

# Cisco Webex Teams

## Questions?

Use Cisco Webex Teams to chat with the speaker during and/or after the session

## How

- 1 Find this session in the Cisco Live Mobile App
- 2 Click “Join the Discussion”
- 3 Follow the instructions on the screen

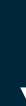
Webex Teams will be moderated by the speaker until November 15, 2019.



Download Webex Teams



[webex.com/download](https://webex.com/download)



[cs.co/track-6](https://cs.co/track-6)



Arjan Toxopeus  
Systems Architect DevNet



# Programming Fundamentals

Arjan Toxopeus  
Systems Architect DevNet  
11 November 2019

# <https://learninglabs.cisco.com/tracks/dnav3-track>

## Programming Fundamentals

Don't know Python? We got you covered. We'll cover all the essentials you need to get started, work through the labs and complete the Missions! From intro an intro to Git to parsing JSON with Python, you'll be coding in no time.

🕒 2 Hours 15 Minutes



### 💡 Intro to Coding and APIs **In Progress**

I design and manage networks of all sizes. I use IEEE 802.1w, IPv4, IPv6, OSPF, and BGP to build communications networks that would make Bob Kahn and Vint Cerf proud. Why should I learn to code?

### 💡 A Brief Introduction to Git **Completed**

Clone, branch, commit... We aren't talking about your family tree. Learn how to use git to download, edit and revise source code!

### 💡 Intro to Python - Part 1 **In Progress**

Basic data types, variables, conditionals and functions - we'll teach you the building blocks on which all great apps are built.

### 💡 Intro to Python - Part 2

Python is an awesome "batteries included" programming language. Learn about a few of Python's powerful built-in container data types and how to use loops to get your computer to do repetitious work for you.

### 💡 Parsing JSON with Python **Completed**

Use this basic template to write educational content for DevNet Express learning labs.



Continue Module

# A Brief Introduction to Git

# The Need for Version Control

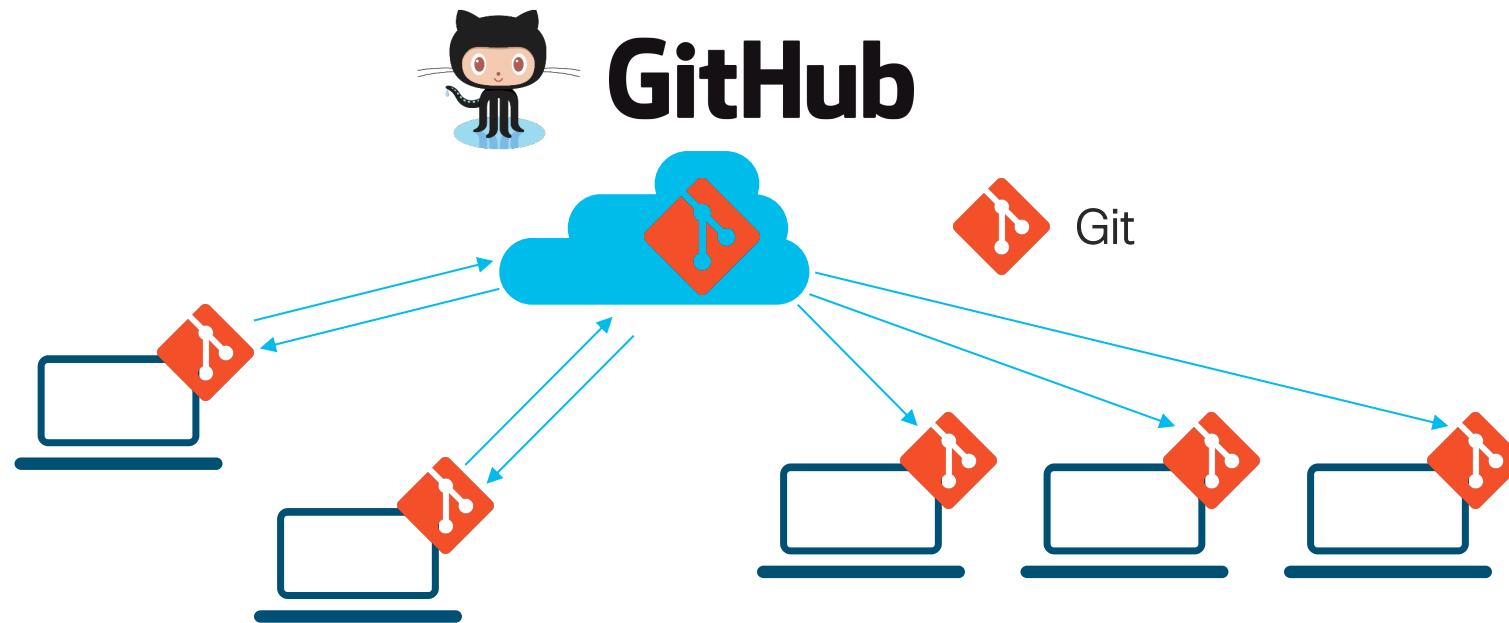
How do I make incremental changes and share my work with others?

How do I go back to the version of this file from (yesterday, last week, last year, ...)?

What changed between version X and version Y of a file?

People have been making changes to the same file (or set of files)...  
How do I reconcile and merge all these changes?

# Git vs. GitHub



# Basic Git Terminology

- **Repository (Repo)** – A vault for storing version controlled files
- **Working Directory** – The visible directory and its contents
- **Versioned Files** – Files you have asked Git to track
- **Un-Versioned Files** – Files in your working directory not tracked by Git
- **Commit** – Snapshot in time (of your version controlled files)
- **Branches** – A safe place for you to work



# A Peak Under the Hood

- Commits contain Trees
- Trees contain links to Files
- Git stores *full copies of all Changed Files*

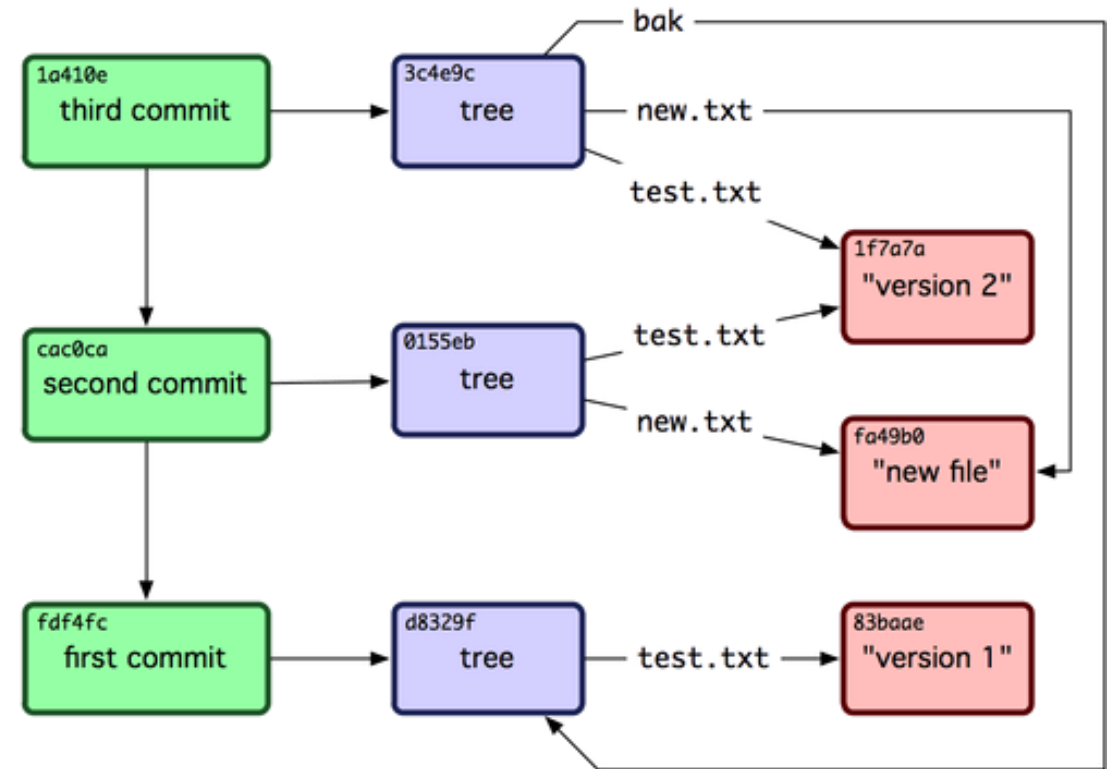


Image Source: <http://git-scm.com>

# What All New Git Users Do



Image Source: [xkcd.com](https://xkcd.com)

# Useful Git Commands

## Setup

Tell git who you are  
*one-time setup*

```
git config --global user.name "your name"  
git config --global user.email your@email.com
```

## Clone

Clone ("download") a git repository

```
git clone url
```

## Status

Check the Status of your local repository

```
git status
```

## Checkout

A Branch

Create and Checkout a local **Branch**  
Creates a "safe place" for your changes

```
git checkout -b new-branch-name
```

## Add

Add a file to your next commit.

```
git add filename
```

## Commit

Commit your changes.

```
git commit -m "Your commit message."
```

## Checkout

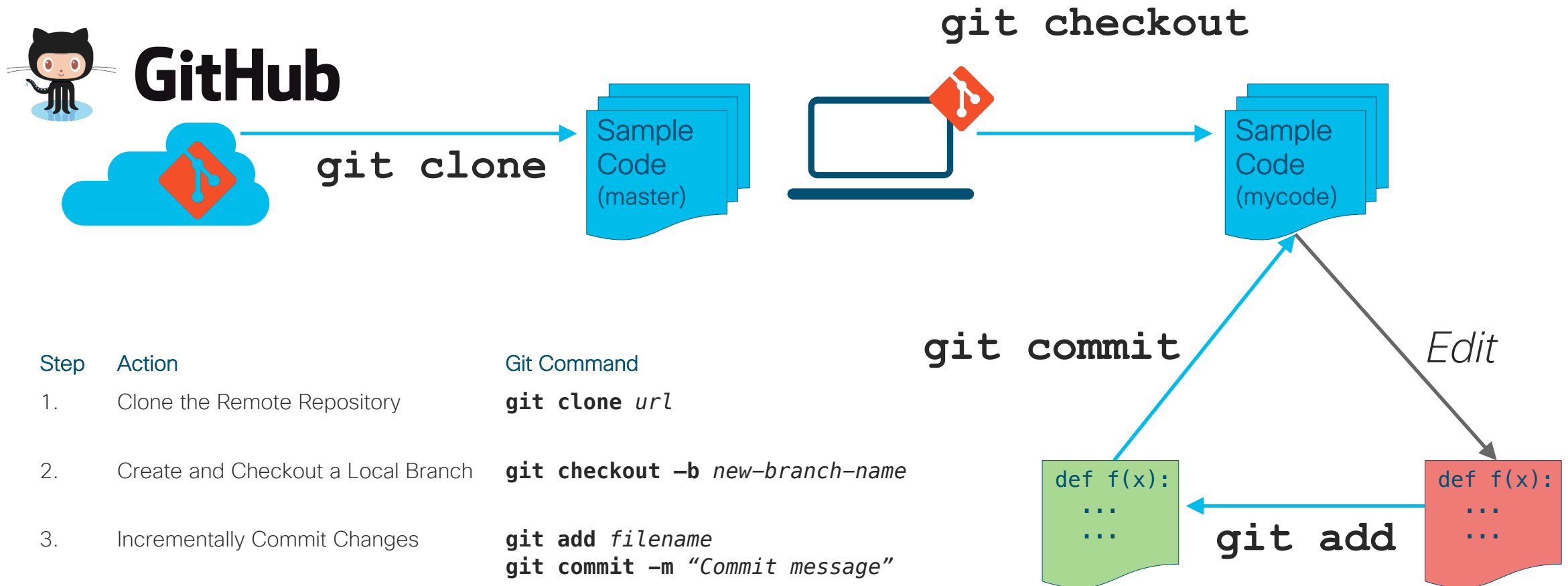
A File

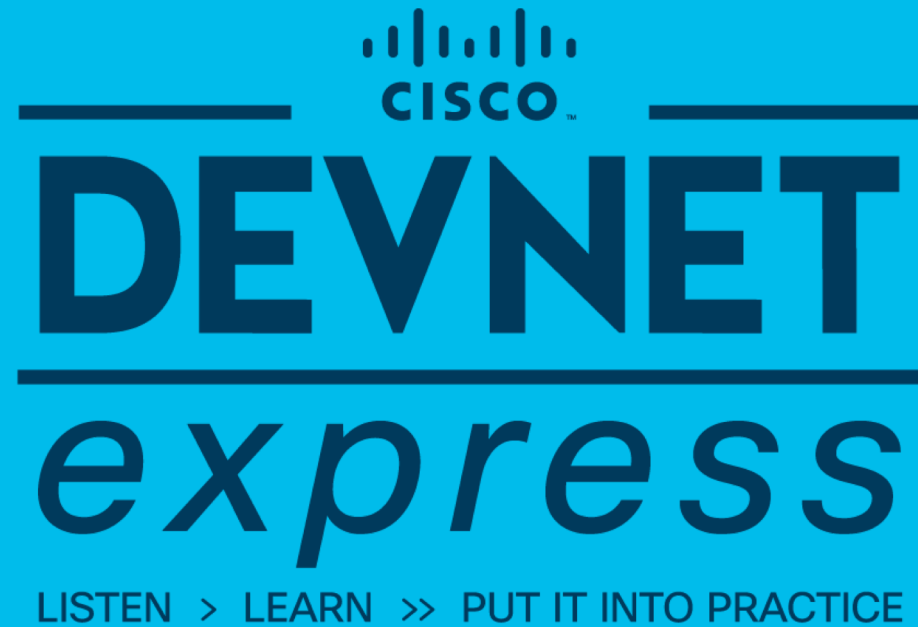
Checks-out a file from the last commit.  
Reverts any changes you have made, and restores  
the last committed version of a file.

```
git checkout filename
```

Learn More: **git --help** and **man git**

# DevNet Sample-Code Workflow





**@CiscoDevNet | #DevNetExpress**

# Intro to Python | Part 1

# 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

Using a Python Interpreter



# Know Thy Interpreter

What interpreter are you using?

- ❑ python
- ❑ python2
- ❑ python3
- ❑ python3.5
- ❑ python3.6
- ❑ *other*

What version is it?

\$ **python -V**

Where is it?

\$ **where** *command*

# What is a Virtual Environment?

- Directory Structure
- Usually associated with a Project
- An *isolated* environment for installing and working with Python Packages

```
$ python3 -m venv venv
$
$ tree -L 1 venv/
venv/
├── bin
├── include
├── lib
└── pyvenv.cfg
$
$ source venv/bin/activate
(venv) $
```

Remember

# Activating a Python Virtual Environment

**`source environment-name/bin/activate`**

- ✓ The activation script will modify your prompt.
- ✓ Inside a virtual environment your interpreter will always be **`python`**.

```
$ source venv/bin/activate
(venv) $
(venv) $
(venv) $ deactivate
$
```

# 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	\$ <b>python</b>
Running a Python script	\$ <b>python</b> <i>script.py</i>
Running a script in 'Interactive' mode Execute the script and then remain in the Interactive Shell	\$ <b>python -i</b> <i>script.py</i>

# 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)
<b>int</b>	-128, 0, 42
<b>float</b>	-1.12, 0, 3.14159
<b>bool</b>	True, False
<b>str</b>	"Hello 😎" Can use <code>' '</code> , <code>" "</code> , and <code>""" """</code>
<b>bytes</b>	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.

## 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)`

```
>>> a = 57
>>> a.bit_length()
6
>>> "Wh0 wRoTe THIs?".lower()
'who wrote this?'
```

# 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	<b>&lt;</b>
Greater than	<b>&gt;</b>
Less than or equal to	<b>&lt;=</b>
Greater than or equal to	<b>&gt;=</b>
Equal	<b>==</b>
Not Equal	<b>!=</b>
Contains element	<b>in</b>

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

## Modularize 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!
```



# Intro to Python | Part 2

# Python Collections & Loops

# Data Structures / Collection Data Types

Name <b>type()</b>	Notes	Example
<b>list</b>	<ul style="list-style-type: none"><li>• Ordered list of items</li><li>• Items can be different data types</li><li>• Can contain duplicate items</li><li>• Mutable (can be changed after created)</li></ul>	<code>['a', 1, 18.2]</code>
<b>tuple</b>	<ul style="list-style-type: none"><li>• Just like a list; except:</li><li>• Immutable (cannot be changed)</li></ul>	<code>('a', 1, 18.2)</code>
dictionary <b>dict</b>	<ul style="list-style-type: none"><li>• Unordered key-value pairs</li><li>• Keys are unique; must be immutable</li><li>• Keys don't have to be the same data type</li><li>• Values may be any data type</li></ul>	<code>{"apples": 5, "pears": 2, "oranges": 9}</code>

# Working with Collections

Name type()	Creating	Accessing Indexing	Updating
<b>list</b>	<code>l = ['a', 1, 18.2]</code>	<pre>&gt;&gt;&gt; l[2] 18.2</pre>	<pre>&gt;&gt;&gt; l[2] = 20.4 &gt;&gt;&gt; l ['a', 1, 20.4]</pre>
<b>tuple</b>	<code>t = ('a', 1, 18.2)</code>	<pre>&gt;&gt;&gt; t[0] 'a'</pre>	You cannot update tuples after they have been created.
<b>dict</b>	<code>d = {"apples": 5, "pears": 2, "oranges": 9}</code>	<pre>&gt;&gt;&gt; d["pears"] 2</pre>	<pre>&gt;&gt;&gt; d["pears"] = 6 &gt;&gt;&gt; d {"apples": 5, "pears": 6, "oranges": 9}</pre>

# 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])
```

# Other “Batteries Included” Collections

Collection	Description
<a href="#"><u>namedtuple()</u></a>	factory function for creating tuple subclasses with named fields
<a href="#"><u>deque</u></a>	list-like container with fast appends and pops on either end
<a href="#"><u>ChainMap</u></a>	dict-like class for creating a single view of multiple mappings
<a href="#"><u>Counter</u></a>	dict subclass for counting hashable objects
<a href="#"><u>OrderedDict</u></a>	dict subclass that remembers the order entries were added
<a href="#"><u>defaultdict</u></a>	dict subclass that calls a factory function to supply missing values
<a href="#"><u>UserDict</u></a>	wrapper around dictionary objects for easier dict subclassing
<a href="#"><u>UserList</u></a>	wrapper around list objects for easier list subclassing
<a href="#"><u>UserString</u></a>	wrapper around string objects for easier string subclassing

Learn More @  
[docs.python.org](https://docs.python.org)

# OrderedDict Collection

```
>>> from collections import OrderedDict
>>> od = OrderedDict()
>>> od["apples"] = 5
>>> od["pears"] = 2
>>> od["oranges"] = 9
>>>
>>> od["pears"]
2
>>> od["bananas"] = 12
>>> od
OrderedDict([('apples', 5), ('pears', 2), ('oranges', 9), ('bananas', 12)])
```

# 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 i < 5:
...     print(i)
...     i += 1
...
0
1
2
3
4
```



# Unpacking

Q: What if you wanted to break out a collection to separate variables?


A: *Unpack them!*

```
>>> a, b, c = [1, 2, 3]
>>> a
1
>>> b
2
>>> c
3
```

# 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
```

# Variable Scope

Open in Your Editor:

`intro-python/part2/variable_scope.py`

Code Review:

- Module-scoped “Global” Variables
- Argument Variables
- Local Variables

```
#!/usr/bin/env python
"""Demonstrate module vs. locally scoped variables."""

# Create a module variable
module_variable = "I am a module variable."

# Define a function that expects to receive a value for an argument variable
def my_function(argument_variable):
    """Showing how module, argument, and local variables are used."""
    # Create a local variable
    local_variable="I am a local variable."

    print(module_variable, "...and I can be accessed inside a function.")
    print(argument_variable, "...and I can be passed to a function.")
    print(local_variable, "...and I can ONLY be accessed inside a function.")

# Call the function; supplying the value for the argument variable
my_function(argument_variable="I am a argument variable.")

# Let's try accessing that local variable here at module scope
print("\nTrying to access local_variable outside of its function...")
try:
    print(local_variable)
except NameError as error:
    print(error)
```

# Python Script Structure and Execution

Open in Your Editor:

`intro-python/part3/structure.py`

Code Review:

- Structure
- Flow
- Execution

```
#!/usr/bin/env python
# """Module docstring."""

# Imports
import os
import sys

# Module Constants
START_MESSAGE = "CLI Inspection Script"

# Module "Global" Variables
location = os.path.abspath(__file__)

# Module Functions and Classes
def main(*args):
    """ My main script function.

    Displays the full path to this script, and a list of the arguments passed
    to the script.
    """
    print(START_MESSAGE)
    print("Script Location:", location)
    print("Arguments Passed:", args)

# Check to see if this file is the "__main__" script being executed
if __name__ == '__main__':
    _, *script_args = sys.argv
    main(*script_args)
```

# Debugging Basics

- Add **print()** statements  
Comment and uncomment them to “enable and disable your debugging”
- Understand how to read a Python Stack 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
```



# Parsing JSON with Python

# What is JSON?

- Standardized format for passing data as text.
- JavaScript Object Notation
- Looks strikingly similar to Python's syntax for dictionaries, lists, strings and number types!
- ...BUT... **JSON is just text!**

# JSON Syntax vs. Python Syntax

```
{
  "ietf-interfaces:interface": {
    "name": "GigabitEthernet2",
    "description": "Wide Area Network",
    "enabled": true,
    "ietf-ip:ipv4": {
      "address": [
        {
          "ip": "172.16.0.2",
          "netmask": "255.255.255.0"
        }
      ]
    }
  }
}
```

JSON

```
{
  'ietf-interfaces:interface': {
    'name': 'GigabitEthernet2',
    'description': 'Wide Area Network',
    'enabled': True,
    'ietf-ip:ipv4': {
      'address': [
        {
          'ip': '172.16.0.2',
          'netmask': '255.255.255.0',
        },
      ],
    },
  },
}
```

Python

# Reading-from and Writing-to Files

Use the python **open()** function.

**open(*file\_path*, mode='r')**

File Methods:

**.read()**

**.write()**

**.close()**

```
>>> file = open("demo.txt")
>>> contents = file.read()
>>> print(contents)
It's easy to work with files in Python!

>>> file.close()
```

```
>>> with open("demo.txt") as file:
...     print(file.read())
...
It's easy to work with files in Python!
```

Use the **with** statement if you don't want to have to remember to close the file after you are done working with a file.

# Parsing JSON

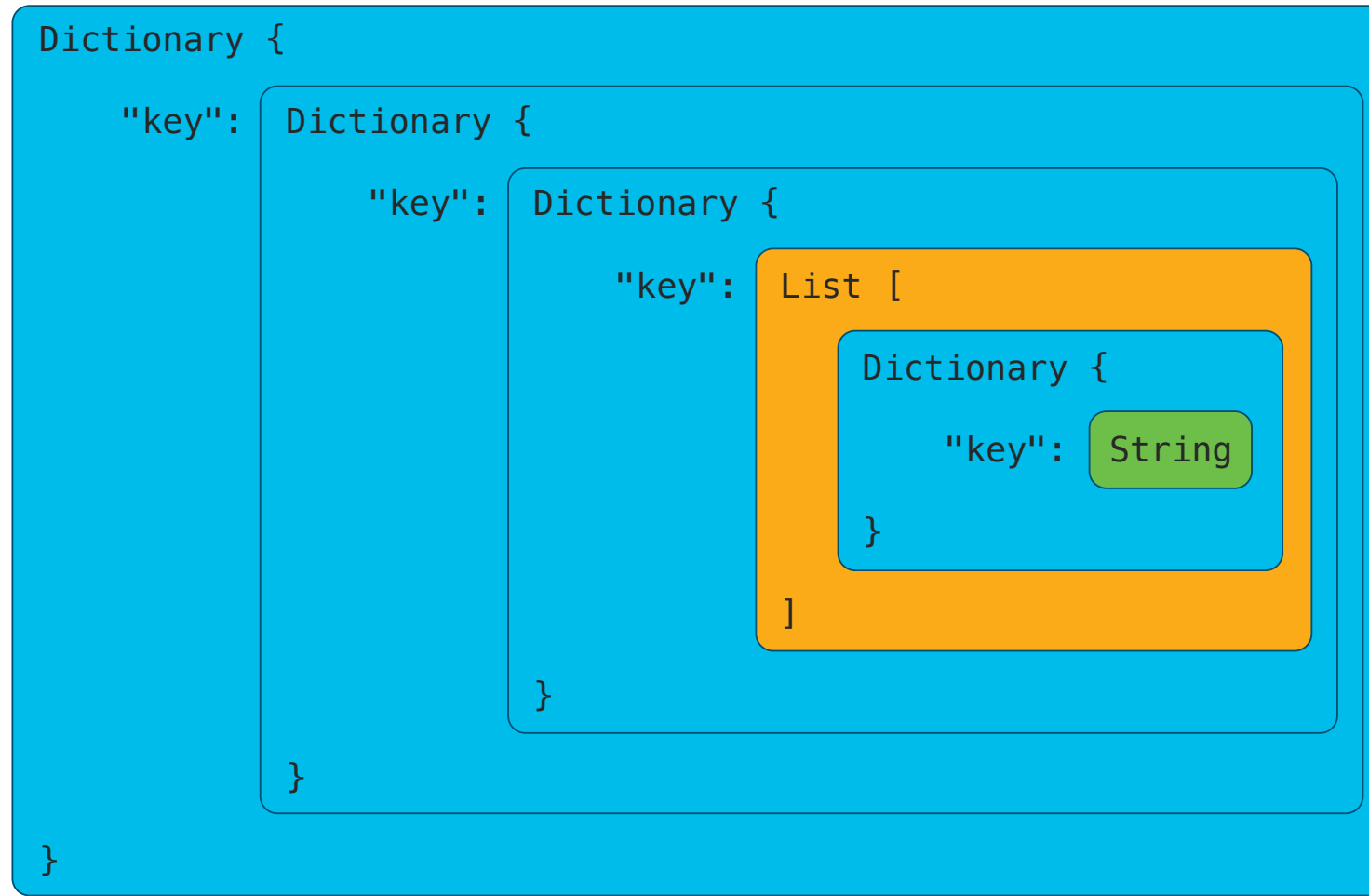
**Parsing:** Converting the text-based JSON data to native Python data types – things you can work with!

Python provides a native JSON parser for you!

```
import json  
data    = json.load(file)  
data    = json.loads(string)  
string = json.dump(file)  
string = json.dumps(data)
```

```
>>> string = '{"pets": ["cat", "dog"]}'  
>>> type(string)  
<class 'str'>  
  
>>> import json  
>>> data = json.loads(string)  
  
>>> type(data)  
<class 'dict'>  
  
>>> data["pets"][1]  
'dog'
```

# Nested Data



# Accessing Nested Data

## Indexing into Nested Data

1. Start with the outermost data structure.
  2. “Extract” from it what we want.
  3. Repeat.
- Play with data in the Python Interactive Shell
  - Takes practice.

How would you access the “ip” address in this example?

```
json_data = {
    'ietf-interfaces:interface': {
        'name': 'GigabitEthernet2',
        'description': 'Wide Area Network',
        'enabled': True,
        'ietf-ip:ipv4': {
            'address': [
                {
                    'ip': '172.16.0.2',
                    'netmask': '255.255.255.0',
                },
            ],
        },
    },
}
```

# Solution

```
>>> json_data["ietf-interfaces:interface"]["ietf-ip:ipv4"]["address"][0]["ip"]  
'172.16.0.2'
```



Wrap Up

# What you learned in this module...

- How to clone a git repo, create branches, and make commits.
- Core Python syntax, operators, conditionals, and functions.
- Python container data types and syntax for Python's loops.
- Python script structure, execution, and variable scoping and passing.
- How to parse JSON (aka. convert to native Python data types) and extract and reference it's data.

# ...where to go to continue learning:

## DevNet (*always!*)

<https://developer.cisco.com>

## Git

[git-scm.com](https://git-scm.com) Tutorials

## GitHub

[GitHub Guides](#)

## Python

[edx.org](https://edx.org) Python Courses

[coursera.com](https://coursera.com) Python Courses

[codecademy.com](https://codecademy.com) Learn Python

[Need a challenge?](#)

# Connect with Us and share Your Story...



@CiscoDevNet



#DevNetExpress

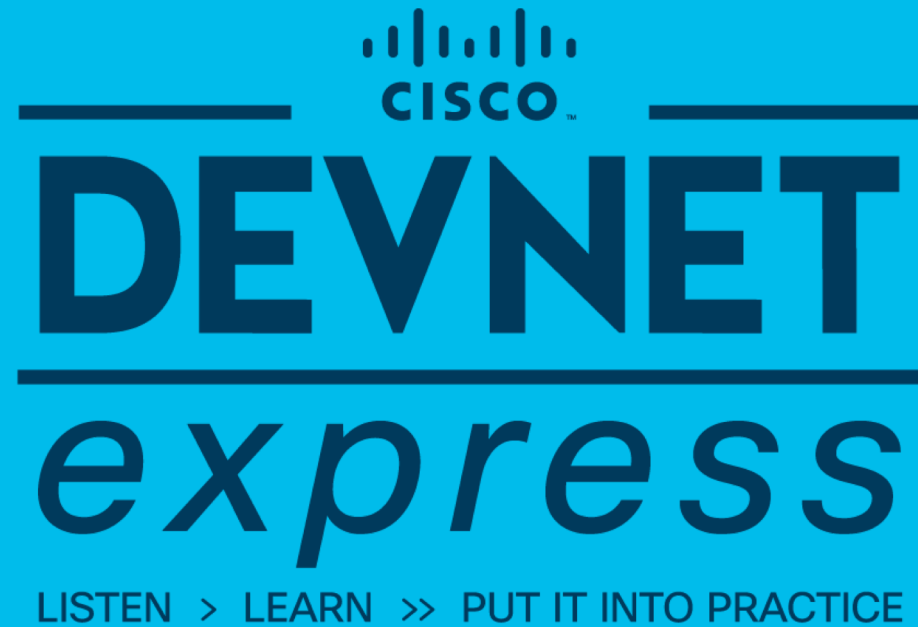


facebook.com/ciscodevnet/



<http://github.com/CiscoDevNet>

[developer.cisco.com](http://developer.cisco.com)



**@CiscoDevNet | #DevNetExpress**