
>>>import myCats
>>>myCats.cats
[{'name': 'Zophie', 'desc': 'chubby'}, {'name': 'Pooka', 'desc': 'fluffy'}]
>>>myCats.cats[0]['name']
'Zophie'
```

The benefit of creating a .py file (as opposed to saving variables with the shelve module) is that because it is a text file, the contents of the file can be read and modified by anyone with a simple text editor. For most applications, however, saving data using the shelve module is the preferred way to save variables to a file. Only basic data types such as integers, floats, strings, lists, and dictionaries can be written to a file as simple text. File objects, for example, cannot be encoded as text.

Folder names and filenames are not case sensitive on Windows and OS X, they are case sensitive on Linux.

random module

import random, sys, os, math

random.randint(1, 10)	random int [1, 10]

from random import *

*With this form of import statement, calls to functions in random will not need the **random.** prefix. However, using the full name makes for more readable code, so it is better to use the normal form of the import statement.*

sys.exit()	досрочное завершение программы

>>>s = list(range(5))
>>>random.shuffle(s)
>>>s
[3, 4, 1, 0, 2]

идф

```
>>>None == print()
True
```

Питон требует чтобы каждая функция возвращала значение и если мы не пишем **return**, то Питон за нас добавляет**return None**.

print("Hello", end="") print("World")	HelloWorld
print("a", "b", "c", sep="/")	a/b/c

```
a = 1
def func():
  global a
  a = 2
func()
print(a)
```

2

идф (cont)

```
def func(d):
  try:
    return 42 / d
  except ZeroDivisionError:
    print("Error: Invalid argument.")
```

Запуск программ в bash

Shebang line

macOS
#! /usr/bin/env python3

Linux
#! /usr/bin/python3

Run

To run your Python programs, save your .py file to your home folder. Then, change the .py file's permissions to make it executable:
chmod +x pythonScript.py

Run your script by entering:
./pythonScript.py

Running Python Programs with Assertions Disabled

You can disable the assert statements in your Python programs for a slight performance improvement. When running Python from the terminal, include the -O switch after python or python3 and before the name of the .py file. This will run an optimized version of your program that skips the assertion checks.

PIP

The pip Tool

PIP (cont)

Beyond the standard library of modules packaged with Python, other developers have written their own modules to extend Python's capabilities even further. The primary way to install third-party modules is to use Python's pip tool. This tool securely downloads and installs Python modules onto your computer from https://pypi.python.org/, the website of the Python Software Foundation. PyPI, or the Python Package Index, is a sort of free app store for Python modules.
The executable file for the pip tool is called pip on Windows and pip3 on OS X and Linux. You can find pip on:
macOS
/Library/Frameworks/Python.framework/Versions/3.4/bin/pip3
Linux
/usr/bin/pip3

While pip comes automatically installed with Python 3.4 on Windows and OS X, you must install it separately on Linux. To install pip3 on:
Ubuntu or Debian Linux:
$ sudo apt-get install python3-pip
Fedora Linux:
$ sudo yum install python3 -pip

Установка модуля

macOS, Linux:
$ sudo pip3 install ModuleName

If you already have the module installed but would like to upgrade it to the latest version available on PyPI, run:
$ pip3 install –U ModuleName