Python Tutorial

What are We Talking About Today?

- Introduction to Python
  - High-level, interpreted, object-oriented
  - Supports multiple programming paradigms
- Data Types & Variables
  - Dynamically typed: integers, strings, lists, etc.
  - Variables can change types based on assigned values
- Control Structures & Functions
  - if-else statements, loops (for, while)
  - Functions defined with def, support parameters and return values

What is Python?

- Interpreted, Object-oriented, High-level programming language
- Has high-level built in data structures
- Has dynamic typing and dynamic binding
- Supports both procedural and object-oriented paradigm
- Downloads, documentation, community support, news and event at:

    https://www.python.org/

What is Python (cont’d)

- Useful for scripting
- Does not have a compilation step
- Various built-in functions/modules allow for fast development
- Compatible with many popular databases like PostgreSQL and MySQL

What does Python code look like?

- Simpler than equivalent C, C++, or Java code
- Shorter than C, C++, Java
- Offers greater error checking than C
- Simple, easy to learn syntax
- Allows splitting program into modules which can be reused

Getting Started with Python Interpreter

- The interpreter usually gets installed at /usr/local/bin/pythonX.Y:

  /usr/local/bin/python3.8

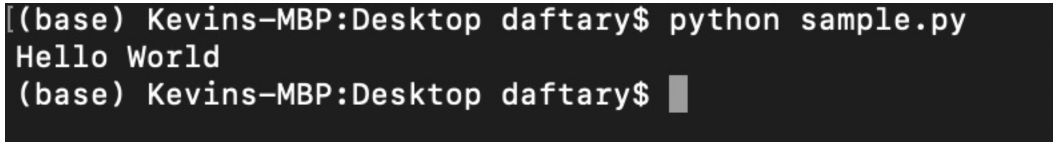
- The interpreter can be started using `python3.8` or later versions (up to 3.13, depending on the cloud support) or simply `python` after putting the interpreter path in Unix shell's search path or the environment variables in Windows.
- Another way to start the interpreter is *`python –c command [arg]`* … This executes the statements in 'command' script
- The script name and arguments are turned into a list of strings and assigned to the **argv** variable in the **sys** module

Hello World Example

- sample.py:

```
’’’sample.py file to print Hello World’’’
print(“Hello World”)
```

- Run `sample.py` as *python sample.py*



```
[(base) Kevins-MBP:Desktop daftary$ python sample.py
Hello World
(base) Kevins-MBP:Desktop daftary$
```

Comments in Python

- There are two types of comments in Python

```
# Single-line Shell-style comments

’’’ These are
    Multi-line comments.’’’
```

Python Variables

- Python variables can be declared by any name or even alphabets like a, aa, abc, etc.
- **Variables** are case-sensitive (abc != aBc)
- **Global** variables can be used anywhere (declared outside a function)
- **Local** variables restricted to a function or class
- **No** keyword called **static** is present
- Variables assigned values inside a class declaration are **class variables**
- Variables assigned values in class methods are **instance variables**

Python Variables (cont’d)

- Variables are containers for storing data values
- Python has no command for declaring a variable
- A variable is created the moment you first assign a value to it
- Variables are **not statically typed**
- Integers can become floats, then can become strings
- Variables take the type of the current value
- If you want to specify the data type of a variable, this can be done with “casting”
- Variable types include :
  - Boolean, Integer
  - Float, String
  - List, Object
  - NULL, Tuple
  - Dictionary, Set

Python Variables (cont'd)

- Assignment by value

```
a = 10
b = "foo"
c = [1, 2, 3, 4]     # List
d = (1, 2)           # Tuple
e = {'key': 'value'} # Dictionary
```

Displaying Variables

- To display a variable, use the **`print`** statement; pass the variable name to the `print` statement, enclosing it in brackets (for python 3.x) or without brackets(for python 2.x):

  ```
  age = 18
  print(age) # python 3.x
  print age  # python 2.x
  ```

- To display both text strings and variables, pass them to the **`print`** statement as individual arguments, separated by commas:

  ```
  print("The legal voting age is ", age)
  ```

Naming Variables

- The following rules and conventions must be followed when naming a variable:
  - Variable names must **begin** with a **letter** or **underscore** (_) character
  - Variable names may contain **alpha**numeric characters (uppercase and lowercase letters), **numbers**, or **underscores** (_).
  - Variable names cannot contain spaces
  - Variable names are **case sensitive**

Python Constants

- Constants are special variables that hold values that **<u>should</u> not be changed**
- Start with letter or underscore (_) followed by letters, numbers or underscores
- Use them for named items that will not change
- Constant names use all uppercase letters
- Constants have global scope
- The constants module of python can be used for some common constants like PI, GRAVITY etc.
- <u>Constants are not really part of Python specification but part of community usage</u>

Python Operators

- Standard **Arithmetic** operators

  `+, -, *, / (always returns a float value), % (modulus), ** (exponentiation) and // (floor division)`

- String **concatenation** with a ‘+’ character

  ```
  car = "SEAT" + " Altea"
  print(car) would output "SEAT Altea"
  ```

- Basic Boolean comparison with “==“
- Using only = will overwrite a variable value (assignment)
- Less than < and greater than >
- <= and >= as above but include equality
- != can be used to check if two variables are not equal

Python Operators (cont’d)

- **Assignment** (=) and combined assignment

```
a = 3
a += 5 # sets a to 8;
b = "Hello "
b += "There!" # sets b to "Hello There!"
```

- **Bitwise** operators (&, |, ^, ~, <<, >>)

```
 a ^ b(Xor: Bits that are set in a or b but not both are set.)
 ~a (Not: Bits that are set in a are not set, and vice versa.)
```

All arithmetic and bitwise operators can be combined with the assignment operator

Note: Python **DOES NOT** support ‘++’ and ‘--’ notation for auto increments and decrement

Python Operators (cont'd)

- **Logical** Operators
  - **and**: returns true if both statements are true (replacement for &&)
    `If x=3, then x<5 and x<10 returns True`
  - **or**: Returns **True** if one of the statements is true (replacement for ||)
    `If x=4, then x<4 or x<5 returns True`
  - **not**: Reverse the result, returns False if the result is true
- **Identity** Operators
  - **is**: Returns true if both variables are the same object
  - **is not**: Returns true if both variables are not the same object
- **Membership** Operators
  - **in**: Returns True if a sequence with the specified value is present in the object
  - **not in**: Returns True if a sequence with the specified value is not present in the object

Data Types

- Python is a **dynamically typed** language (similar to JavaScript)
- Python supports the following types:
  - **Boolean**: True or False
  - **Numeric** types
    - **Integer**: Positive or negative whole numbers, complex numbers (eg., 3 + 5j)
    - **Float**: Any real number
  - **Sequence** types
    - **String**: Sequence-type data type allowing for individual character access
    - **List**: *Ordered* collection of one or more data items, could be of different types, enclosed in square brackets (eg., [1, ‘Hello’, 3.41, True])
    - **Tuple**: *Ordered* collection of one or more data items, could be of different types, enclosed in parentheses (e.g., (1,2,”Hello”))
  - **Set**: *Unordered*, mutable collection that does not allow duplicate elements.
  - **Dictionary**: *Unordered* collection of data in *key:value* pairs form, enclosed in curly brackets (e.g., `{1:"Professor", 2:"Marco",3:"Papa"}`

Data Types Example

```
vat_rate = 0.175 # VAT Rate is numeric
print(vat_rate * 100 + "%") # throws TypeError
print(str(vat_rate * 100) + "%") # outputs "17.5%"
```

Numeric Data Types

- Python supports two numeric data types:
  - An **integer** is a positive or negative whole number with no decimal places `(-250, 2, 100, 10,000)` or complex numbers with ‘j’ denoting the imaginary part `(2 + 4j)`
  - A **floating-point number** is a number that contains decimal places or that is written in scientific notation `(-6.16, 3.17, 2.7541)`

Boolean Values

- A **Boolean value** is a value of **True** or **False** (true and false, in lower case ,are invalid)
- In Python programming, you can only use True or False Boolean values
- In other programming languages, you can use integers such as 1 = True, 0 = False

Strings in Python

- A **collection** of **one or more characters**, enclosed in single or double **quotes**
- Can use backslash as escape character
- Concatenate strings using ‘+’. Repeat using ‘*’
- Strings can be indexed

```
a = "Hello "
print(a[0]) # prints H
print(a[-1]) # prints o – reverse indexing
```

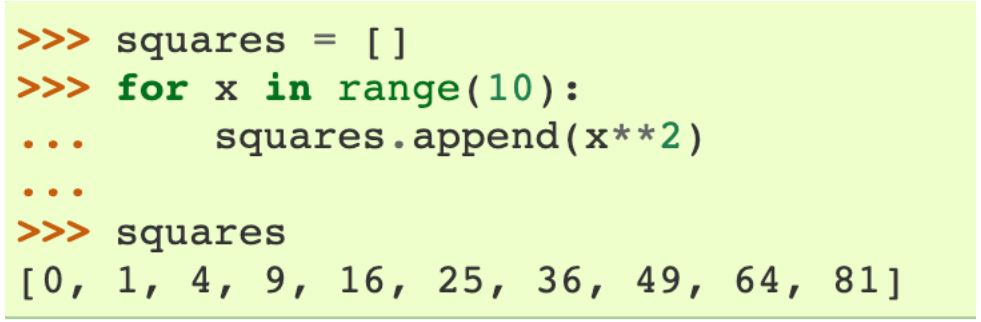
- Strings can be sliced

```
print(a[1:3]) # prints ‘el’
```

- Strings are **immutable**

Lists in Python

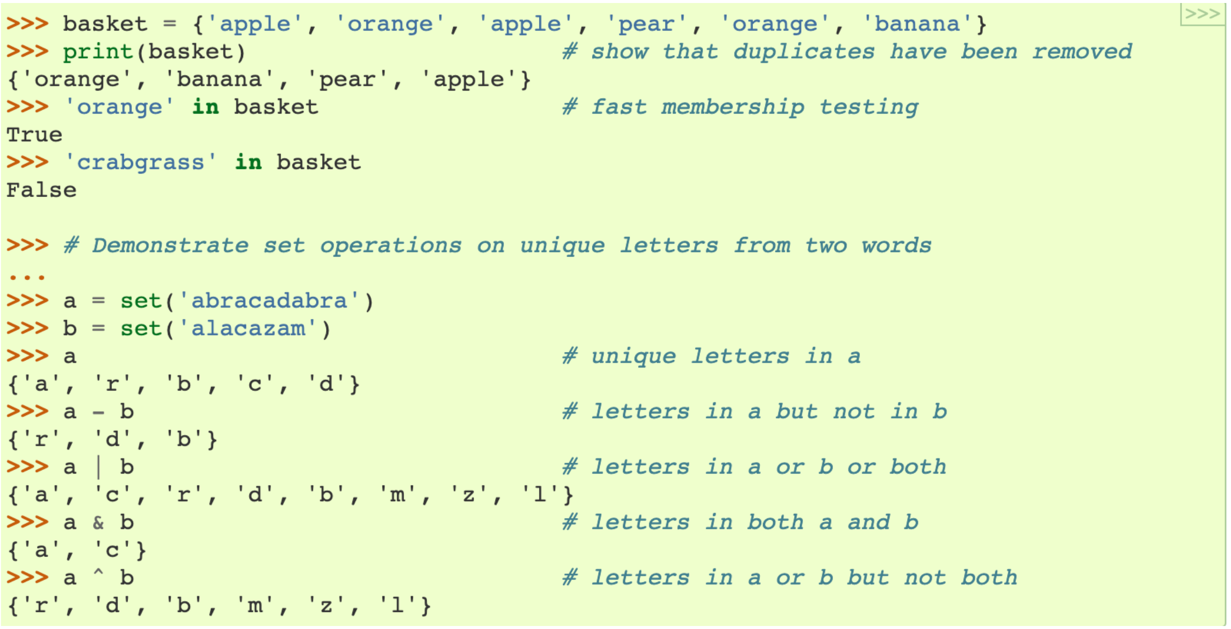
- An **ordered collection** of one or more data items, **not** necessarily of **same type**, enclosed by **square** [] brackets
  - Insertion order is preserved
  - Maintain order in iteration
- Lists have multiple methods like append(), insert(), remove(), sort(), count(), reverse(), etc. to manipulate the elements of the list
- The **del** statement allows deletion of elements and even complete lists (converts to empty list) as well as variables (reference error if you try to access the same variable)



```
>>> squares = []
>>> for x in range(10):
...     squares.append(x**2)
...
>>> squares
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Sets in Python

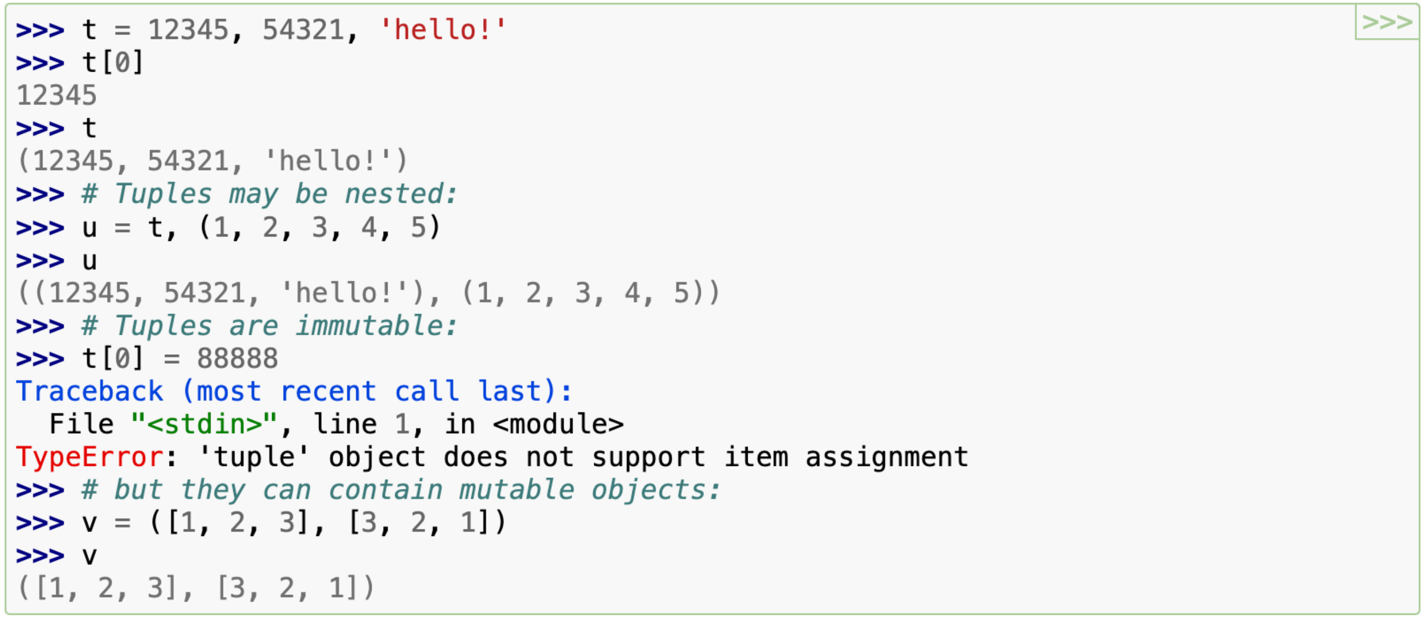
- An **unordered collection** of objects with **no duplicate** elements, **not** necessarily of **same type**, separated by commas, and enclosed by **curly** {} brackets
- Used for membership tests, eliminating duplicates, union, intersection, difference and symmetric difference



```
>>> basket = {'apple', 'orange', 'apple', 'pear', 'orange', 'banana'}
>>> print(basket)                      # show that duplicates have been removed
{'orange', 'banana', 'pear', 'apple'}
>>> 'orange' in basket                 # fast membership testing
True
>>> 'crabgrass' in basket
False

>>> # Demonstrate set operations on unique letters from two words
...
>>> a = set('abracadabra')
>>> b = set('alacazam')
>>> a                                  # unique letters in a
{'a', 'r', 'b', 'c', 'd'}
>>> a - b                              # letters in a but not in b
{'r', 'd', 'b'}
>>> a | b                              # letters in a or b or both
{'a', 'c', 'r', 'd', 'b', 'm', 'z', 'l'}
>>> a & b                              # letters in both a and b
{'a', 'c'}
>>> a ^ b                              # letters in a or b but not both
{'r', 'd', 'b', 'm', 'z', 'l'}
```

Tuples in Python

- An **ordered**, immutable collection of elements. It is like a list but cannot be modified after creation.
- Tuples are commonly used for grouping related data and ensuring that values remain unchanged.



```
>>> t = 12345, 54321, 'hello!'
>>> t[0]
12345
>>> t
(12345, 54321, 'hello!')
>>> # Tuples may be nested:
>>> u = t, (1, 2, 3, 4, 5)
>>> u
((12345, 54321, 'hello!'), (1, 2, 3, 4, 5))
>>> # Tuples are immutable:
>>> t[0] = 88888
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
>>> # but they can contain mutable objects:
>>> v = ([1, 2, 3], [3, 2, 1])
>>> v
([1, 2, 3], [3, 2, 1])
```

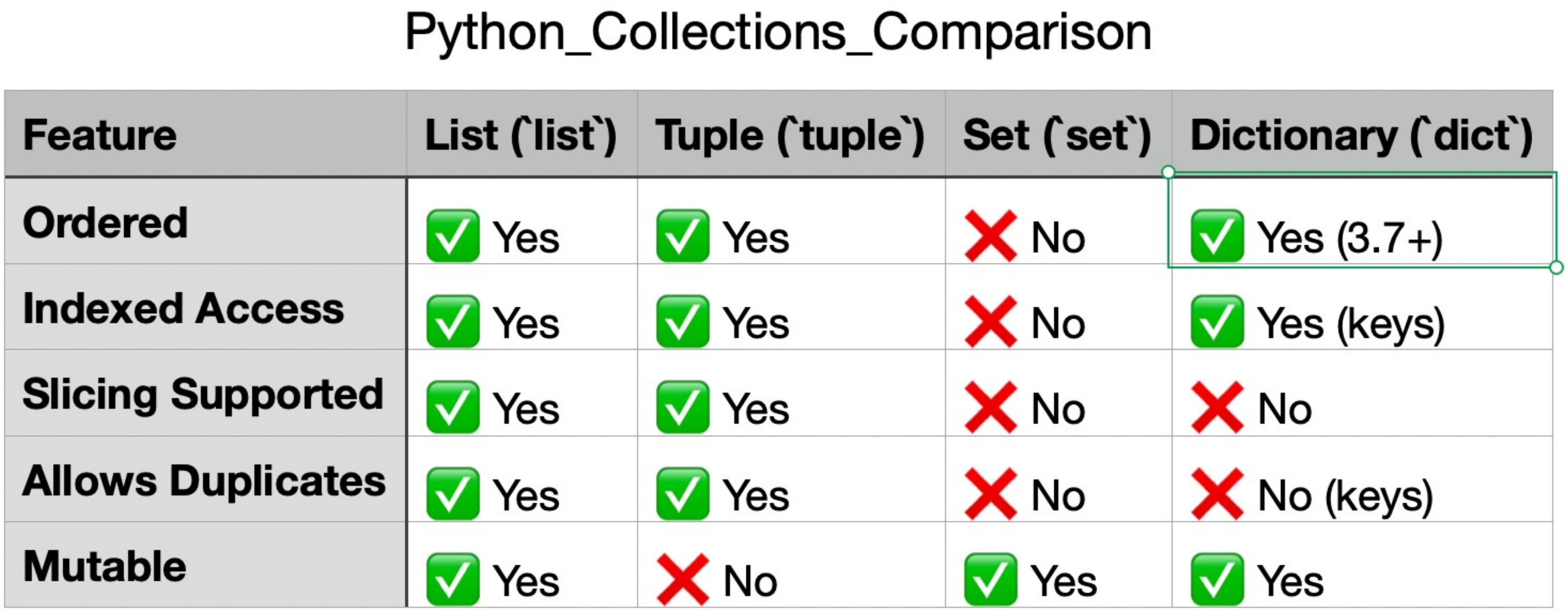
Dictionaries in Python

- Dictionaries are like ‘associative arrays’, **ordered (since Python 3.7) sequences** of **key-value pairs**, that do not allow duplicates.
- Indexed using **keys** that are of **immutable** types (strings, numbers or tuples)
- **Values** can be any type (lists, strings, other dictionaries, etc.).
- General format:

```
     dict = { key1:value1, key2:value2,...keyN:valueN }
```

- `list(dictionaryName)` returns list of keys
- Can be created using either {} or dict()
- The key and value can be retrieved at the same time using items() method

Type Comparison

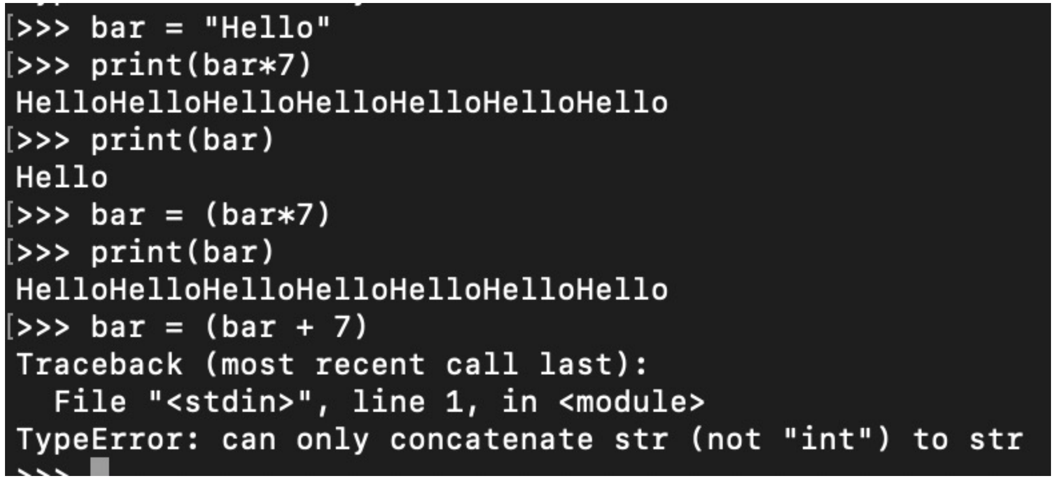


Python_Collections_Comparison

Feature	List (`list`)	Tuple (`tuple`)	Set (`set`)	Dictionary (`dict`)
Ordered	✅ Yes	✅ Yes	❌ No	✅ Yes (3.7+)
Indexed Access	✅ Yes	✅ Yes	❌ No	✅ Yes (keys)
Slicing Supported	✅ Yes	✅ Yes	❌ No	❌ No
Allows Duplicates	✅ Yes	✅ Yes	❌ No	❌ No (keys)
Mutable	✅ Yes	❌ No	✅ Yes	✅ Yes

Variable usage

```
foo = 25 # Numerical variable
bar = "Hello" # String variable

foo = (foo * 7) # Multiples foo by 7
bar = (bar * 7) # VALID expression (bar becomes
                # “HelloHelloHelloHelloHelloHelloHello”
bar = (bar + 7) # Invalid expression, throws error
```



```
[>>> bar = "Hello"
[>>> print(bar*7)
HelloHelloHelloHelloHelloHelloHello
[>>> print(bar)
Hello
[>>> bar = (bar*7)
[>>> print(bar)
HelloHelloHelloHelloHelloHelloHello
[>>> bar = (bar + 7)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only concatenate str (not "int") to str
>>>
```

‘print’ example

```
foo = 25           # Numerical variable
bar = "Hello"      # String variable
print(bar)         # Outputs Hello
print(foo,bar)     # Outputs 25 Hello
print("5x5=",foo) # Outputs 5x5= 25
print("5x5=foo")  # Outputs 5x5=foo
```

Notice how `print "5x5=foo"` outputs ‘foo’ rather than replacing it with 25

Arithmetic Operations

```
a = 15
b = 30
c = 2
total = a+b
a_squared = a**c  # 15**2
print(total)      # total is 45
print(a_squared)  # 225

a - b    # subtraction
a * b    # multiplication
a / b    # division
a += 5   # a=a+5 - also works for *= and /=
```

Concatenation

Use a ‘+’ to join strings into one.

```
string1 = "Hello"
string2 = "Python"
string3 = string1 + " " + string2
print(string3)
```

```
Output: Hello Python
```

Escaping Characters

- If the string has a set of double quotation marks that must remain visible, use the \ [backslash] before the quotation marks to ignore and display them.

```
heading = "\"Computer Science\""
print(heading)
```

```
Output: "Computer Science"
```

Python Control Structures

- Control Structures: the structures within a language that allow us to control the flow of execution through a program or script.
- Grouped into **conditional** / **branching** structures (e.g. if/else) and **repetition** structures (e.g. while loops).
- Example if/elif/else statement: [notice the ":"]

```
if foo == 0:
  print("The variable foo is equal to 0")
elif foo > 0 and foo <= 5:
  print("The variable foo is between 1 and 5")
else:
  print("The variable foo is equal to", foo)
```

If … Else…

```
If (condition):
      Statements
Else:
      Statement
```

No `Then` in Python !

Example:

```
if (user == "John"):
  print("Hello John.")
else:
  print("You are not John.")
```

While Loops

General format:

```
While (condition):
      Statements;
```

Example:

```
count = 0
while count < 3:
  print("hello Python. ")
  count += 1
  # count = count + 1
```

```
Output: hello Python. hello Python. hello Python.
```

For Loops and range()

- **Iterate** over a **sequence**
  - The built-in `range()` function helps iterate over a range of numbers

```
for i in range(5):
  print(i) # Prints 0,1,2,3 and 4
```

- **Iterate** over **elements** (for each)
  - Used with sequence type data-types like string, lists and tuples.

```
word = "Hello"
for letter in word:
  print(letter) # Prints Hello character by character
```

General format:
```
for condition:
         Statements
```

Date Display

```
import datetime
datedisplay = datetime.datetime.now()
print(datedisplay.strftime("%Y/%-m/%-d"))
# If the date is April 1st, 2012
# It would display as 2012/4/1
```

Output: 2012/4/1

```
datedisplay = datetime.datetime.now()
print(datedisplay.strftime("%A, %B %-d, %Y"))
# If the date is April 1st, 2012
# Wednesday, April 1, 2012
```

Output: Wednesday, April 1, 2012

Month, Day & Date Format Symbols

%b	Jan
%B	January
%m	01
%-m (for Linux) %#m (for Windows)	1

Day of Month	%d	01
Day of Month	%-d (for Linux) %#d (for Windows)	1
Day of Week	%A	Monday
Day of Week	%a	Mon

Functions

- Functions MUST be **defined** before they can be **called**
- Function headers are of the format [notice the ':']

```
def functionName(arg_1, arg_2, …, arg_n):
```

  - Note that **no return type** is specified
- Function names are case sensitive

```
(foo(…) != Foo(…) != FoO(…))
```

- Functions can have **default argument values**

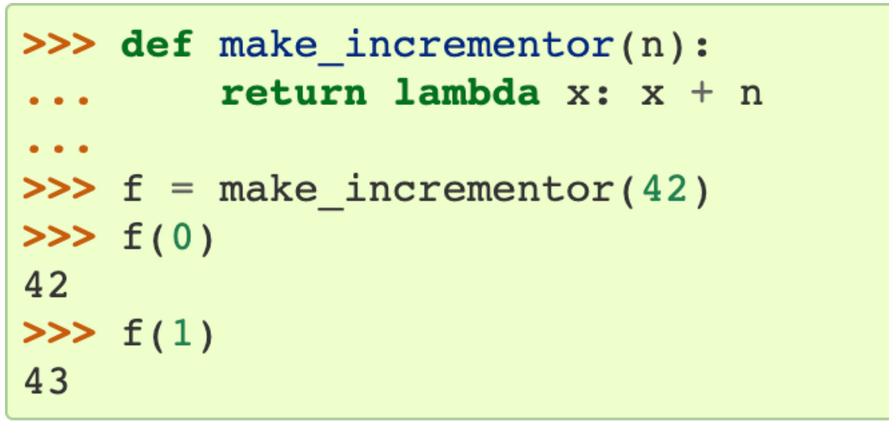
```
def functionName(a, b=2, c=[]):  # Note: Default argument
                                 # values are initialized
                                 # only once
```

Functions example

```
# This is a function

def foo(arg_1, arg_2):
  arg_2 = arg_1 * arg_2
  return arg_2


result_1 = foo(12, 3) # Store the function
print(result_1)       # Outputs 36
print(foo(12, 3))     # Outputs 36
```

Lambda Expressions

- Small anonymous functions
- Used wherever function objects are required
- Example:



```
>>> def make_incrementor(n):
...     return lambda x: x + n
...
>>> f = make_incrementor(42)
>>> f(0)
42
>>> f(1)
43
```

- In the above example, the ‘x’ represents the argument for the lambda function

Include Files

- Include “hello.py” within another python file as

```
import hello
from hello import Hello # imports the hello() from #hello.py
```

- The file hello.py might look like:

```
def Hello():
  print("Hello")
```

- In the aforementioned python file, the Hello() function can be called as

```
hello.Hello()
```

- Using ‘*’ allows importing all submodules from a package

```
from packageName import *
```

Classes in Python

- Syntax:

```
class className:
  statement
```

- To instantiate a class object, use function notation
- A constructor can be defined as

```
def __init__(self):
  statement
```

- The dot (‘.’) notation can be used to access class variables and methods
- **Inheritance** can be done as:

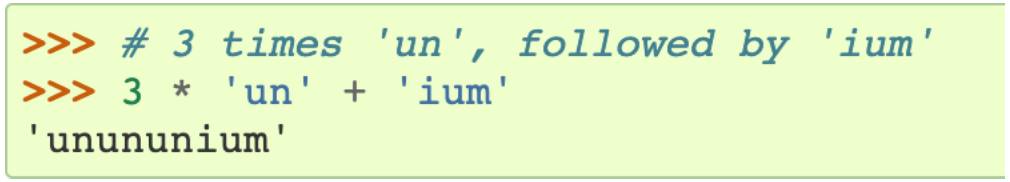
```
class DerivedClassName(moduleName.BaseClassName):
   statement
```

Code Examples

- All following code samples from “The Python Tutorial” at:

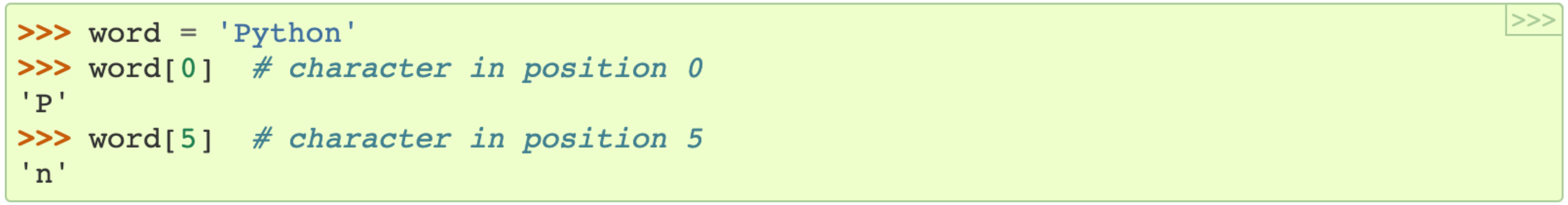
  https://docs.python.org/3/tutorial/index.html

String Examples



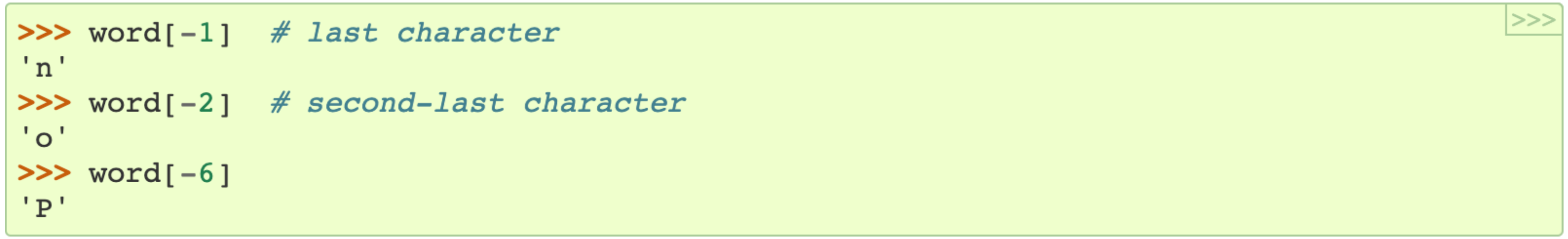
```
>>> # 3 times 'un', followed by 'ium'
>>> 3 * 'un' + 'ium'
'unununium'
```

Strings can be *indexed* (subscripted), with the first character having index 0. There is no separate character type; a character is simply a string of size one:



```
>>> word = 'Python'
>>> word[0]  # character in position 0
'P'
>>> word[5]  # character in position 5
'n'
```

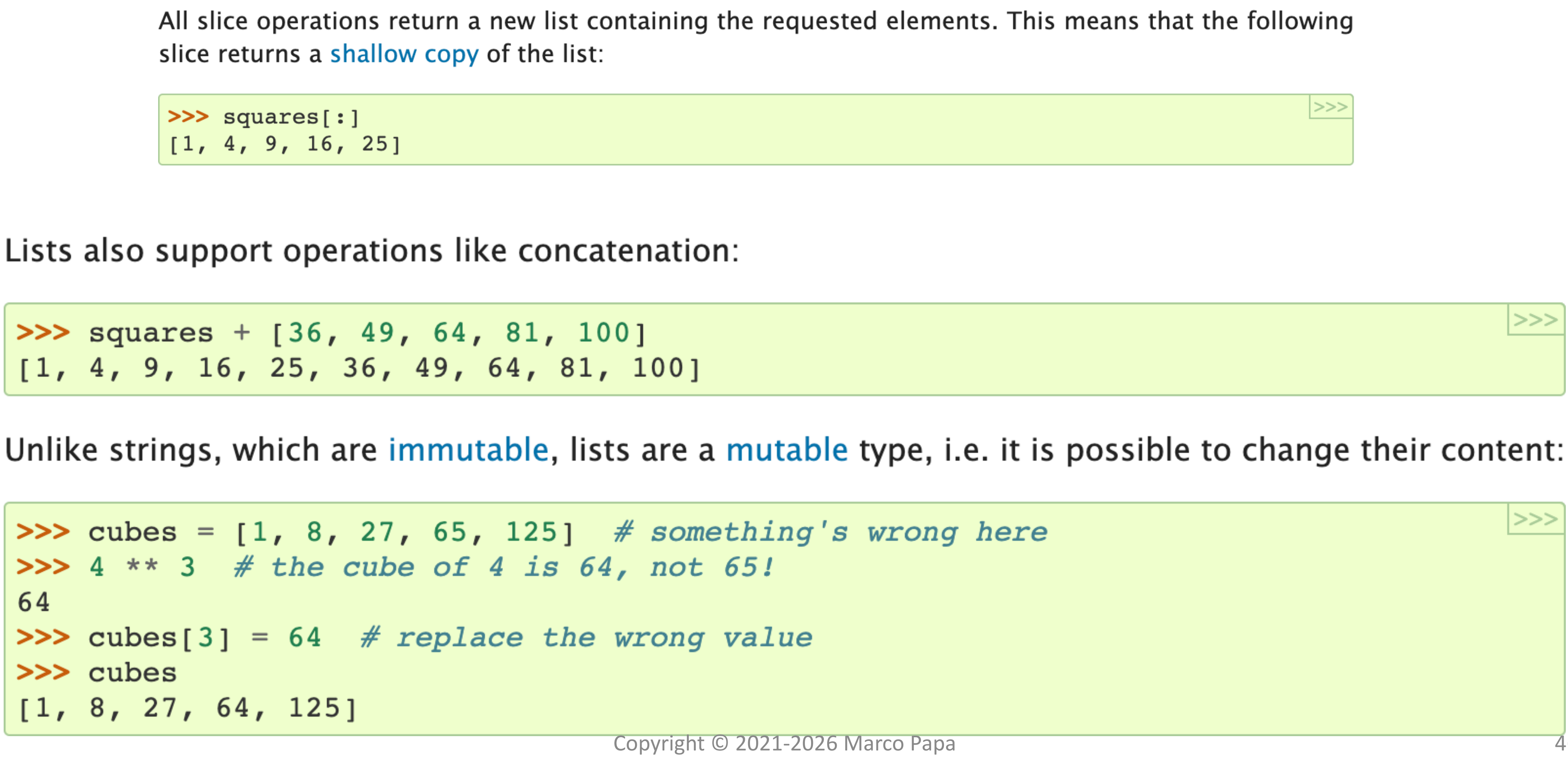
Indices may also be negative numbers, to start counting from the right:



```
>>> word[-1]  # last character
'n'
>>> word[-2]  # second-last character
'o'
>>> word[-6]
'P'
```

Note that since -0 is the same as 0, negative indices start from -1.

List Examples



All slice operations return a new list containing the requested elements. This means that the following slice returns a shallow copy of the list:

```
>>> squares[:]
[1, 4, 9, 16, 25]
```

Lists also support operations like concatenation:

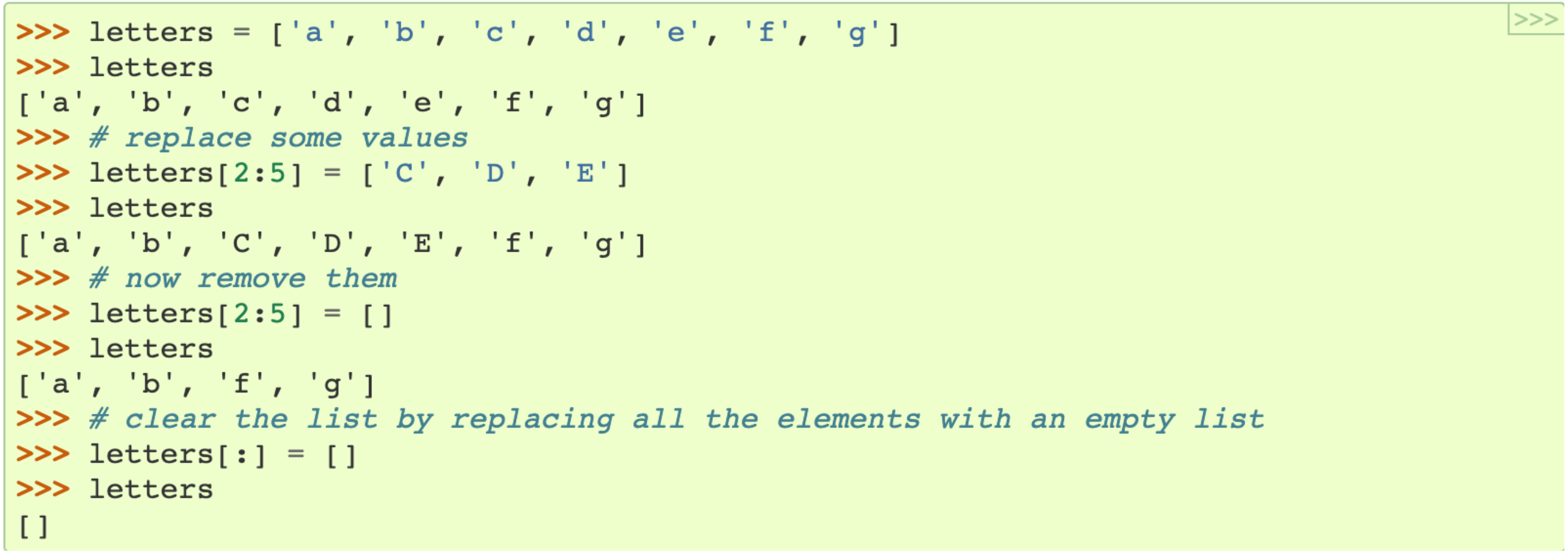
```
>>> squares + [36, 49, 64, 81, 100]
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

Unlike strings, which are immutable, lists are a mutable type, i.e. it is possible to change their content:

```
>>> cubes = [1, 8, 27, 65, 125]  # something's wrong here
>>> 4 ** 3  # the cube of 4 is 64, not 65!
64
>>> cubes[3] = 64  # replace the wrong value
>>> cubes
[1, 8, 27, 64, 125]
```

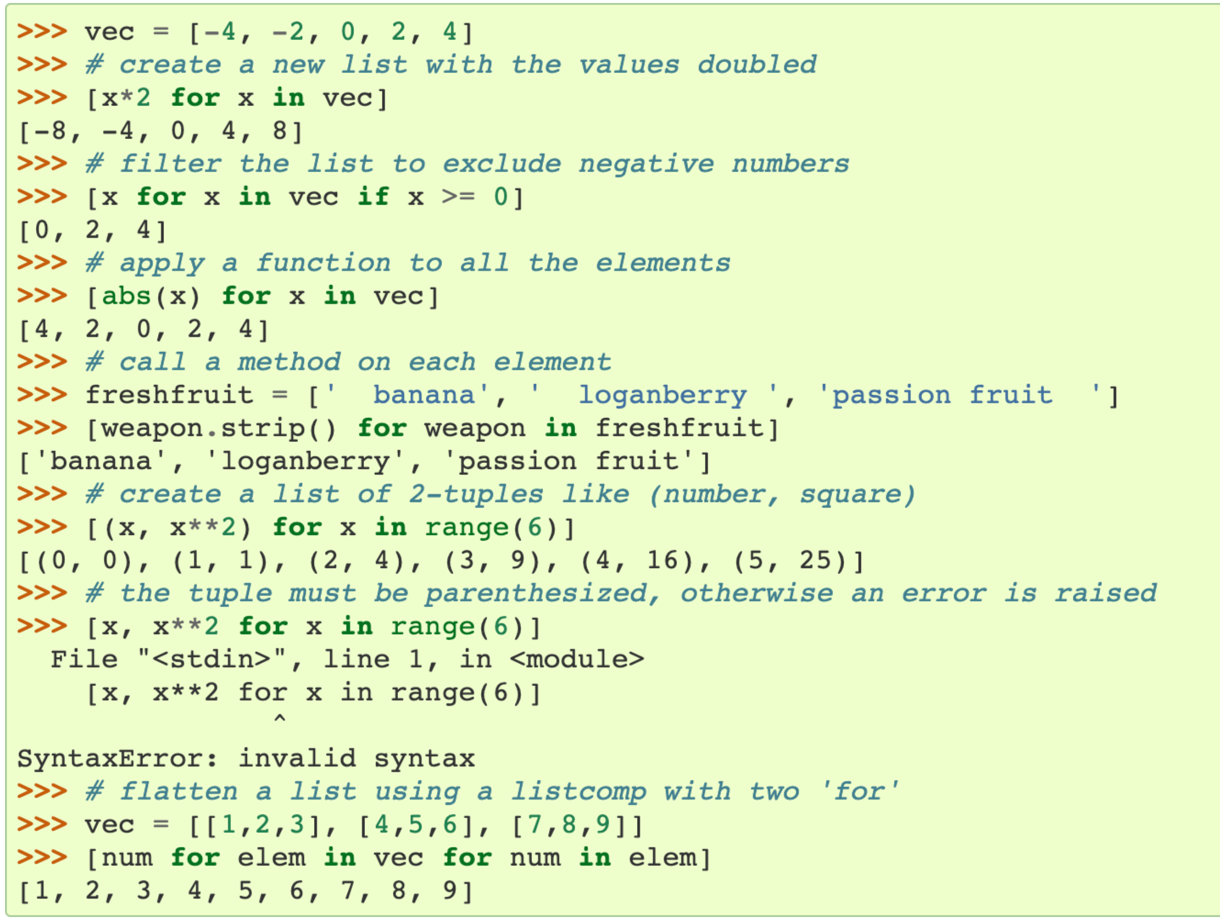
List Examples (cont’d)

Assignment to slices is also possible, and this can even change the size of the list or clear it entirely:



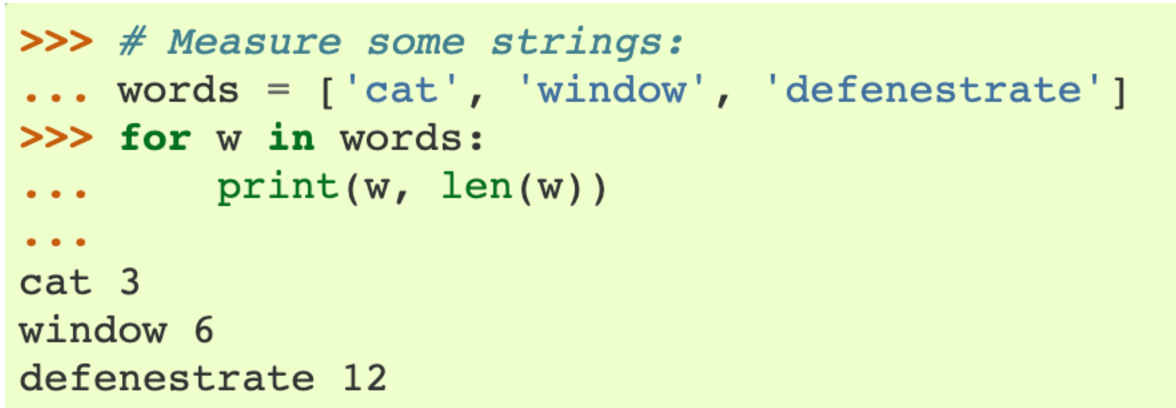
```
>>> letters = ['a', 'b', 'c', 'd', 'e', 'f', 'g']
>>> letters
['a', 'b', 'c', 'd', 'e', 'f', 'g']
>>> # replace some values
>>> letters[2:5] = ['C', 'D', 'E']
>>> letters
['a', 'b', 'C', 'D', 'E', 'f', 'g']
>>> # now remove them
>>> letters[2:5] = []
>>> letters
['a', 'b', 'f', 'g']
>>> # clear the list by replacing all the elements with an empty list
>>> letters[:] = []
>>> letters
[]
```

List Examples (cont’d)



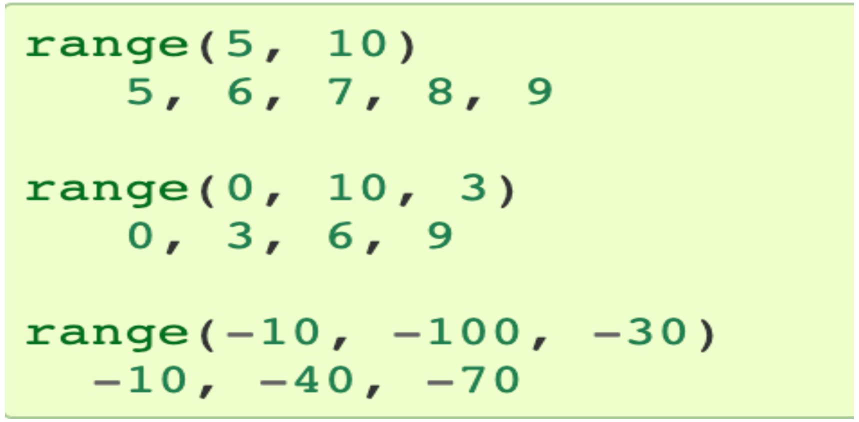
```
>>> vec = [-4, -2, 0, 2, 4]
>>> # create a new list with the values doubled
>>> [x*2 for x in vec]
[-8, -4, 0, 4, 8]
>>> # filter the list to exclude negative numbers
>>> [x for x in vec if x >= 0]
[0, 2, 4]
>>> # apply a function to all the elements
>>> [abs(x) for x in vec]
[4, 2, 0, 2, 4]
>>> # call a method on each element
>>> freshfruit = ['  banana', '  loganberry ', 'passion fruit  ']
>>> [weapon.strip() for weapon in freshfruit]
['banana', 'loganberry', 'passion fruit']
>>> # create a list of 2-tuples like (number, square)
>>> [(x, x**2) for x in range(6)]
[(0, 0), (1, 1), (2, 4), (3, 9), (4, 16), (5, 25)]
>>> # the tuple must be parenthesized, otherwise an error is raised
>>> [x, x**2 for x in range(6)]
  File "<stdin>", line 1, in <module>
    [x, x**2 for x in range(6)]
               ^

SyntaxError: invalid syntax
>>> # flatten a list using a listcomp with two 'for'
>>> vec = [[1,2,3], [4,5,6], [7,8,9]]
>>> [num for elem in vec for num in elem]
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Looping examples



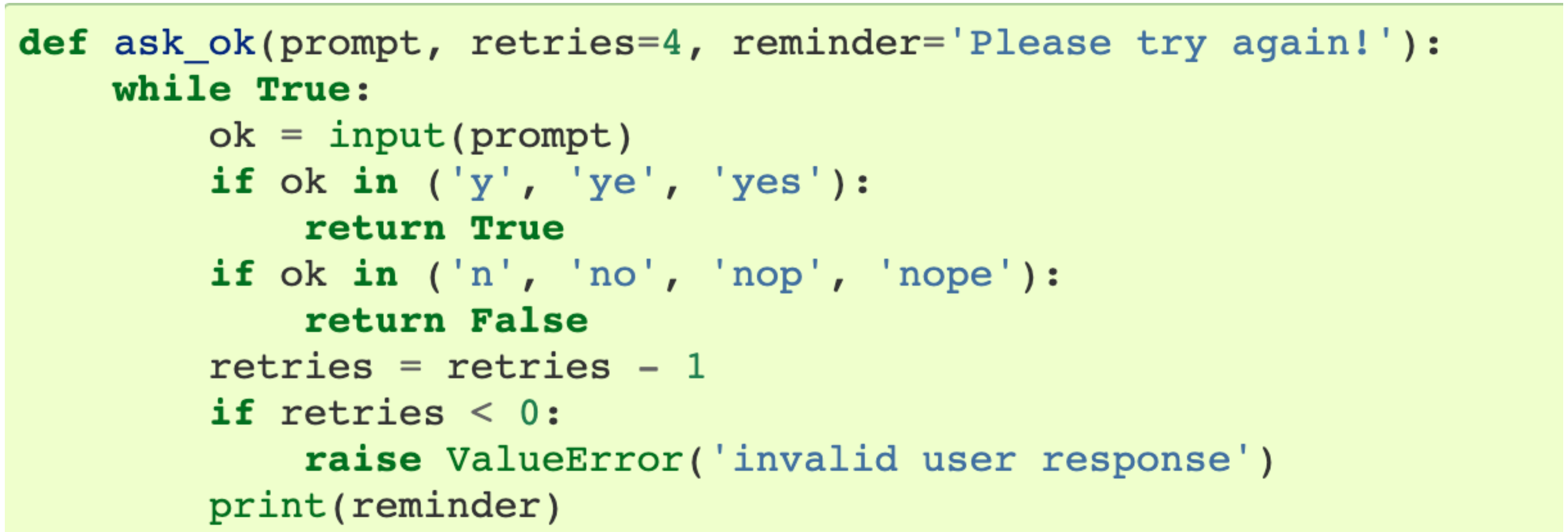
```
>>> # Measure some strings:
... words = ['cat', 'window', 'defenestrate']
>>> for w in words:
...     print(w, len(w))
...
cat 3
window 6
defenestrate 12
```

Use of range



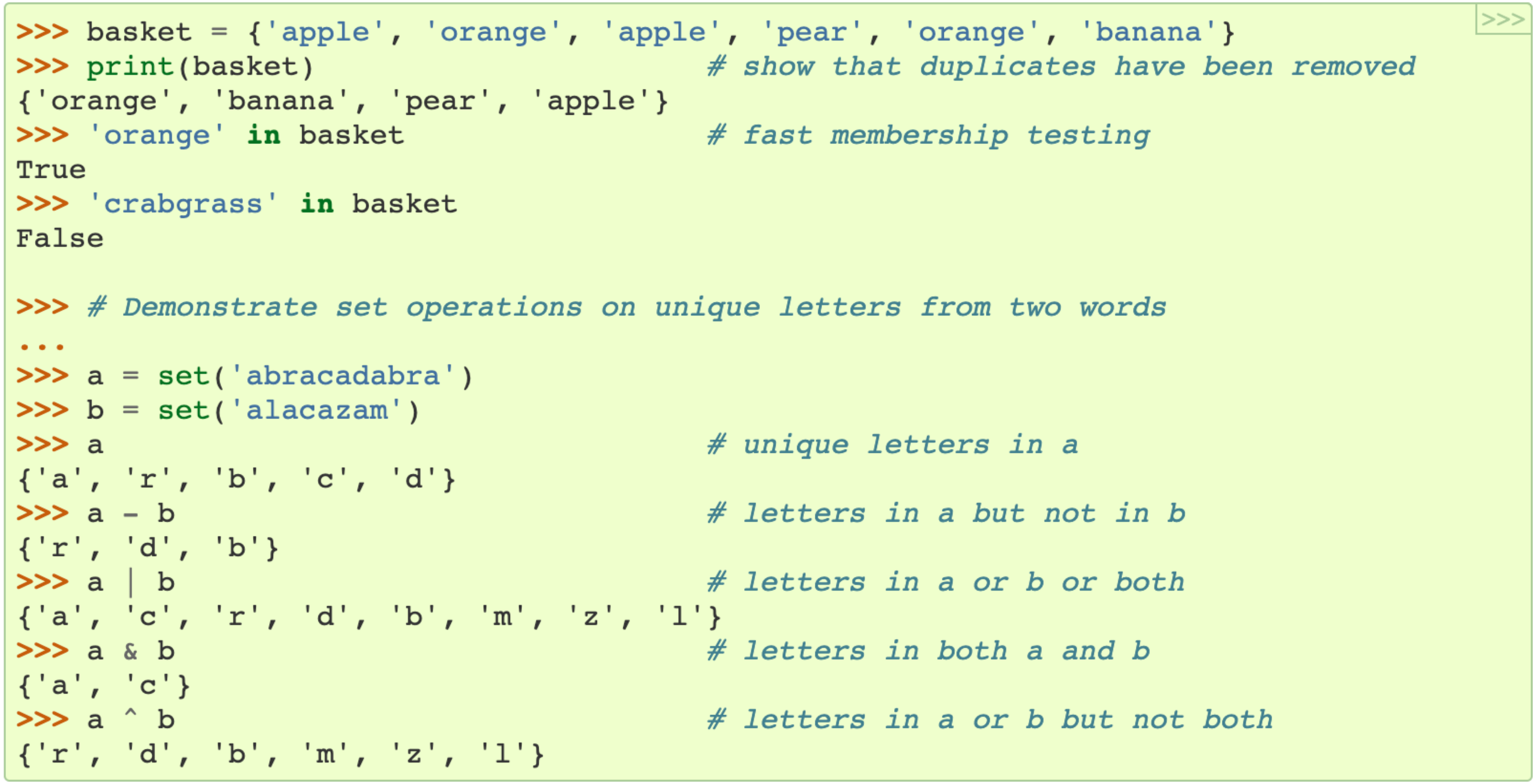
```
range(5, 10)
    5, 6, 7, 8, 9

range(0, 10, 3)
    0, 3, 6, 9

range(-10, -100, -30)
   -10, -40, -70
```

Function example



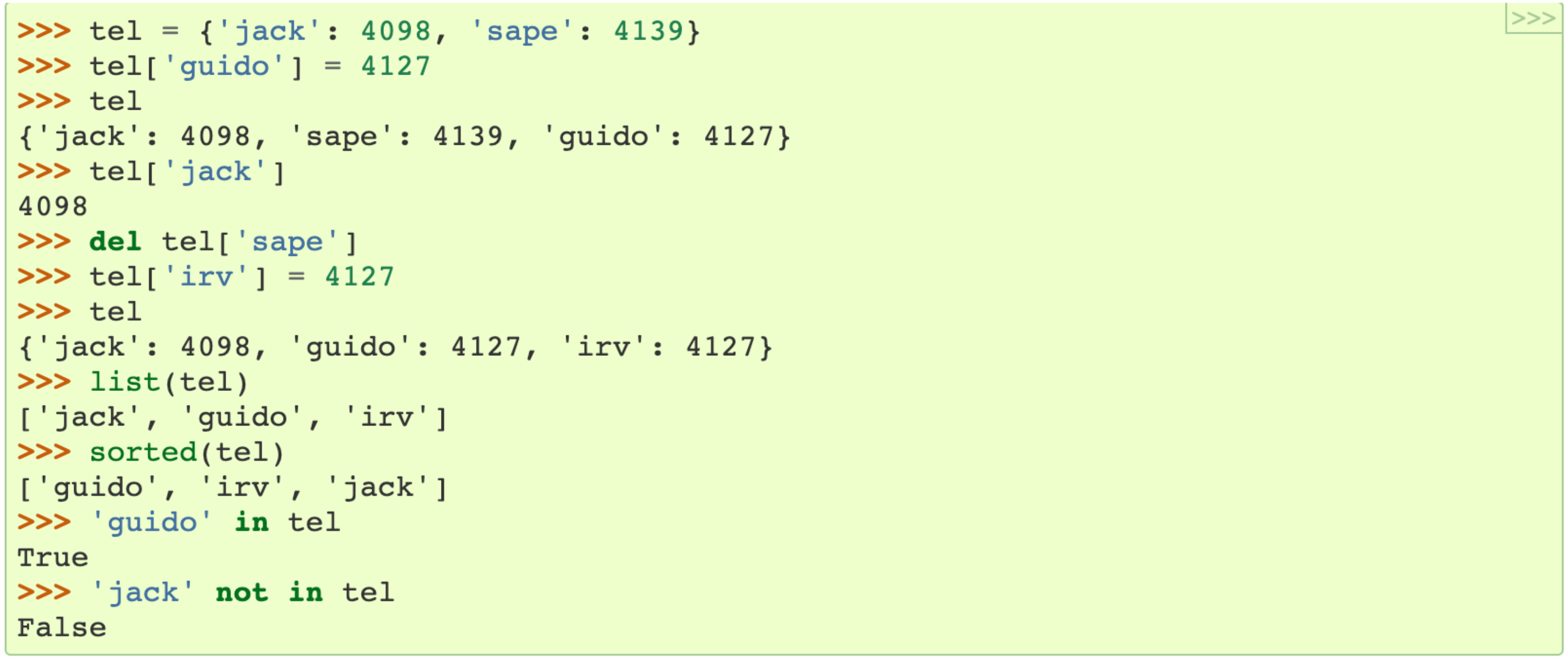
```python
def ask_ok(prompt, retries=4, reminder='Please try again!'):
    while True:
        ok = input(prompt)
        if ok in ('y', 'ye', 'yes'):
            return True
        if ok in ('n', 'no', 'nop', 'nope'):
            return False
        retries = retries - 1
        if retries < 0:
            raise ValueError('invalid user response')
        print(reminder)
```

Sets Examples



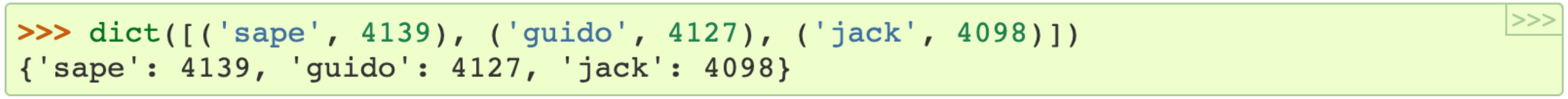
```
>>> basket = {'apple', 'orange', 'apple', 'pear', 'orange', 'banana'}
>>> print(basket)                      # show that duplicates have been removed
{'orange', 'banana', 'pear', 'apple'}
>>> 'orange' in basket                 # fast membership testing
True
>>> 'crabgrass' in basket
False

>>> # Demonstrate set operations on unique letters from two words
...
>>> a = set('abracadabra')
>>> b = set('alacazam')
>>> a                                  # unique letters in a
{'a', 'r', 'b', 'c', 'd'}
>>> a - b                              # letters in a but not in b
{'r', 'd', 'b'}
>>> a | b                              # letters in a or b or both
{'a', 'c', 'r', 'd', 'b', 'm', 'z', 'l'}
>>> a & b                              # letters in both a and b
{'a', 'c'}
>>> a ^ b                              # letters in a or b but not both
{'r', 'd', 'b', 'm', 'z', 'l'}
```

Dictionary Examples



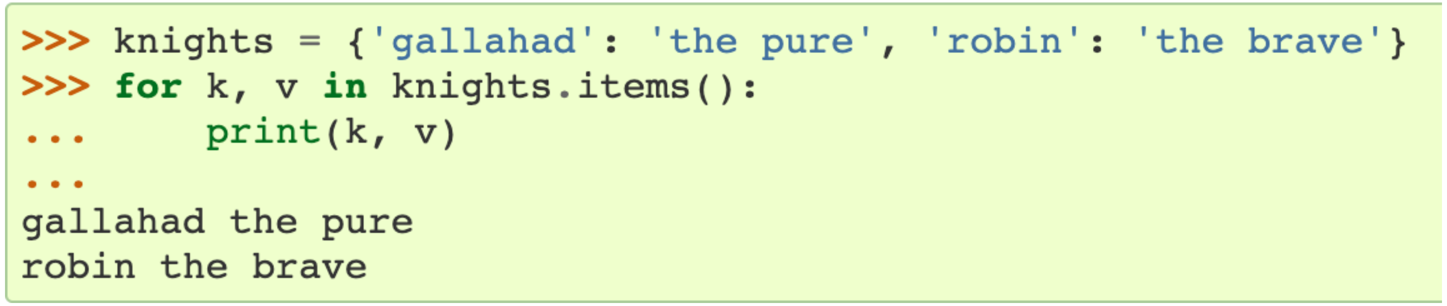
```
>>> tel = {'jack': 4098, 'sape': 4139}
>>> tel['guido'] = 4127
>>> tel
{'jack': 4098, 'sape': 4139, 'guido': 4127}
>>> tel['jack']
4098
>>> del tel['sape']
>>> tel['irv'] = 4127
>>> tel
{'jack': 4098, 'guido': 4127, 'irv': 4127}
>>> list(tel)
['jack', 'guido', 'irv']
>>> sorted(tel)
['guido', 'irv', 'jack']
>>> 'guido' in tel
True
>>> 'jack' not in tel
False
```

The `dict()` constructor builds dictionaries directly from sequences of key-value pairs:

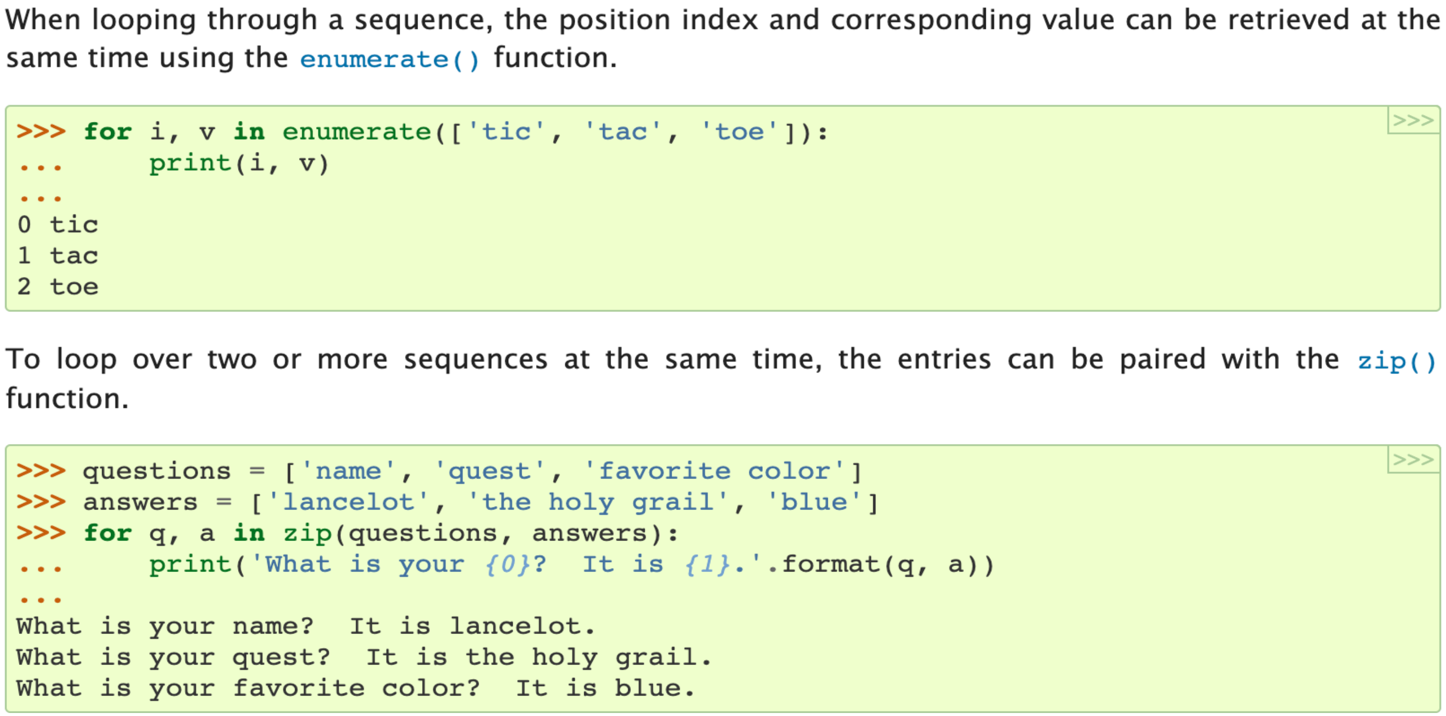


```
>>> dict([('sape', 4139), ('guido', 4127), ('jack', 4098)])
{'sape': 4139, 'guido': 4127, 'jack': 4098}
```

Looping Techniques



```
>>> knights = {'gallahad': 'the pure', 'robin': 'the brave'}
>>> for k, v in knights.items():
...     print(k, v)
...
gallahad the pure
robin the brave
```



When looping through a sequence, the position index and corresponding value can be retrieved at the same time using the `enumerate()` function.

```
>>> for i, v in enumerate(['tic', 'tac', 'toe']):
...     print(i, v)
...
0 tic
1 tac
2 toe
```

To loop over two or more sequences at the same time, the entries can be paired with the `zip()` function.

```
>>> questions = ['name', 'quest', 'favorite color']
>>> answers = ['lancelot', 'the holy grail', 'blue']
>>> for q, a in zip(questions, answers):
...     print('What is your {0}?  It is {1}.'.format(q, a))
...
What is your name?  It is lancelot.
What is your quest?  It is the holy grail.
What is your favorite color?  It is blue.
```

Handling Exceptions

- Use `try … except` statement for exception handling
- If no exception occurs, **except** clause will be skipped. Else, the **try** clause is stopped, and the matched exception clause will be executed.

```
while True:
  try:
    x = int(input("Please enter a number: "))
    break
  except ValueError:
    print("Oops! That was no valid number. Try again...")
```

Import Module

- A module is a file containing Python definitions and statement with the suffix ‘.py’ appended
- Import a module by its name

```
import fibo
```

- Import the methods from a module

```
from fibo import fib, fib2
```

- Import all that a module defines

```
from fibo import *
```

Flask

- Flask is a lightweight **WSGI** (**Web Server Gateway Interface**) web application framework
- WSGI is a Python standard defined in PEP 3333

Python Web Server Gateway Interface v1.0.1

- PEP 3333 dds Python 3 compatibility
- WSGI specifies a standard interface between web servers and Python web applications or frameworks
- Flask is designed to make getting started quickly and easily, with the ability to scale up to complex applications
- Flask offers suggestions but doesn't enforce any dependencies or project layout
- Documentation at: https://palletsprojects.com/p/flask/

Installation

- Use native virtual environment for Python3

```
$ python3 -m venv venv
```

or

```
$ pip3 install virtualenv
```

- Use third party for any version of Python older than 3.4 (includes 2.7)

```
$ virtualenv venv
$ source venv/bin/activate
```

- Install Flask in venv

```
(venv) $ pip3 install flask
```

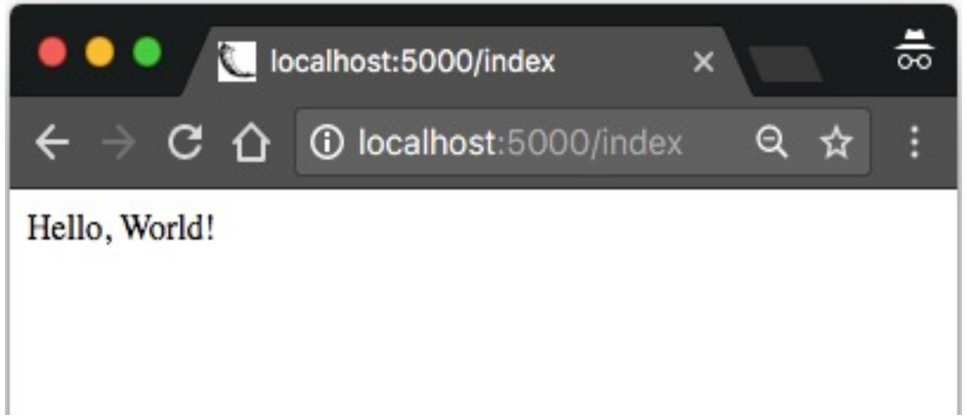
Flask Hello World

hello. py

```
from flask import Flask
app = Flask(__name__)


@app.route('/')
def hello_world():
    return "Hello, World"
```

Flask Hello World (cont’d)

```
(venv) $ export FLASK_APP=hello.py
(venv) $ flask run
Serving Flask app ”hello_world" *
Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
```

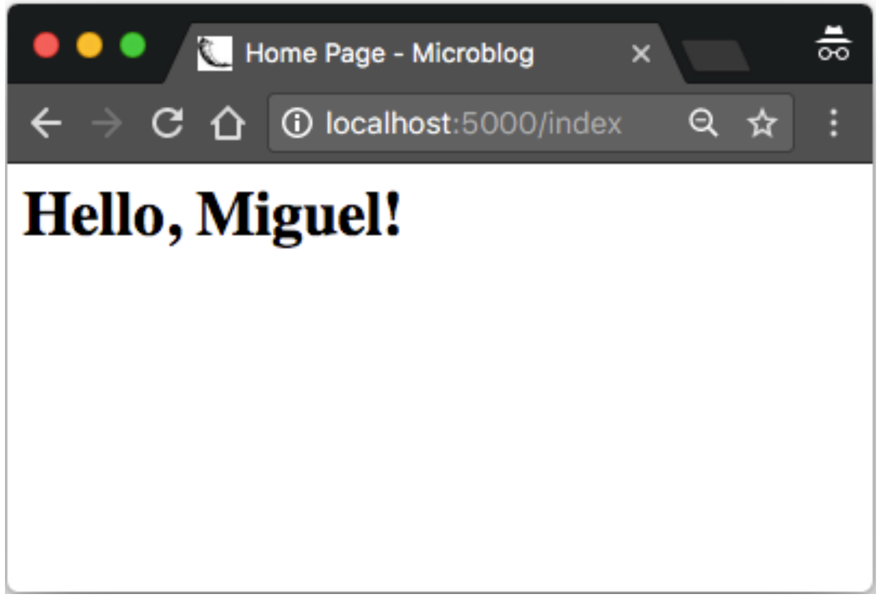


Hello, World!

Templates

app/routes.py

```
from app import app

@app.route('/')
@app.route('/index')
def index():
    user = {'username': 'Miguel’}
    return '''<html>
        <head> <title>Home Page - Microblog</title> </head>
        <body>
            <h1>Hello, ''' + user['username'] + '''!</h1>
        </body> </html>'''
```



Hello, Miguel!

render_template

app/routes.py: Fake post in view function

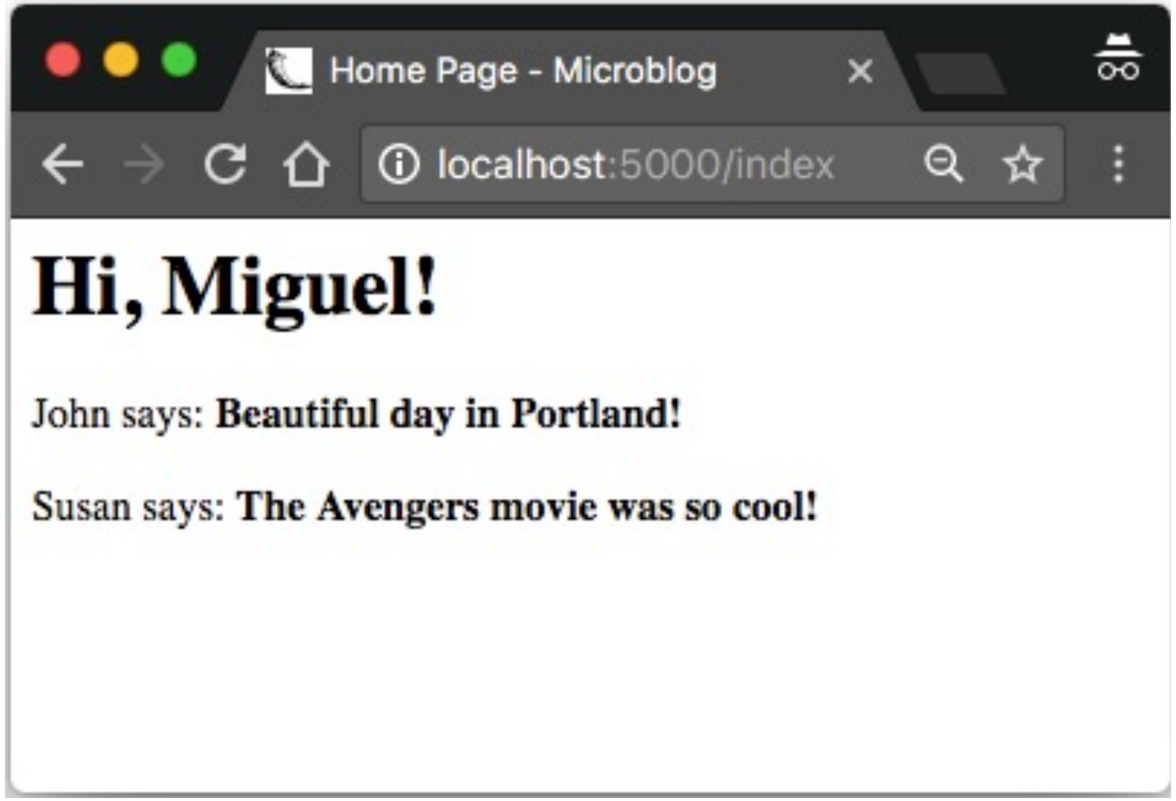
```
from flask import render_template
from app import app

@app.route('/')
@app.route('/index')
def index():
  user = {'username': 'Miguel’}
  posts = [
    {'author': {'username': 'John'},'body': 'Beautiful day in Portland!'},
    {'author': {'username': 'Susan'}, 'body': 'The Avengers was so cool!’}
  ]
  return render_template('index.html', title='Home', user=user, posts=posts)
```

render_template (con’d)

app/templates/index.html

```
<html> <head>
    {% if title %}
        <title>{{ title }} - Microblog</title>
    {% else %}
        <title>Welcome to Microblog</title>
    {% endif %}
</head>
<body>
    <h1>Hi, {{ user.username }}!</h1>
    {% for post in posts %}
        <div><p>{{ post.author.username }} says:
             <b>{{ post.body }}</b></p></div>
    {% endfor %}
</body> </html>
```

Web Output



Hi, Miguel!

John says: **Beautiful day in Portland!**

Susan says: **The Avengers movie was so cool!**

Templates are old tech

Pre-Ajax coding patterns:

- Python templates + HTML {% … %}
- PHP + (Twig?) + HTML <? … ?>
- ASP (Active Server Pages) + HTML <% …%>
- JSP (Java Server Pages) + HTML <% … %>

Post-Ajax coding patterns: All **RESTful APIs**, returning data only (JSON, XML) and no HTML

Do not use render_template()!

Except `render_template('index.html’); containing no Python`

RESTful Service in Flask

rest.py

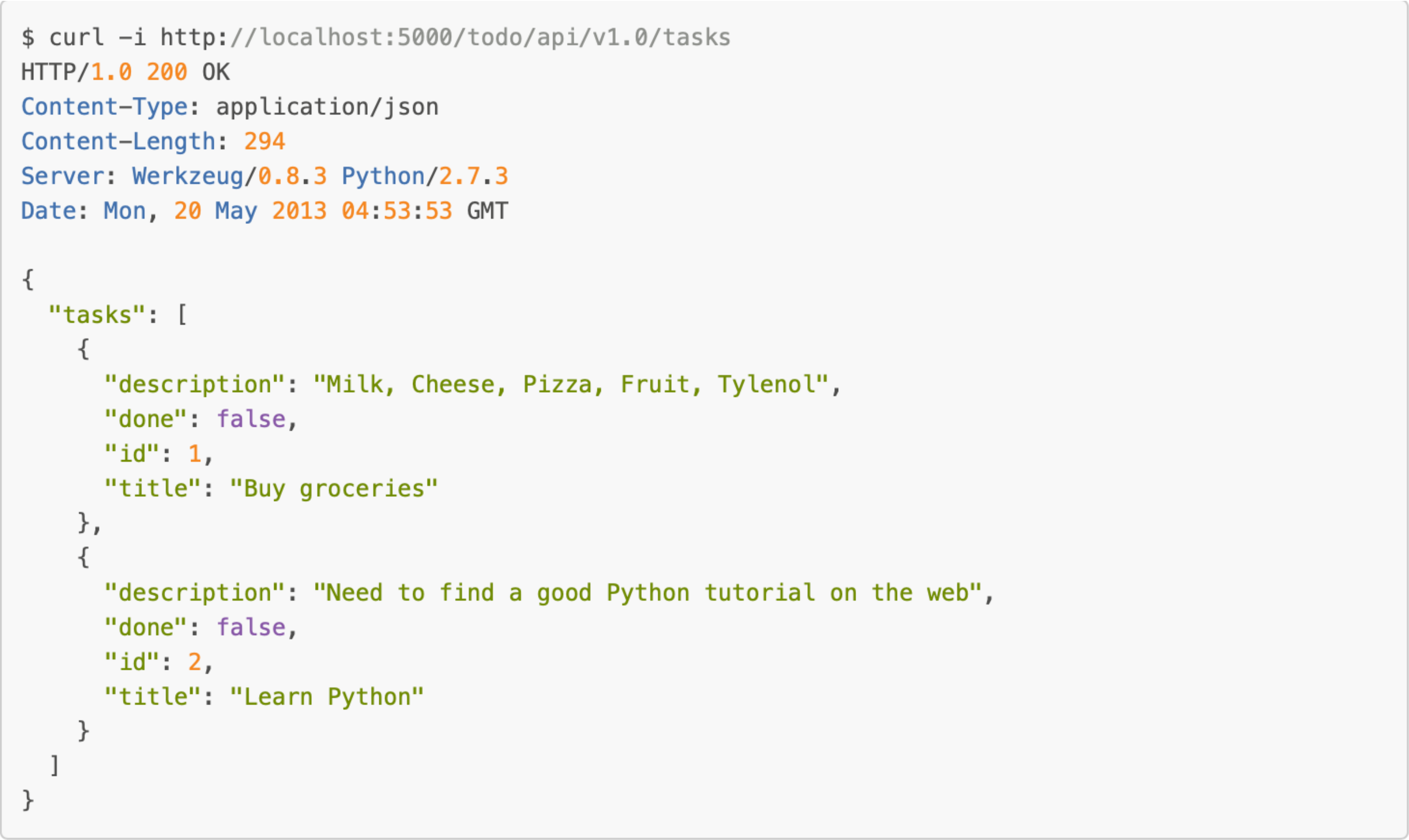
```python
from flask import Flask, jsonify

app = Flask(__name__)

tasks = [
{ 'id': 1, 'title': u'Buy groceries', 'description’:
  u'Milk, Cheese, Pizza, Fruit, Tylenol', 'done': False },
{ 'id': 2, 'title': u'Learn Python', 'description': u'Need
  to find a good Python tutorial on the web', 'done': False } ]

if __name__ == '__main__’:
  app.run(debug=True)
```

RESTful Service in Flask

```
# retrieve the list of task
@app.route('/todo/api/v1.0/tasks', methods=['GET'])
def get_tasks():
  return jsonify({'tasks': tasks})
```

See: https://flask.palletsprojects.com/en/stable/api/#flask.json.jsonify

Result

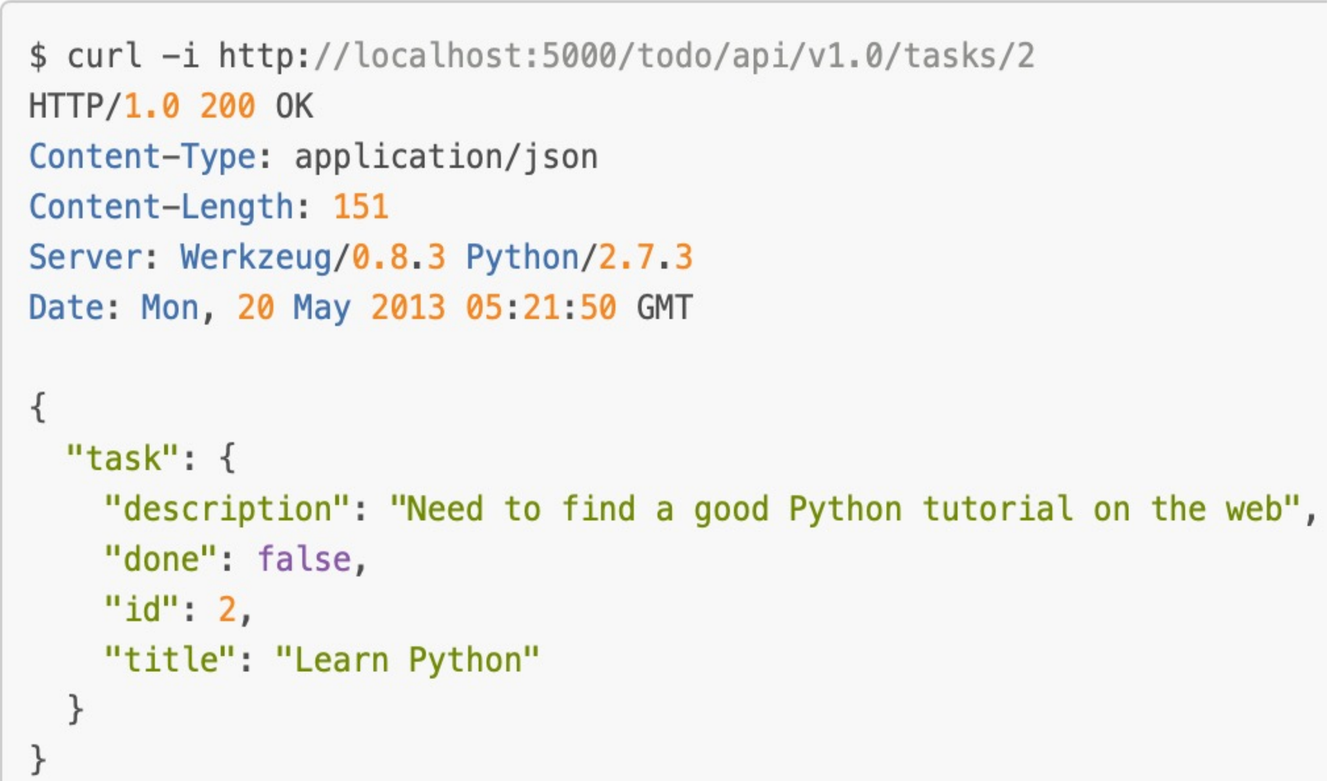


```
$ curl -i http://localhost:5000/todo/api/v1.0/tasks
HTTP/1.0 200 OK
Content-Type: application/json
Content-Length: 294
Server: Werkzeug/0.8.3 Python/2.7.3
Date: Mon, 20 May 2013 04:53:53 GMT

{
  "tasks": [
    {
      "description": "Milk, Cheese, Pizza, Fruit, Tylenol",
      "done": false,
      "id": 1,
      "title": "Buy groceries"
    },
    {
      "description": "Need to find a good Python tutorial on the web",
      "done": false,
      "id": 2,
      "title": "Learn Python"
    }
  ]
}
```

RESTful Service in Flask (2)

```
from flask import abort

# retrieve a task
@app.route('todo/api/v1.0/tasks/<int:task_id>', methods=['GET'])
def get_task(task_id):
    task= [task for task in tasks if task['id'] == task_id]
    if len(task) == 0:
        abort(404)
    return jsonify({'task': task[0]})
```

Result



```
$ curl -i http://localhost:5000/todo/api/v1.0/tasks/2
HTTP/1.0 200 OK
Content-Type: application/json
Content-Length: 151
Server: Werkzeug/0.8.3 Python/2.7.3
Date: Mon, 20 May 2013 05:21:50 GMT

{
  "task": {
    "description": "Need to find a good Python tutorial on the web",
    "done": false,
    "id": 2,
    "title": "Learn Python"
  }
}
```

Send Static File

- Put index.html into the **static** folder (same for CSS and JS files)
- Send the static file using `send_static_file`
- See: https://flask.palletsprojects.com/en/stable/api/#flask.Flask.send_static_file
- Can also use `send_from_directory`
- See: https://flask.palletsprojects.com/en/stable/api/#flask.send_from_directory

```
app = Flask(__name__)

@app.route('/')
def homepage():
    return app.send_static_file("index.html")
```

Requests: HTTP for Humans

- Simple HTTP library for Python
- See: https://requests.readthedocs.io/en/master/
- Supports Python 3.8–3.14

```
>>> import requests
>>> payload = {'key1': 'value1', 'key2': 'value2’}
>>> r = requests.get('https://api.github.com/events’,
params=payload)
>>> r.json()
[{'repository': {'open_issues': 0, 'url': 'https://github.com/...
```

FastAPI

- FastAPI is a modern, high-performance web framework for building APIs.
- Fast API implements **ASGI** (Asynchronous Server Gateway Interface)
- ASGI official documentation at: https://asgi.readthedocs.io
- It is built on Starlette and Pydantic for speed and efficiency.
- Uses Python's type hints for data validation and auto-generated docs.
- Asynchronous (`async/await`) support for non-blocking API execution.
- Designed for high-performance applications, WebSocket connections, async I/O, HTTP 2, rivaling Node.js & Go.
- Documentation at: https://fastapi.tiangolo.com/

FastAPI Features and Advantages

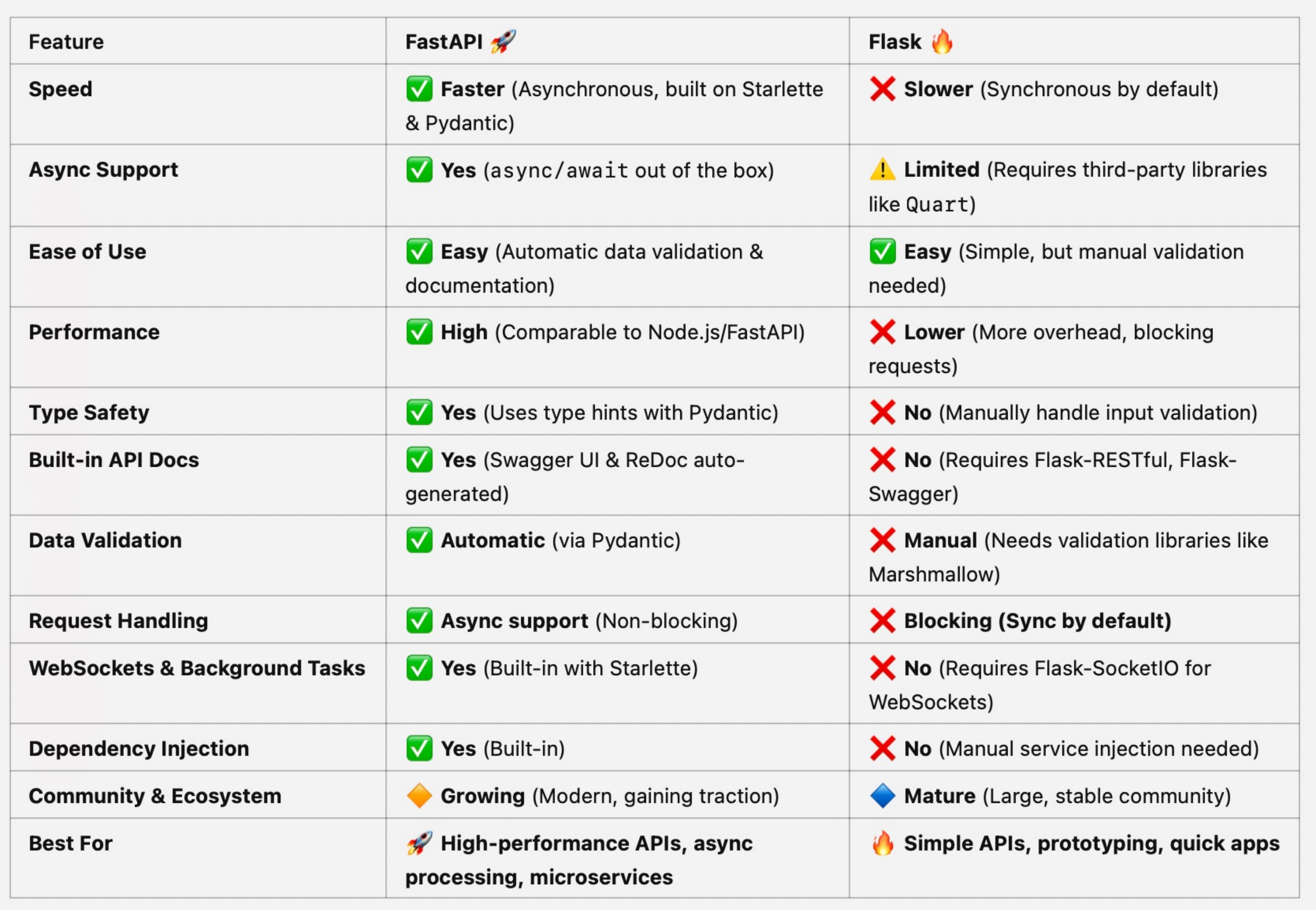
- Features
  - High Performance - Comparable to Node.js & Go.
  - Auto-Generated API Docs - Swagger UI & ReDoc.
  - Type Safety & Validation - Uses Pydantic models.
  - Asynchronous Execution - `async/await` support.
  - Dependency Injection - Efficient resource management.
- Advantages
  - Blazing Fast - One of the fastest Python web frameworks.
  - Developer-Friendly - Less code, easy to maintain.
  - Asynchronous & Concurrent - Handles thousands of requests.
  - Secure & Scalable - Ideal for microservices & cloud deployments.

RESTful Service in FastAPI

rest.py

```
from FASTAPI import FastAPI

app = FastAPI()

@app.get("/")
async def root
 return {"message": "Hello World"}
```

FastAPI vs. Flask: Key Differences



Feature	FastAPI 🚀	Flask 🔥
Speed	✅ **Faster** (Asynchronous, built on Starlette & Pydantic)	❌ **Slower** (Synchronous by default)
Async Support	✅ **Yes** (`async`/`await` out of the box)	⚠️ **Limited** (Requires third-party libraries like `Quart`)
Ease of Use	✅ **Easy** (Automatic data validation & documentation)	✅ **Easy** (Simple, but manual validation needed)
Performance	✅ **High** (Comparable to Node.js/FastAPI)	❌ **Lower** (More overhead, blocking requests)
Type Safety	✅ **Yes** (Uses type hints with Pydantic)	❌ **No** (Manually handle input validation)
Built-in API Docs	✅ **Yes** (Swagger UI & ReDoc auto-generated)	❌ **No** (Requires Flask-RESTful, Flask-Swagger)
Data Validation	✅ **Automatic** (via Pydantic)	❌ **Manual** (Needs validation libraries like Marshmallow)
Request Handling	✅ **Async support** (Non-blocking)	❌ **Blocking (Sync by default)**
WebSockets & Background Tasks	✅ **Yes** (Built-in with Starlette)	❌ **No** (Requires Flask-SocketIO for WebSockets)
Dependency Injection	✅ **Yes** (Built-in)	❌ **No** (Manual service injection needed)
Community & Ecosystem	🔶 **Growing** (Modern, gaining traction)	🔷 **Mature** (Large, stable community)
Best For	🚀 **High-performance APIs, async processing, microservices**	🔥 **Simple APIs, prototyping, quick apps**

Django

- Django is a high-level Python Web framework that encourages rapid development and clean, pragmatic design.
- Django was designed to help developers take applications from concept to completion as quickly as possible.
- Django takes security seriously and helps developers avoid many common security mistakes.
- Some of the busiest sites on the Web leverage Django’s ability to quickly and flexibly scale.
- Django supports both WSGI and ASGI
- Documentation at: https://www.djangoproject.com/

QuickStart

- Installation

```
$ python -m pip install Django
```

- Create a Django project

```
$ django-admin startproject mysite
```

- manage.py:

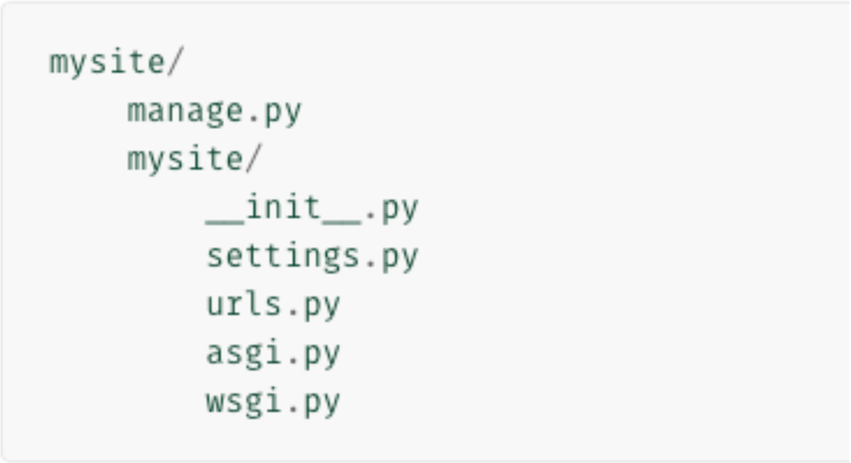
  A command-line utility that lets you interact with this Django project in various ways.

- mysite/settings.py:

  Settings/configuration for this Django project.

- mysite/urls.py:

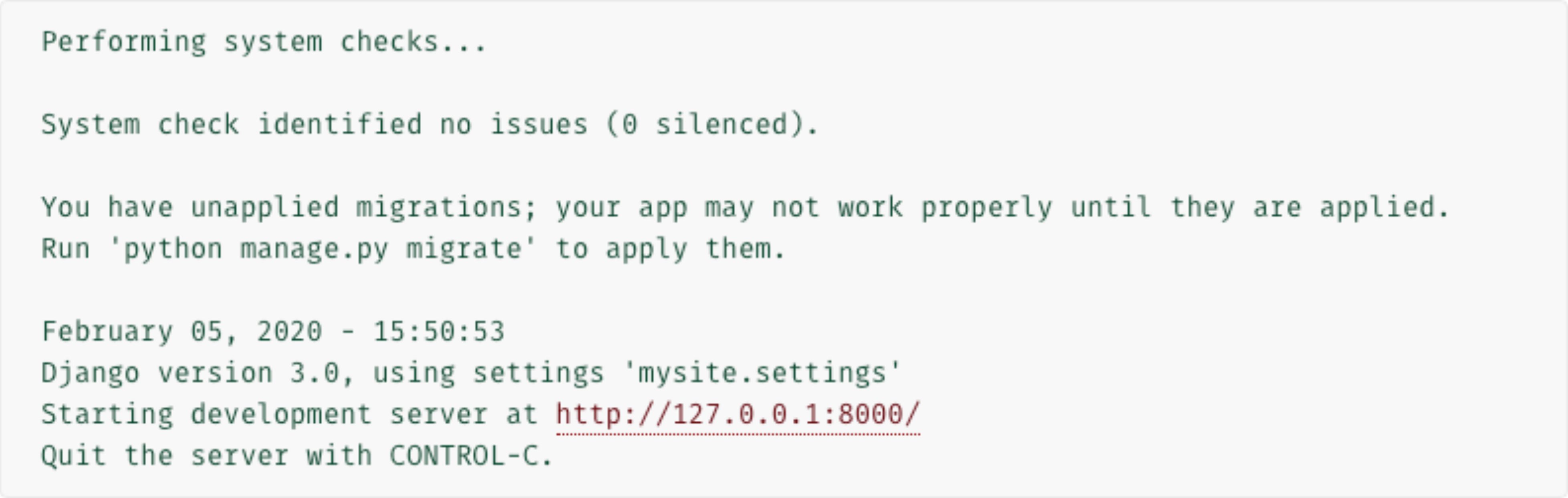
  The URL declarations for this Django project; a “table of contents” of your Django-powered site.



```
mysite/
    manage.py
    mysite/
        __init__.py
        settings.py
        urls.py
        asgi.py
        wsgi.py
```

Development Server

```
$ python manage.py runserver
```



```
Performing system checks...

System check identified no issues (0 silenced).

You have unapplied migrations; your app may not work properly until they are applied.
Run 'python manage.py migrate' to apply them.

February 05, 2020 - 15:50:53
Django version 3.0, using settings 'mysite.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CONTROL-C.
```

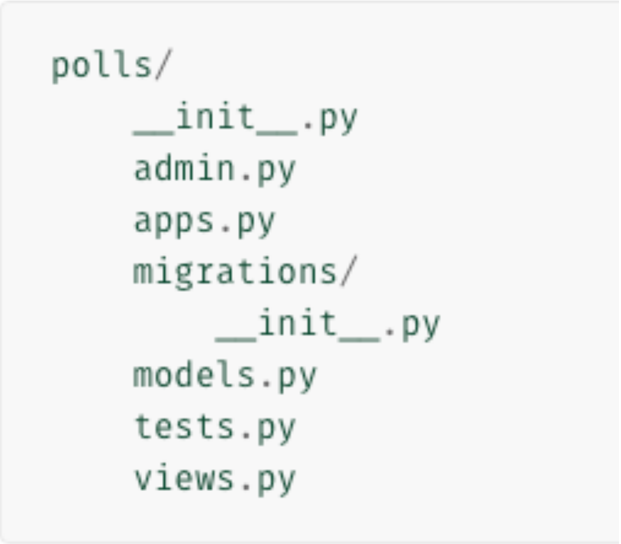
Creating the First App

- Create the polls app

```
$ python manage.py startapp polls
```

- Edit views.py

```
from django.http import HttpResponse

def index(request):
  return HttpResponse("Hello, world. You're at the polls index.")
```



```
polls/
    __init__.py
    admin.py
    apps.py
    migrations/
        __init__.py
    models.py
    tests.py
    views.py
```

Change the URL Config

polls/urls.py

```
from django.urls import path
from . import views
urlpatterns = [
    path('',views.index,name='index'),
]
```

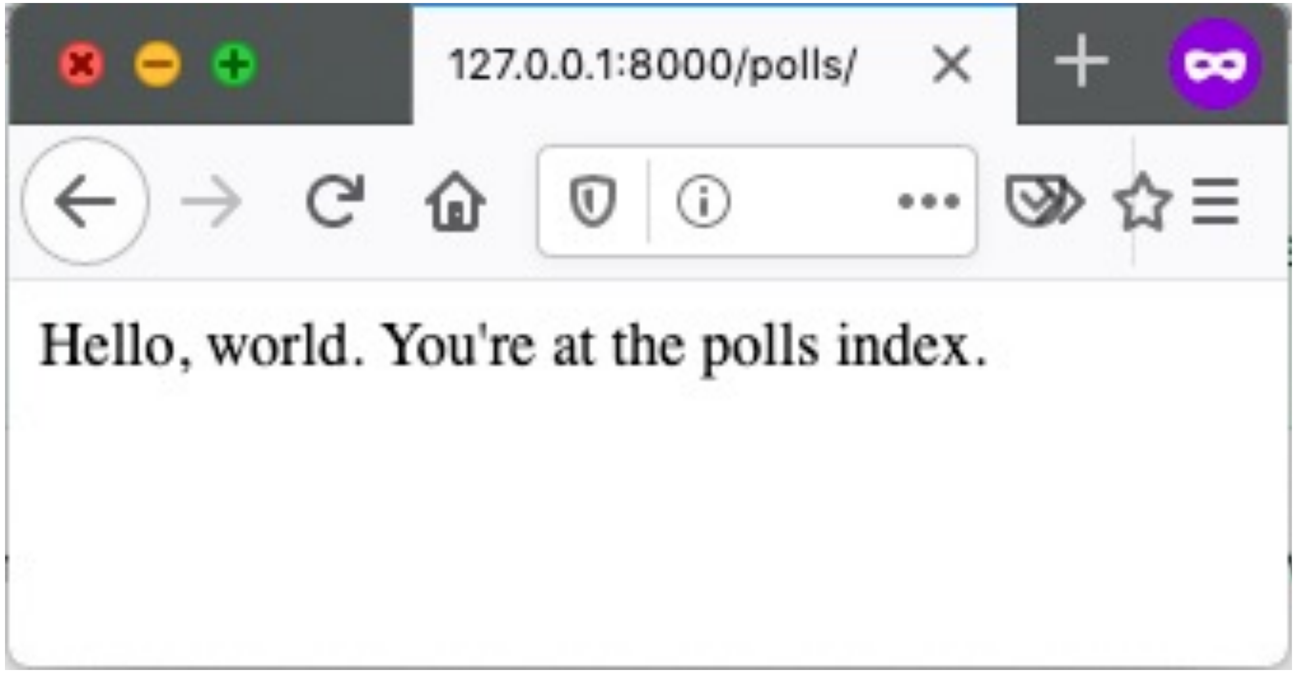
mysite/urls.py

```
from django.contrib import admin
from django.urls import include, path
urlpatterns = [
    path('polls/', include('polls.urls')),
    path('admin/', admin.site.urls),
]
```

Result

- Run the server

```
$ python manage.py runserver
```



Hello, world. You're at the polls index.

Python on Google Cloud

- To quickly deploy Python applications on Google Cloud, see:

  https://cloud.google.com/python

- Cloud Code, IDE Integration with IntelliJ, PyCharm and Visual Studio Code:

  https://cloud.google.com/code/

- Building a Python 3 App on App Engine using Flask:

  https://cloud.google.com/appengine/docs/standard/python3/building-app

  (app does not exceed free quotas)

- Quickstart: Build deploy a Python (Flask) web app to Cloud Run

  https://cloud.google.com/run/docs/quickstarts/build-and-deploy/deploy-python-service