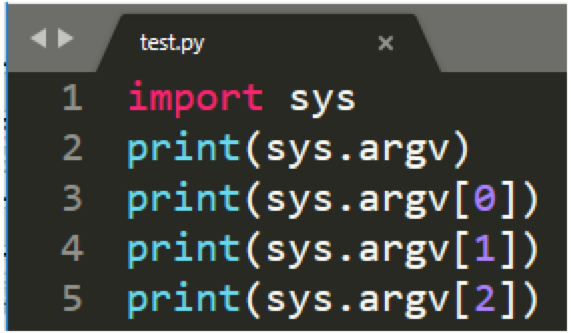
Working with files (write)

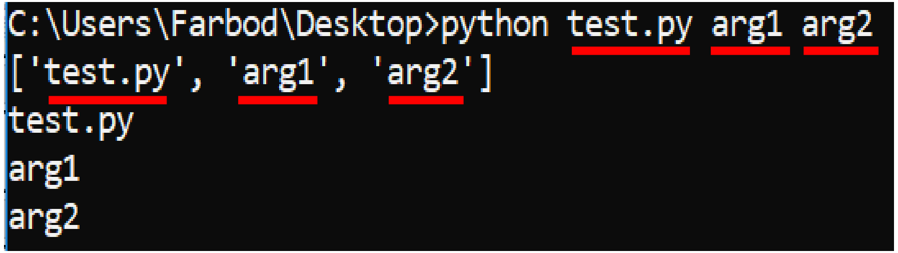
- Creating a new file:
  - `new_file = open(‘new.txt’, mode=‘w’)`
- Writing some text in the file:
  - `new_file.write(‘this is a text file’)`
- Text will be shown when you close the file:
  - `new_file.close()`
- NOTE: if you open the file again, all of the previous text will be deleted.

sys module

- The sys module provides information about constants, functions and methods of the Python interpreter.
- One of the most important features: `sys.argv`
- `sys.argv` is a list, which contains the command-line **arguments** passed to the script. The **first item** of this list contains the name of the **script itself**. The arguments follow the script name.



```python
import sys
print(sys.argv)
print(sys.argv[0])
print(sys.argv[1])
print(sys.argv[2])
```



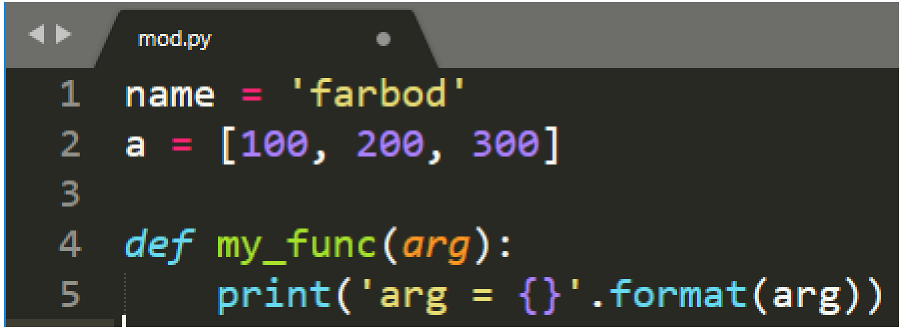
```
C:\Users\Farbod\Desktop>python test.py arg1 arg2
['test.py', 'arg1', 'arg2']
test.py
arg1
arg2
```

Modules and Packages

- Module: just a .py script that you call in another .py script.
- Package: a collection of modules
- Why we use them?
  - **Modular programming** refers to the process of breaking a large programming task into separate, smaller, more manageable subtasks or **modules**. Individual modules can then be assembled together like building blocks to create a larger application.
- Ways to define a module:
  - A module can be written in Python itself.
  - A module can be written in C and loaded dynamically at run-time, like the re (regular expression) module.
  - A built-in module is intrinsically contained in the interpreter, like the itertools module

Modules

- Module: just a .py script that you call in another .py script.
  - □ An example: *mod.py*



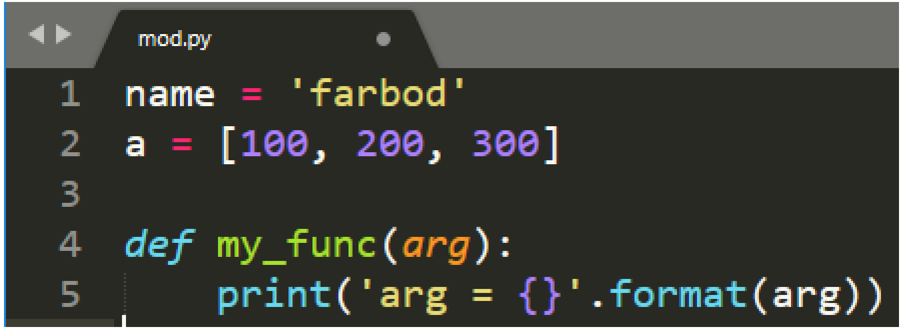
```python
name = 'farbod'
a = [100, 200, 300]

def my_func(arg):
    print('arg = {}'.format(arg))
```

  - □ All of the definitions of this file can be used in another file with the `import` keyword:
    - □ `import mod`

Modules

- From the caller file, objects in the module are only accessible when prefixed with <module_name> via **dot notation (.)**, as illustrated below:
  - >> import mod
  - >> name ☞ returns error
  - >> mod.name = ‘farbod’
  - >> a ☞ returns error
  - >> mod.a ☞ returns [100, 200, 300]
  - >> mod.my_func(‘farbod’) ☞ prints ‘arg = farbod’

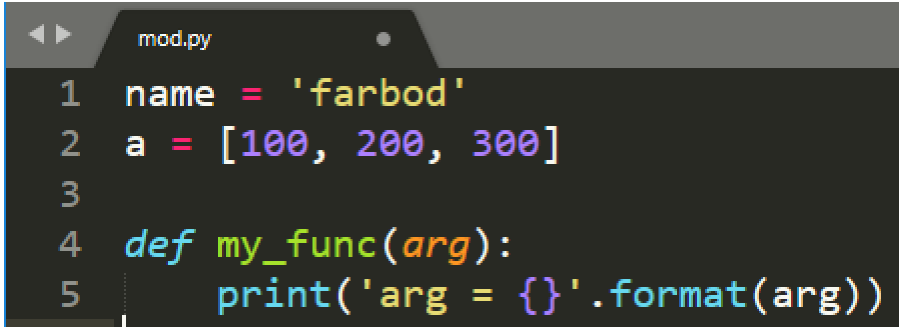


```python
name = 'farbod'
a = [100, 200, 300]

def my_func(arg):
    print('arg = {}'.format(arg))
```

Modules

- Another form of import is to import only a specific number of objects with keyword `from:` (Note: dot notation is not required here!!!)
  - `>> from mod import name, my_func`
  - `>> name` ☞ returns `‘farbod’`
  - `>> my_func(‘farbod’)` ☞ prints ‘`arg = farbod’`
  - `>> a` ☞ returns error



```
name = 'farbod'
a = [100, 200, 300]

def my_func(arg):
    print('arg = {}'.format(arg))
```

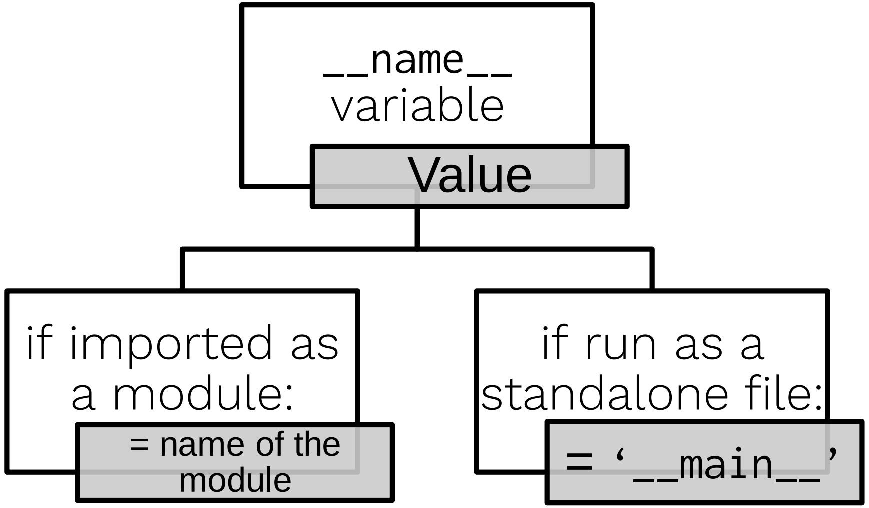
Modules

- Alternate ways to import a module

Definition	Code
Import everything from a module	`from mod import *`
Import objects but enter them with alternate names	`from mod import name as n, my_func as f`
Import an entire module under an alternate name	`import mod as m`

__name__ and ‘__main__’

- __name__ is a variable.
- Its value is determined based on how you use a .py file.
- when we run a file, the statement `__name__=='__main__‘` is `True` and the statements under if condition is executed.
- on the other hand, if we import a .py file, __name__ is equal to the module name and `__name__==‘__main__’` is false.

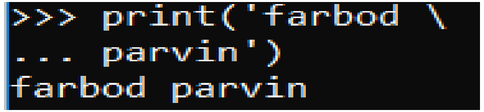
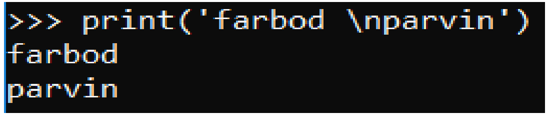
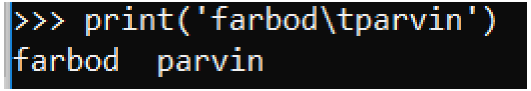


Locating modules

- When you import a module, the Python interpreter searches for the module in the following sequences:
  1. The current directory.
  2. If the module isn't found, Python then searches each directory in the shell variable PYTHONPATH.
  3. If all else fails, Python checks the default path.
- The module search path (all of the three locations) is stored in the system module sys as the `sys.path` variable.
  - □ `>> import sys`
  - □ `>> print(sys.path)` ☞ returns a list of directories.

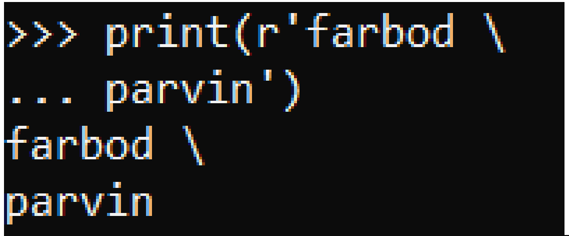
Escape sequence and raw string

- An **escape sequence** is a sequence of characters that does not represent itself when used inside a string literal, but is translated into another character or a sequence of characters that may be difficult or impossible to represent directly.
- Two important escape sequences in python are:

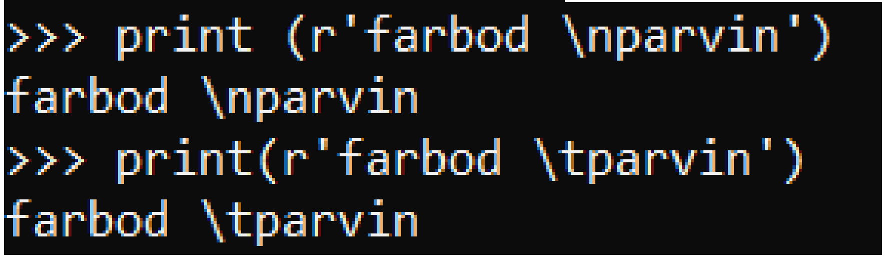
Escape sequence	Description	Example
\newline	Backslash and newline ignored	 >>> print('farbod \ ... parvin') farbod parvin
\n	New line	 >>> print('farbod \nparvin') farbod parvin
\t	TAB	 >>> print('farbod\tparvin') farbod parvin

Escape sequence and raw string

- However, if you put a ‘r’ before the string, it will be treated as a **raw string** and the escape characters are printed exactly as they appear.
- Example:



```
>>> print(r'farbod \
... parvin')
farbod \
parvin
```



```
>>> print (r'farbod \nparvin')
farbod \nparvin
>>> print(r'farbod \tparvin')
farbod \tparvin
```

Regular Expression (RegEx)
Source: Google's python class

- Regular Expression is a sequence of characters that forms a search pattern.
- RegEx can be used to check if a string contains the specified search pattern.
- Importing the module:
  - `>> import re`
- searching for a specific pattern:
  - `match = re.search(pat, text)`
  - `pat` is the pattern that we are searching for (string)
  - `text` is the source text that we want to find the pattern in it. (string)
  - `match` is a match object if the pattern is found or it is None if the pattern is not found. (not string, not Boolean, just a match object!!!)

Regular Expression (RegEx)

Pattern	Meaning
`a, B, 1, …`	ordinary characters just match themselves exactly. The meta-characters which do not match themselves because they have special meanings are: `. ^ $ * + ? { [ ] \ | ( )`
`.` (a period)	matches any single character except newline '\n'
`\w`	(lowercase w) matches a "word" character: a letter or digit or underbar [a-zA-Z0-9_].
`\s`	(lowercase s) matches a single whitespace character
`\d`	digit [0-9]
`\S`	(upper case S) matches any non-whitespace character
`\`	Inhibit the "specialness" of a character. So, for example, use \. to match a period or \\ to match a slash.

Regular Expression (RegEx)

- Rules:
  - □ The search proceeds through the string from start to end, stopping at the first match found
  - □ All of the pattern must be matched, not just a part of it
  - □ If `match = re.search(pat, str)` is successful, match is not `None` and in particular

    `match.group()` is the matching text
- Basic examples:
  - □ `## Search for pattern 'iii' in string 'piiig'.`
  - □ `## All of the pattern must match, but it may appear anywhere.`
  - □ `## On success, match.group() is matched text.`
  - □ `>> match = re.search(r'iii', 'piiig') # found, match.group() == "iii"`
  - □ `>> match = re.search(r'igs', 'piiig') # not found, match == None`

Regular Expression (RegEx)

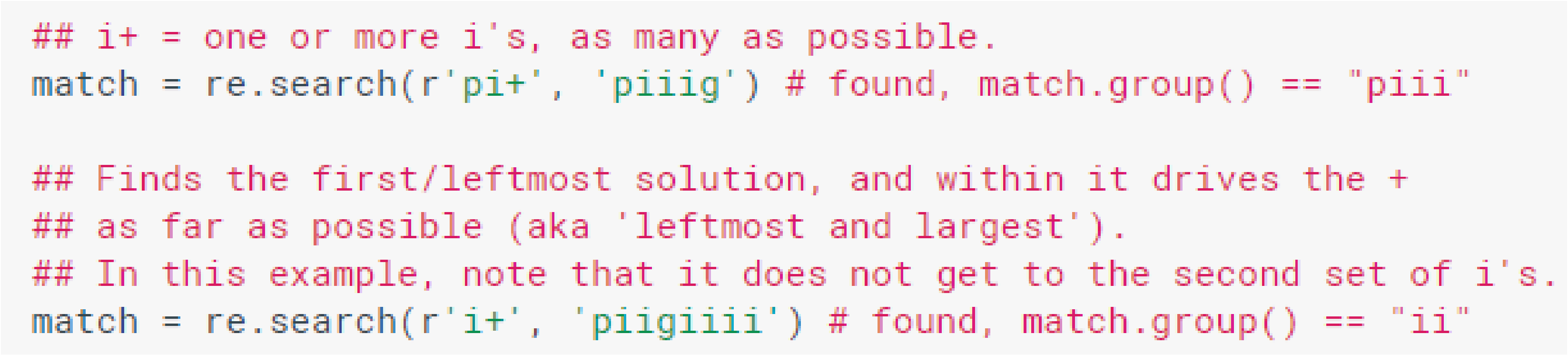
- Basic Examples:
  - ## . = any char but \n
  - `>> match=re.search(r'..g', 'piiig')` #found, match.group() == "iig“
  - ## \d = digit char, \w = word char
  - `>> match=re.search(r'\d\d\d', 'p123g')` #found, match.group()=="123”
  - `>> match=re.search(r'\w\w\w','@@abcd!!')` #found, match.group()=="abc”

Regular Expression (RegEx)

- Repitition:

Symbol	Meaning
+	1 or more occurrences of the pattern to its left, e.g. 'i+' = one or more i's
*	0 or more occurrences of the pattern to its left

- Examples:

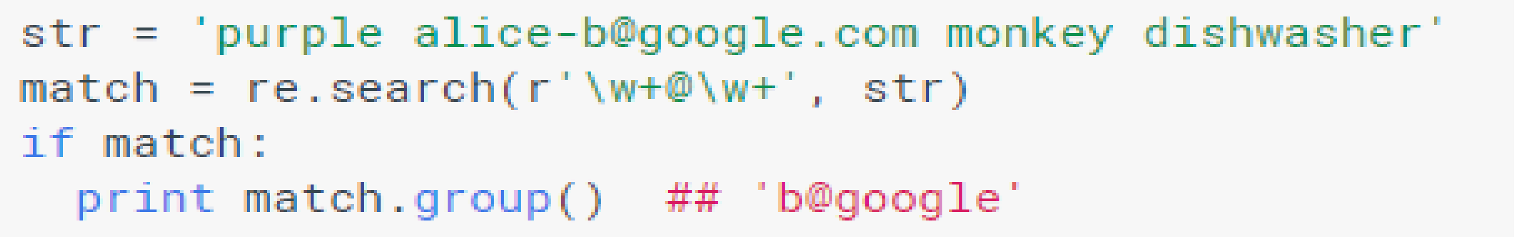


```
## i+ = one or more i's, as many as possible.
match = re.search(r'pi+', 'piiig') # found, match.group() == "piii"

## Finds the first/leftmost solution, and within it drives the +
## as far as possible (aka 'leftmost and largest').
## In this example, note that it does not get to the second set of i's.
match = re.search(r'i+', 'piigiiii') # found, match.group() == "ii"
```

Regular Expression (RegEx)

- Email example:
  - □ We can use the following pattern but the it doesn't get the whole address since \w does not match ‘-’ or ‘.’

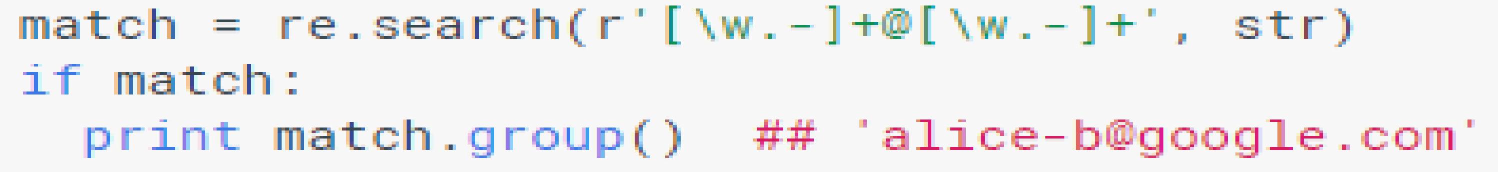


```
str = 'purple alice-b@google.com monkey dishwasher'
match = re.search(r'\w+@\w+', str)
if match:
  print match.group()  ## 'b@google'
```

  - □ Instead we can use brackets [] to indicate a set of chars, so [abc] matches 'a' or 'b' or 'c'. The codes \w, \s etc. work inside square brackets too with the one exception that dot (.) just means a literal dot.

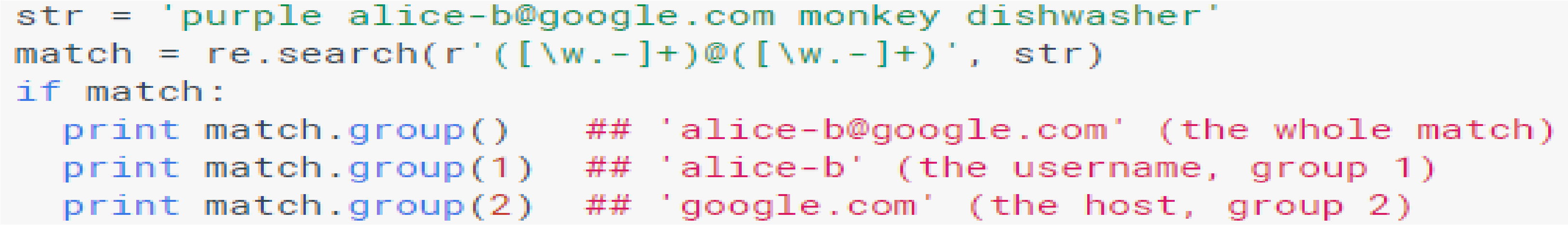
Regular Expression (RegEx)

- For the emails problem, the square brackets are an easy way to add '.' and '-' to the set of chars which can appear around the @ with the pattern r'[\w.-]+@[\w.-]+' to get the whole email address:



```
match = re.search(r'[\w.-]+@[\w.-]+', str)
if match:
  print match.group()  ## 'alice-b@google.com'
```

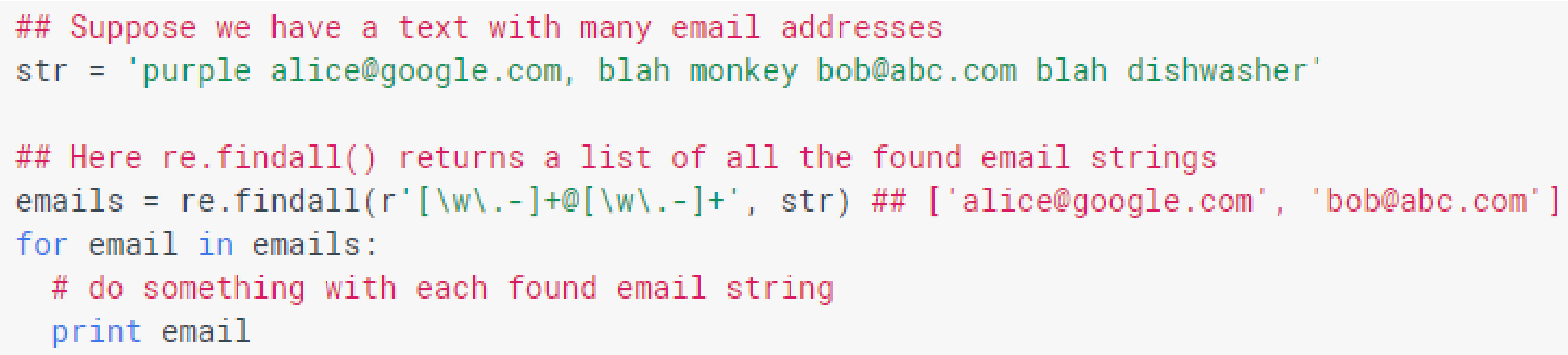
- Group extraction: The "group" feature of a regular expression allows you to pick out parts of the matching text



```
str = 'purple alice-b@google.com monkey dishwasher'
match = re.search(r'([\w.-]+)@([\w.-]+)', str)
if match:
  print match.group()   ## 'alice-b@google.com' (the whole match)
  print match.group(1)  ## 'alice-b' (the username, group 1)
  print match.group(2)  ## 'google.com' (the host, group 2)
```

Regular Expression (RegEx)

- `findall()` is probably the single most powerful function in the re module. Above we used re.search() to find the first match for a pattern. findall() finds *all* the matches and returns them as a list of strings, with each string representing one match.

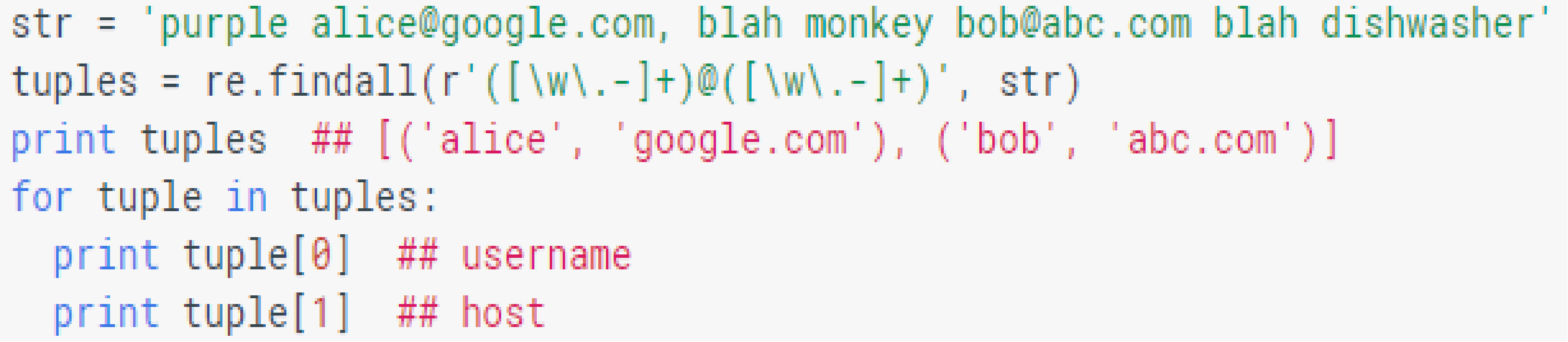


```
## Suppose we have a text with many email addresses
str = 'purple alice@google.com, blah monkey bob@abc.com blah dishwasher'

## Here re.findall() returns a list of all the found email strings
emails = re.findall(r'[\w\.-]+@[\w\.-]+', str) ## ['alice@google.com', 'bob@abc.com']
for email in emails:
  # do something with each found email string
  print email
```

Regular Expression (RegEx)

- The parenthesis ( ) group mechanism can be combined with findall(). If the pattern includes 2 or more parenthesis groups, then instead of returning a list of strings, findall() returns a list of *tuples*. Each tuple represents one match of the pattern, and inside the tuple is the group(1), group(2) .. data.



```
str = 'purple alice@google.com, blah monkey bob@abc.com blah dishwasher'
tuples = re.findall(r'([\w\.-]+)@([\w\.-]+)', str)
print tuples  ## [('alice', 'google.com'), ('bob', 'abc.com')]
for tuple in tuples:
  print tuple[0]  ## username
  print tuple[1]  ## host
```