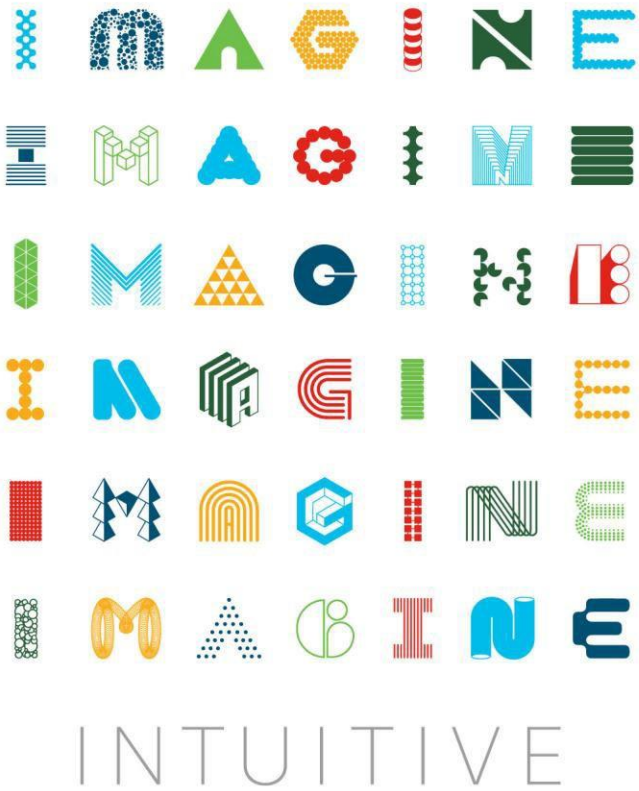
cisco

Coding 1002: Intro Python



Agenda

- Why Python?
- Basic Python Syntax
- Collections and Loops
- Script Structure and Execution

Why Python?

- Domain Applicability
  Established online DevOps Community
- Power and Flexibility
  Create & Work With: Shell Scripts, Back- end Web APIs, Databases, Machine Learning, …
- Platform Flexibility
  Run Your Code: Laptop, Server, VM, Container, Cloud, Cisco IOS Device

Python Scripts

- ✓ Text Files (UTF-8)
- ✓ May contain Unicode
  Some editors / terminals don’t support Unicode
- ✓ Use any Text Editor
  Using a Python- aware editor will make your life better
- ✓ No Need to Compile Them

PIP Installs Packages

- Included with Python v3+
  Coupled with a Python installation; may be called `pip3` outside a venv
- Uses the open PyPI Repository
  Python Package Index
- Installs packages and their dependencies
- You can post your packages to PyPI!

```
(venv) $ pip install requests
Collecting requests
 Downloading
<-- output omitted for brevity -->
Installing collected packages: idna,
certifi, chardet, urllib3, requests
Successfully installed certifi-2018.4.16
chardet-3.0.4 idna-2.6 requests-2.18.4
urllib3-1.22
(venv) $
```

Using your Python Interpreter

How to...	Command
Access the Python Interactive Shell	`$ python`
Running a Python script	`$ python script.py`
Running a script in ‘Interactive’ mode Execute the script and then remain in the Interactive Shell	`$ python -i script.py`

Python’s Interactive Shell

Accepts all valid Python statements

Use It To:

- ✓ Play with Python syntax
- ✓ Incrementally write Code
- ✓ Play with APIs and Data

To Exit:

`Ctrl + D` or `exit()`

```
(venv) $ python
Python 3.6.5 (default, Apr 2 2018, 15:31:03)[GCC 4.8.5 20150623 (Red Hat 4.8.5-16)] on
linuxType "help", "copyright", "credits" or "license" for more information.
>>>
```

Basic Python Syntax

Basic Data Types

Python type()	Values (examples)
`int`	`-128, 0, 42`
`float`	`-1.12, 0, 3.14159`
`bool`	`True, False`
`str`	“Hello 😎” Can use ‘ ’, “”, and “””””””””
`bytes`	`b”Hello \xf0\x9f\x98\x8e”`

- `>>> type(3)`
  <class ‘int’>
- `>>> type(1.4)`
- <class ‘float’>
- `>>> type(True)`
  <class ’bool’>
- `>>> type("Hello")`
  <class ’str’>
- `>>> type(b"Hello")`
- <class ‘bytes’>

Numerical Operators

Math Operations

Addition:	+
Subtraction:	-
Multiplication:	*
Division:	/
Floor Division:	//
Modulo:	%
Power:	**

```
>>> 5 + 2
7
>>> 9 * 12
108
>>> 13 / 4
3.25
>>> 13 // 4
3
>>> 13 % 4
1
>>> 2 ** 10
1024
```

Variables

Names

- Cannot start with a number [0-9]
- Cannot conflict with a language keyword
- Can contain: [A-Za-z0-9_-]
- Recommendations for naming (variables, classes, functions, etc.) can be found in PEP8

Created with the = assignment operator

Can see list of variables in the current scope with `dir()`

```
>>> b = 7
>>> c = 3
>>> a = b + c
>>> a
10

>>> string_one = "Foo"
>>> string_two = "Bar"
>>> new_string = string_one + string_two
>>> new_string
'FooBar'
```

In Python, Everything is an Object!

Use `.` *(dot)* syntax to access *“things”* inside an object.

```
>>> a = 57
>>> a.bit_length()
6
>>> "WhO wRoTe THIs?".lower()
'who wrote this?'
```

Terminology

When contained inside an object, we call…

Variable → Attribute

Function → Method

Check an object’s type with `type(object)`

Look inside an object with `dir(object)`

Working with Strings

String Operations

- Concatenation: +
- Multiplication: *

Some Useful String Methods

- Composition: “{}”.format()
- Splitting: “”.split()
- Joining: “”.join()

```
>>> "One" + "Two"
'OneTwo'

>>> "Abc" * 3
'AbcAbcAbc'

>>> "Hi, my name is {}!".format("Chris")
'Hi, my name is Chris!'

>>> "a b c".split(" ")
['a', 'b', 'c']

>>> ",".join(['a', 'b', 'c'])
'a,b,c'
```

Basic I/O

Get Input with `input()`

- Pass it a prompt string
- It will return the user's input as a string
- You can convert the returned string to the data type you need int(), float(), etc.

Display Output with `print()`

- Can pass multiple values
- It will concatenate those values with separators in between (default = spaces)
- It will add (by default) a newline ('\n') to the end

```
>>> print('a', 'b', 'c')
a b c

>>> i = input("Enter a Number: ")
Enter a Number: 1
>>> int(i)
1
```

Conditionals

Syntax:

```
if expression1:
    statements…
elif expression2:
    statements…
else:
    statements…
```

- ✓ Indentation is important!
- ✓ 4 spaces indent recommended
- ✓ You can nest if statements

Comparison Operators:

Less than	`<`
Greater than	`>`
Less than or equal to	`<=`
Greater than or equal to	`>=`
Equal	`==`
Not Equal	`!=`
Contains element	`in`

Combine expressions with: `and`, `or`

Negate with: `not`

Conditionals | Examples

```
>>> b = 5
>>> if b < 0:
...     print("b is less than zero")
... elif b == 0:
...     print("b is exactly zero")
... elif b > 0:
...     print("b is greater than zero")
... else:
...     print("b is something else")
...
b is greater than zero
```

```
>>> words = "Foo Bar"
>>> if "Bar" in words:
...     print("words contains 'Bar'")
... elif "Foo" in words:
...     print("words contains 'Foo'")
...
words contains 'Bar'
```

Functions | Don’t Repeat Yourself

Modularise your code

- Defining your own Functions
- (optionally) Receive arguments
- (optionally) Return a value

Syntax:

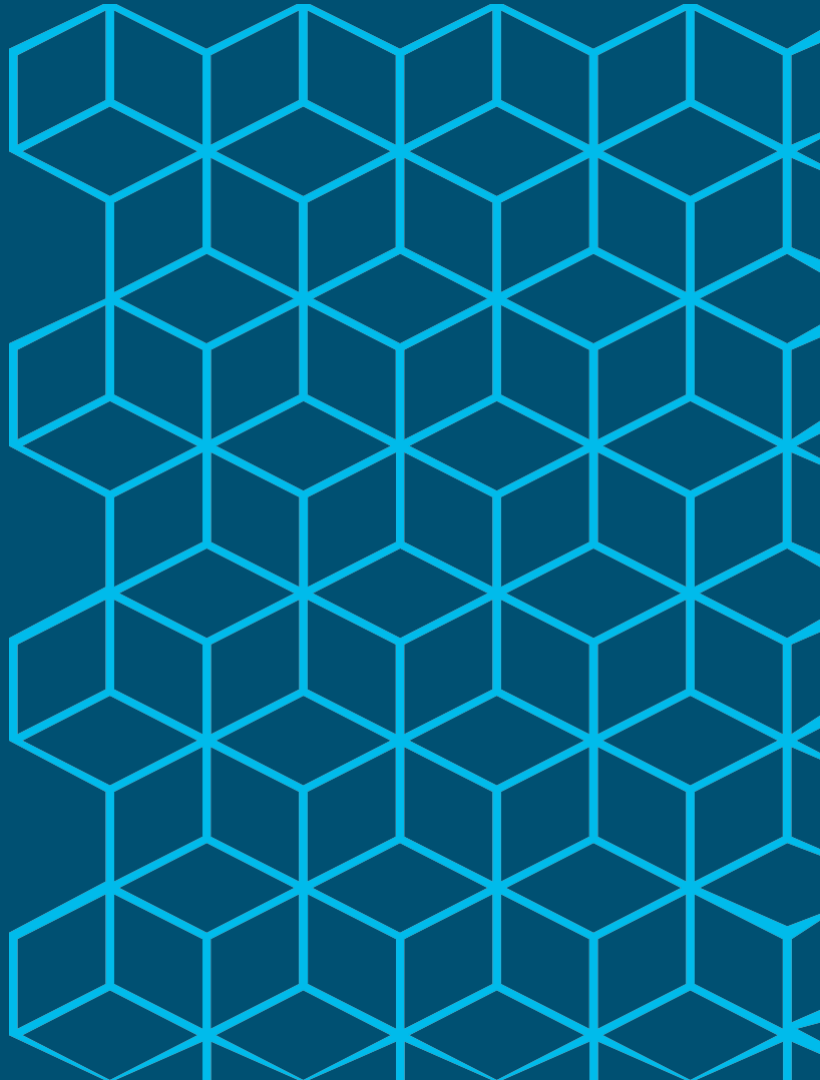
```
def function_name(arg_names):
    statements…
    return value
...
function_name(arg_values)
```

```
>>> def add(num1, num2):
...     result = num1 + num2
...     return result
...
>>>
>>> add(3, 5)
8



>>> def say_hello():
...     print("Hello!")
>>>
>>> say_hello()
Hello!
```

Python Collections and Loops



Data Structures / Collection Data Types

Name `type()`	Notes	Example
`list`	• Ordered list of items • Items can be different data types • Can contain duplicate items • Mutable (can be changed after created)	`['a', 1, 18.2]`
`tuple`	• Just like a list; except: • Immutable (cannot be changed)	`('a', 1, 18.2)`

dictionary
`dict`
	• Unordered key-value pairs
	• Keys are unique; must be immutable
	• Keys don't have to be the same data type
	• Values may be any data type
		`{"apples": 5,`
		`"pears": 2,`
		`"oranges": 9}`

Working with Collections

Name `type()`	Creating	Accessing Indexing	Updating

`list`	`l = [‘a’, 1, 18.2]`	`>>> l[2]`
		`18.2`
			`>>> l[2] = 20.4`
			`>>> l`
			`[‘a’, 1, 20.4]`

`tuple`	`t = (‘a’, 1, 18.2)`	`>>> t[0]` `‘a’`	You cannot update tuples after they have been created.

`dict`	`d = {“apples”: 5,`
	`“pears”: 2,`
	`“oranges”: 9}`
		`>>> d[“pears”]`
		`2`
			`>>> d[“pears”] = 6`
			`>>> d`
			`{“apples”: 5, “pears”: 6, “oranges”: 9}`

Dictionary Methods

Some useful dictionary methods:

- `{}.items()`
- `{}.keys()`
- `{}.values()`

There are many more! 😎

```
>>> d = {"a": 1, "b": 2, "c": 3}

>>> d.items()
dict_items([('a',1), ('b',2), ('c',3)])

>>> d.keys()
dict_keys(['a', 'b', 'c'])

>>> d.values()
dict_values([1, 2, 3])
```

Loops

Iterative Loops

```
for individual_item in iterator:
    statements…
```

```
>>> names = ["chris", "iftach", "jay"]
>>> for name in names:
...     print(name)
...
chris
iftach
jay
```

Conditional Loops

```
while logical_expression:
    statements…
```

```
>>> i = 0
>>> while True:
...     print(i)
...     i += 1
...
0
1
2
3
4
```

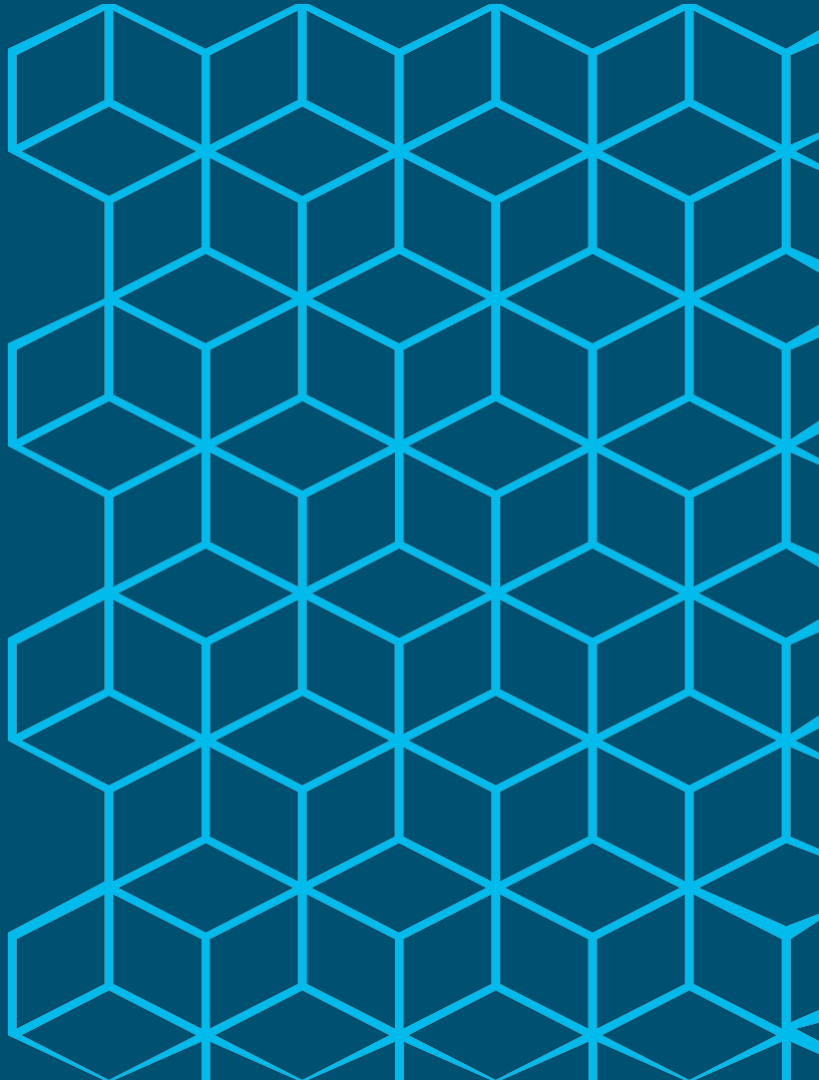
Iterating through a Dictionary

- Use the dictionary `.items()` method, which returns a “list of tuples”
- Unpack each tuple into variable names of your choosing to use within your block of statements

Method returns dictionary items as a list of **(key, value) tuples**, which the **for** loop will iteratively **unpack** into your variable names.

```
>>> for fruit, quantity in fruit.items():
...     print("You have {} {}.".format(quantity,  fruit))
...
You have 5 apples.
You have 2 pears.
You have 9 oranges.
```

Python Script Structure and Execution



Importing and Using Packages & Modules

Import “other people’s” code into your script.

Syntax:

```
import module
from module import thing
```

Tons of Packages:

- Python Standard Library
- Python Package Index
- GitHub

```
>>> import requests
>>> requests.get('https://google.com')
<Response [200]>


>>> response = requests.get('https://google.com')
>>> response.status_code
200
```

Debugging Basics

- Add `print()` statements
  Comment and uncomment them to “enable and disable your debugging”
- Understand how to read a Python Stake Trace
  1. Last Line First
  2. Top to Bottom
  3. **Stop when you reach someone else’s code**
- Run a script and then stay in the Python Interactive Shell
  Use the python –i option

Reading a Python Stack Trace

fortune_cookie.py → main() → create_fortune_cookie_message()

```
$ python intro-python/part2/fortune_cookie.py
Get your fortune cookie!
How many lucky numbers would you like? 5
Traceback (most recent call last):
  File "intro-python/part2/fortune_cookie.py", line 56, in <module>
    main()
  File "intro-python/part2/fortune_cookie.py", line 50, in main
    fortune_cookie_message = create_fortune_cookie_message(qty_lucky_numbers)
  File "intro-python/part2/fortune_cookie.py", line 38, in
create_fortune_cookie_message
    raise NotImplementedError()
NotImplementedError
```

cisco

Thank you