

Python Cheatsheet

Table of Content:

- Python Cheatsheet
 - Python Basics
 - Math Operators
 - Data Types
 - String Concatenation and Replication
 - Variables
 - Comments
 - The print() Function
 - The input() Function
 - The len() Function
 - The str(), int(), and float() Functions
 - Flow Control
 - Comparison Operators
 - Boolean Operators
 - Mixing Boolean and Comparison Operators
 - if Statements
 - else Statements
 - elif Statements
 - while Loop Statements
 - break Statements
 - continue Statements
 - for Loops and the range() Function
 - Importing Modules
 - Ending a Program Early with sys.exit()
 - Functions
 - Return Values and return Statements
 - The None Value
 - Keyword Arguments and print()
 - Local and Global Scope
 - The global Statement
 - Exception Handling
 - Dictionaries and Structuring Data
 - The keys(), values(), and items() Methods
 - Checking Whether a Key or Value Exists in a Dictionary
 - The get() Method
 - The setdefault() Method
 - Pretty Printing
 - Manipulating Strings
 - Escape Characters
 - Raw Strings

- Multiline Strings with Triple Quotes
- Indexing and Slicing Strings
- The in and not in Operators with Strings
- The upper(), lower(), isupper(), and islower() String Methods
- The isX String Methods
- The startswith() and endswith() String Methods
- The join() and split() String Methods
- Justifying Text with rjust(), ljust(), and center()
- Removing Whitespace with strip(),rstrip(), and lstrip()
- Copying and Pasting Strings with the pyperclip Module
- Regular Expressions
 - Matching Regex Objects
 - Grouping with Parentheses
 - Matching Multiple Groups with the Pipe
 - Optional Matching with the Question Mark
 - Matching Zero or More with the Star
 - Matching One or More with the Plus
 - Matching Specific Repetitions with Curly Brackets
 - Greedy and Nongreedy Matching
 - The findall() Method
 - Making Your Own Character Classes
 - The Caret and Dollar Sign Characters
 - The Wildcard Character
 - Matching Everything with Dot-Star
 - Matching Newlines with the Dot Character
 - Review of Regex Symbols
 - Case-Insensitive Matching
 - Substituting Strings with the sub() Method
 - Managing Complex Regexes

Python Basics

Math Operators

From **Highest** to **Lowest** precedence:

Operators	Operation	Example
**	Exponent	2 ** 3 = 8
%	Modulus/Remaider	22 % 8 = 16
//	Integer division	22 // 8 = 2
/	Division	22 / 8 = 2.75
*	Multiplication	3 * 3 = 15
-	Subtraction	5 - 2 = 3

+	Addition	$2 + 2 = 4$
---	----------	-------------

Examples of expressions in the interactive shell:

```
>>> 2 + 3 * 6
20

>>> (2 + 3) * 6
30

>>> 2 ** 8
256

>>> 23 // 7
3

>>> 23 % 7
2

>>> (5 - 1) * ((7 + 1) / (3 - 1))
16.0
```

Data Types

Data Type	Examples
Integers	-2, -1, 0, 1, 2, 3, 4, 5
Floating-point numbers	-1.25, -1.0, --0.5, 0.0, 0.5, 1.0, 1.25
Strings	'a', 'aa', 'aaa', 'Hello!', '11 cats'

String Concatenation and Replication

String concatenation:

```
>>> 'Alice' + 'Bob'
'AliceBob'
```

String Replication:

```
>>> 'Alice' * 5
'AliceAliceAliceAliceAlice'
```

Variables

You can name a variable anything as long as it obeys the following three rules:

1. It can be only one word.
2. It can use only letters, numbers, and the underscore (_) character.
3. It can't begin with a number.

Example:

```
>>> spam = 'Hello'
>>> spam
'Hello'
```

Comments

Inline comment:

```
# This is a comment
```

Multiline Comment:

```
"""
This is a Multiline Comment
You can also use:
''' multiline comment '''
"""
```

The print() Function

```
>>> print('Hello world!')
Hello world!
```

The input() Function

Example Code:

```
>>> print('What is your name?')    # ask for their name
>>> myName = input()
>>> print('It is good to meet you, ' + myName)
```

Output:

```
What is your name?  
Al  
It is good to meet you, Al
```

The len() Function

Evaluates to the integer value of the number of characters in a string:

```
>>> len('hello')  
5
```

The str(), int(), and float() Functions

Convert Between Data Types:

Integer to String or Float:

```
>>> str(29)  
'29'  
  
>>> print('I am ' + str(29) + ' years old.')  
I am 29 years old.  
  
>>> str(-3.14)  
'-3.14'
```

Float to Integer:

```
>>> int(7.7)  
7  
  
>>> int(7.7) + 1  
8
```

Flow Control

Comparison Operators

Operator	Meaning
==	Equal to
!=	Not equal to

<	Less than
>	Greater Than
<=	Less than or Equal to
>=	Greater than or Equal to

These operators evaluate to True or False depending on the values you give them:

Examples:

```
>>> 42 == 42
True

>>> 40 == 42
False

>>> 'hello' == 'hello'
True

>>> 'hello' == 'Hello'
False

>>> 'dog' != 'cat'
True

>>> True == True
True

>>> True != False
True

>>> 42 == 42.0
True

>>> 42 == '42'
False
```

Boolean Operators

There are three Boolean operators: and, or, and not.

The *and* Operator's *Truth* Table:

Expression	Evaluates to
True and True	True
True and False	False

False and True	False
----------------	-------

False and False	False
-----------------	-------

The *or* Operator's *Truth* Table:

Expression	Evaluates to
True or True	True
True or False	True
False or True	True
False or False	False

The *not* Operator's *Truth* Table:

Expression	Evaluates to
not True	False
not False	True

Mixing Boolean and Comparison Operators

```
>>> (4 < 5) and (5 < 6)
True

>>> (4 < 5) and (9 < 6)
False

>>> (1 == 2) or (2 == 2)
True
```

You can also use multiple Boolean operators in an expression, along with the comparison operators:

```
>>> 2 + 2 == 4 and not 2 + 2 == 5 and 2 * 2 == 2 + 2
True
```

if Statements

```
if name == 'Alice':
    print('Hi, Alice.')
```

else Statements

```

name = 'Bob'
if name == 'Alice':
    print('Hi, Alice.')
else:
    print('Hello, stranger.')

```

elif Statements

```

name = 'Bob'
age = 5
if name == 'Alice':
    print('Hi, Alice.')
elif age < 12:
    print('You are not Alice, kiddo.')

```

```

name = 'Bob'
age = 30
if name == 'Alice':
    print('Hi, Alice.')
elif age < 12:
    print('You are not Alice, kiddo.')
else:
    print('You are neither Alice nor a little kid.')

```

while Loop Statements

```

spam = 0
while spam < 5:
    print('Hello, world.')
    spam = spam + 1

```

break Statements

If the execution reaches a break statement, it immediately exits the while loop's clause.

```

while True:
    print('Please type your name.')
    name = input()
    if name == 'your name':
        break
print('Thank you!')

```


continue Statements

When the program execution reaches a continue statement, the program execution immediately jumps back to the start of the loop.

```
while True:
    print('Who are you?')
    name = input()
    if name != 'Joe':          #(1)
        continue              #(2)
    print('Hello, Joe. What is the password? (It is a fish.)')
    password = input()        #(3)
    if password == 'swordfish':
        break                  #(4)
    print('Access granted.')   #(5)
```

for Loops and the range() Function

```
print('My name is')
for i in range(5):
    print('Jimmy Five Times (' + str(i) + ')')
```

Output:

My name is Jimmy Five Times (0) Jimmy Five Times (1) Jimmy Five Times (2) Jimmy Five Times (3) Jimmy Five Times (4)

The `range()` function can also be called with three arguments. The first two arguments will be the start and stop values, and the third will be the step argument. The step is the amount that the variable is increased by after each iteration.

```
for i in range(0, 10, 2):
    print(i)
```

Output:

```
0
2
4
6
8
```

You can even use a negative number for the step argument to make the for loop count down instead of up.

```
for i in range(5, -1, -1):  
    print(i)
```

Output:

```
5  
4  
3  
2  
1  
0
```

Importing Modules

```
import random  
for i in range(5):  
    print(random.randint(1, 10))
```

```
import random, sys, os, math
```

```
from random import *
```

Ending a Program Early with sys.exit()

```
import sys  
  
while True:  
    print('Type exit to exit.')  
    response = input()  
    if response == 'exit':  
        sys.exit()  
    print('You typed ' + response + '.')
```

Functions

```
def hello(name):  
    print('Hello ' + name)  
  
hello('Alice')  
hello('Bob')
```

Output:

```
Hello Alice  
Hello Bob
```

Return Values and return Statements

When creating a function using the def statement, you can specify what the return value should be with a return statement. A return statement consists of the following:

- The return keyword.
- The value or expression that the function should return.

```
import random  
def getAnswer(answerNumber):  
    if answerNumber == 1:  
        return 'It is certain'  
    elif answerNumber == 2:  
        return 'It is decidedly so'  
    elif answerNumber == 3:  
        return 'Yes'  
    elif answerNumber == 4:  
        return 'Reply hazy try again'  
    elif answerNumber == 5:  
        return 'Ask again later'  
    elif answerNumber == 6:  
        return 'Concentrate and ask again'  
    elif answerNumber == 7:  
        return 'My reply is no'  
    elif answerNumber == 8:  
        return 'Outlook not so good'  
    elif answerNumber == 9:  
        return 'Very doubtful'  
  
r = random.randint(1, 9)  
fortune = getAnswer(r)  
print(fortune)
```

The None Value

```
>>> spam = print('Hello!')
Hello!
>>> None == spam
True
```

Keyword Arguments and print()

```
print('Hello', end='')
print('World')
```

Output:

```
HelloWorld
```

```
>>> print('cats', 'dogs', 'mice')
cats dogs mice
```

```
>>> print('cats', 'dogs', 'mice', sep=',')
cats,dogs,mice
```

Local and Global Scope

- Code in the global scope cannot use any local variables.
- However, a local scope can access global variables.
- Code in a function's local scope cannot use variables in any other local scope.
- You can use the same name for different variables if they are in different scopes. That is, there can be a local variable named spam and a global variable also named spam.

The global Statement

If you need to modify a global variable from within a function, use the global statement:

```
def spam():
    global eggs
```

```
eggs = 'spam'

eggs = 'global'
spam()
print(eggs)
```

Output:

```
spam
```

There are four rules to tell whether a variable is in a local scope or global scope:

1. If a variable is being used in the global scope (that is, outside of all functions), then it is always a global variable.
2. If there is a global statement for that variable in a function, it is a global variable.
3. Otherwise, if the variable is used in an assignment statement in the function, it is a local variable.
4. But if the variable is not used in an assignment statement, it is a global variable.

Exception Handling

```
def spam(divideBy):
    try:
        return 42 / divideBy
    except ZeroDivisionError:
        print('Error: Invalid argument.')

print(spam(2))
print(spam(12))
print(spam(0))
print(spam(1))
```

Output:

```
21.0
3.5
Error: Invalid argument.
None
42.0
```

Dictionaries and Structuring Data

Example Dictionary:

```
myCat = {'size': 'fat', 'color': 'gray', 'disposition': 'loud'}
```

The keys(), values(), and items() Methods

values():

```
>>> spam = {'color': 'red', 'age': 42}
>>> for v in spam.values():
    print(v)
```

Output:

```
red
42
```

keys():

```
>>> for k in spam.keys():
    print(k)
```

Output:

```
color
age
```

items():

```
>>> for i in spam.items():
    print(i)
```

Output:

```
('color', 'red')
('age', 42)
```

Using the `keys()`, `values()`, and `items()` methods, a for loop can iterate over the keys, values, or key-value pairs in a dictionary, respectively

```
>>> spam = {'color': 'red', 'age': 42}
>>> for k, v in spam.items():
    print('Key: ' + k + ' Value: ' + str(v))
```

Output:

```
Key: age Value: 42
Key: color Value: red
```

Checking Whether a Key or Value Exists in a Dictionary

```
>>> spam = {'name': 'Zophie', 'age': 7}

>>> 'name' in spam.keys()
True

>>> 'Zophie' in spam.values()
True

>>> 'color' in spam.keys()
False

>>> 'color' not in spam.keys()
True

>>> 'color' in spam
False
```

The `get()` Method

```
>>> picnicItems = {'apples': 5, 'cups': 2}
>>> 'I am bringing ' + str(picnicItems.get('cups', 0)) + ' cups.'
'I am bringing 2 cups.'
>>> 'I am bringing ' + str(picnicItems.get('eggs', 0)) + ' eggs.'
'I am bringing 0 eggs.'
```

The `setdefault()` Method

```
spam = {'name': 'Pooka', 'age': 5}
if 'color' not in spam:
    spam['color'] = 'black'
```

The above code is equal to:

```
>>> spam = {'name': 'Pooka', 'age': 5}
>>> spam.setdefault('color', 'black')
'black'
>>> spam
{'color': 'black', 'age': 5, 'name': 'Pooka'}
>>> spam.setdefault('color', 'white')
'black'
>>> spam
{'color': 'black', 'age': 5, 'name': 'Pooka'}
```

Pretty Printing

```
import pprint
message = 'It was a bright cold day in April, and the clocks were striking
thirteen.'
count = {}

for character in message:
    count.setdefault(character, 0)
    count[character] = count[character] + 1

pprint.pprint(count)
```

Output:

```
{ ' ': 13,
  ',': 1,
  '.': 1,
  'A': 1,
  'I': 1,
  'a': 4,
  'b': 1,
  'c': 3,
  'd': 3,
  'e': 5,
  'g': 2,
  'h': 3,
  'i': 6,
  'k': 2,
```



```
'l': 3,  
'n': 4,  
'o': 2,  
'p': 1,  
'r': 5,  
's': 3,  
't': 6,  
'w': 2,  
'y': 1}
```

Manipulating Strings

Escape Characters

Escape character	Prints as
'	Single quote
"	Double quote
\t	Tab
\n	Newline (line break)
\	Backslash

Example:

```
>>> print("Hello there!\nHow are you?\nI\'m doing fine.")
```

Output:

```
Hello there!  
How are you?  
I'm doing fine.
```

Raw Strings

A raw string completely ignores all escape characters and prints any backslash that appears in the string.

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

Output:

```
That is Carol\'s cat.
```

Multiline Strings with Triple Quotes

```
print('''Dear Alice,  
  
Eve's cat has been arrested for catnapping, cat burglary, and extortion.  
  
Sincerely,  
Bob''')
```

Output:

```
Dear Alice,  
  
Eve's cat has been arrested for catnapping, cat burglary, and extortion.  
  
Sincerely,  
Bob
```

Indexing and Slicing Strings

H	e	l	l	o		w	o	r	l	d	!
0	1	2	3	4	5	6	7	8	9	10	11

```
>>> spam = 'Hello world!'

>>> spam[0]
'H'

>>> spam[4]
'o'

>>> spam[-1]
'!'

>>> spam[0:5]
'Hello'

>>> spam[:5]
'Hello'
```

```
>>> spam[6:]  
'world!'
```

Slicing:

```
>>> spam = 'Hello world!'  
  
>>> fizz = spam[0:5]  
  
>>> fizz  
'Hello'
```

The in and not in Operators with Strings

```
>>> 'Hello' in 'Hello World'  
True  
  
>>> 'Hello' in 'Hello'  
True  
  
>>> 'HELLO' in 'Hello World'  
False  
  
>>> '' in 'spam'  
True  
  
>>> 'cats' not in 'cats and dogs'  
False
```

The upper(), lower(), isupper(), and islower() String Methods

upper() and lower():

```
>>> spam = 'Hello world!'  
  
>>> spam = spam.upper()  
  
>>> spam  
'HELLO WORLD!'  
  
>>> spam = spam.lower()  
  
>>> spam  
'hello world!'
```

isupper() and islower():

```
>>> spam = 'Hello world!'

>>> spam.islower()
False

>>> spam.isupper()
False

>>> 'HELLO'.isupper()
True

>>> 'abc12345'.islower()
True

>>> '12345'.islower()
False

>>> '12345'.isupper()
False
```

The isX String Methods

- **isalpha()** returns True if the string consists only of letters and is not blank.
- **isalnum()** returns True if the string consists only of letters and numbers and is not blank.
- **isdecimal()** returns True if the string consists only of numeric characters and is not blank.
- **isspace()** returns True if the string consists only of spaces, tabs, and new-lines and is not blank.
- **istitle()** returns True if the string consists only of words that begin with an uppercase letter followed by only lowercase letters.

The startswith() and endswith() String Methods

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

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

>>> 'abc123'.startswith('abcdef')
False

>>> 'abc123'.endswith('12')
False

>>> 'Hello world!'.startswith('Hello world!')
```

```
True
```

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

The join() and split() String Methods

join():

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

>>> ' '.join(['My', 'name', 'is', 'Simon'])
'My name is Simon'

>>> 'ABC'.join(['My', 'name', 'is', 'Simon'])
'MyABCnameABCisABCSimon'
```

split():

```
>>> 'My name is Simon'.split()
['My', 'name', 'is', 'Simon']

>>> 'MyABCnameABCisABCSimon'.split('ABC')
['My', 'name', 'is', 'Simon']

>>> 'My name is Simon'.split('m')
['My na', 'e is Si', 'on']
```

Justifying Text with rjust(), ljust(), and center()

rjust() and ljust():

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

>>> 'Hello'.rjust(20)
'           Hello'

>>> 'Hello World'.rjust(20)
'         Hello World'

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

An optional second argument to `rjust()` and `ljust()` will specify a fill character other than a space character. Enter the following into the interactive shell:

```
>>> 'Hello'.rjust(20, '*')
'*****Hello'

>>> 'Hello'.ljust(20, '-')
'Hello-----'
```

`center()`:

```
>>> 'Hello'.center(20)
'      Hello      '

>>> 'Hello'.center(20, '=')
'====Hello===='
```

Removing Whitespace with `strip()`, `rstrip()`, and `lstrip()`

```
>>> spam = '    Hello World    '

>>> spam.strip()
'Hello World'

>>> spam.lstrip()
'Hello World '

>>> spam.rstrip()
'    Hello World'
```

```
>>> spam = 'SpamSpamBaconSpamEggsSpamSpam'
>>> spam.strip('ampS')

'BaconSpamEggs'
```

Copying and Pasting Strings with the `pyperclip` Module

```
>>> import pyperclip

>>> pyperclip.copy('Hello world!')
```

```
>>> pyperclip.paste()  
'Hello world!'
```

Regular Expressions

1. Import the regex module with `import re`.
2. Create a Regex object with the `re.compile()` function. (Remember to use a raw string.)
3. Pass the string you want to search into the Regex object's `search()` method. This returns a Match object.
4. Call the Match object's `group()` method to return a string of the actual matched text.

All the regex functions in Python are in the `re` module:

```
>>> import re
```

Matching Regex Objects

```
>>> phoneNumRegex = re.compile(r'\d\d\d-\d\d\d-\d\d\d\d')  
  
>>> mo = phoneNumRegex.search('My number is 415-555-4242.')  
  
>>> print('Phone number found: ' + mo.group())  
Phone number found: 415-555-4242
```

Grouping with Parentheses

```
>>> phoneNumRegex = re.compile(r'(\d\d\d)-(\d\d\d-\d\d\d\d)')  
  
>>> mo = phoneNumRegex.search('My number is 415-555-4242.')  
  
>>> mo.group(1)  
'415'  
  
>>> mo.group(2)  
'555-4242'  
  
>>> mo.group(0)  
'415-555-4242'  
  
>>> mo.group()  
'415-555-4242'
```

To retrieve all the groups at once: `groups()` method—note the plural form for the name.

```
>>> mo.groups()
('415', '555-4242')

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

>>> print(areaCode)
415

>>> print(mainNumber)
555-4242
```

Matching Multiple Groups with the Pipe

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'.

```
>>> 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:

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

>>> mo.group()
'Batmobile'

>>> mo.group(1)
'mobile'
```

Optional Matching with the Question Mark

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'
```

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.

```
>>> 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

While `*` means “match zero or more,” the `+` (or plus) means “match one or more”. 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
```

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'.

```
>>> haRegex = re.compile(r'(Ha){3}')
>>> mo1 = haRegex.search('HaHaHa')
>>> mo1.group()
'HaHaHa'

>>> mo2 = haRegex.search('Ha')
>>> mo2 == None
True
```

Greedy and Nongreedy Matching

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 possible, 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'
```

The findall() Method

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') # has no groups
>>> phoneNumRegex.findall('Cell: 415-555-9999 Work: 212-555-0000')
['415-555-9999', '212-555-0000']
```

To summarize what the `findall()` method returns, remember the following:

- When called on a regex with no groups, such as `\d-\d\d\d-\d\d\d\d`, the method `findall()` returns a list of

ng matches, such as ['415-555-9999', '212-555-0000'].

- When called on a regex that has groups, such as `(\d\d\d)-\d\d-(\d\d\d\d)`, the method `findall()` returns a list of tuples of strings (one string for each group), such as `[('415', '9999'), ('212', '555', '0000')]`.

Making Your Own Character Classes

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. For example, the character class `[aeiouAEIOU]` will match any vowel, both lowercase and uppercase.

```
>>> 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.

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. For example, enter the following into the interactive shell:

```
>>> 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

- You can also use the caret symbol (^) at the start of a regex to indicate that a match must occur at the beginning of the searched text.
- Likewise, you can put a dollar sign (\$) at the end of the regex to indicate the string must end with this regex pattern.
- And you can use the ^ and \$ together to indicate that the entire string must match the regex—that is, it's not enough for a match to be made on some subset of the string.

The `r'^Hello'` regular expression string matches strings that begin with 'Hello':

```
>>> beginsWithHello = re.compile(r'^Hello')
>>> beginsWithHello.search('Hello world!')
<_sre.SRE_Match object; span=(0, 5), match='Hello'>
```

```
>>> beginsWithHello.search('He said hello.') == None
True
```

The `r'\d$'` regular expression string matches strings that end with a numeric character from 0 to 9:

```
>>> wholeStringIsNum = re.compile(r'^\d+$')

>>> wholeStringIsNum.search('1234567890')
<_sre.SRE_Match object; span=(0, 10), match='1234567890'>

>>> wholeStringIsNum.search('12345xyz67890') == None
True

>>> wholeStringIsNum.search('12 34567890') == None
True
```

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']
```

Matching Everything with Dot-Star

```
>>> nameRegex = re.compile(r'First Name: (.*) Last Name: (.*)')

>>> mo = nameRegex.search('First Name: Al Last Name: Sweigart')

>>> mo.group(1)
'Al'

>>> mo.group(2)
'Sweigart'
```

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 `(.*?)`. The question mark tells Python to match in a nongreedy way:

```

>>> 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

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:

```

>>> noNewlineRegex = re.compile('.*')
>>> noNewlineRegex.search('Serve the public trust.\nProtect the innocent.
\nUphold the law.').group()
'Serve the public trust.'

>>> newlineRegex = re.compile('.*', re.DOTALL)
>>> newlineRegex.search('Serve the public trust.\nProtect the innocent.
\nUphold the law.').group()
'Serve the public trust.\nProtect the innocent.\nUphold the law.'

```

Review of Regex Symbols

Symbol	Matches
?	zero or one of the preceding group.
*	zero or more of the preceding group.
+	one or more of the preceding group.
{n}	exactly n of the preceding group.
{n,}	n or more of the preceding group.
{,m}	0 to m of the preceding group.
{n,m}	at least n and at most m of the preceding p.
{n,m}? or *? or +?	performs a nongreedy match of the preceding p.
^spam	means the string must begin with spam.
spam\$	means the string must end with spam.

.	any character, except newline characters.
\d, \w, and \s	a digit, word, or space character, ectively.
\D, \W, and \S	anything except a digit, word, or space acter, respectively.
[abc]	any character between the brackets (such as a, b,).
[^abc]	any character that isn't between the brackets.

Case-Insensitive Matching

To make your regex case-insensitive, you can pass `re.IGNORECASE` or `re.I` as a second argument to `re.compile()`:

```
>>> robocop = re.compile(r'robocop', re.I)
>>> robocop.search('Robocop is part man, part machine, all cop.').group()
'Robocop'

>>> robocop.search('ROBOCOP protects the innocent.').group()
'ROBOCOP'

>>> robocop.search('Al, why does your programming book talk about robocop so
much?').group()
'robocop'
```

Substituting Strings with the sub() Method

The `sub()` method for Regex objects is passed two arguments:

1. The first argument is a string to replace any matches.
2. The second is the string for the regular expression.

The `sub()` method returns a string with the substitutions applied:

```
>>> 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.'

Another example:

```
>>> 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

To tell the `re.compile()` function to ignore whitespace and comments inside the regular expression string, “verbose mode” can be enabled by passing the variable `re.VERBOSE` as the second argument to `re.compile()`.

Now instead of a hard-to-read regular expression like this:

```
phoneRegex = re.compile(r'((\d{3}|\(\d{3}\))?(s|-|\.)?\d{3}(s|-|\.)\d{4})(s*(ext|x|ext.)s*\d{2,5})?')
```

you can spread the regular expression over multiple lines with comments like this:

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