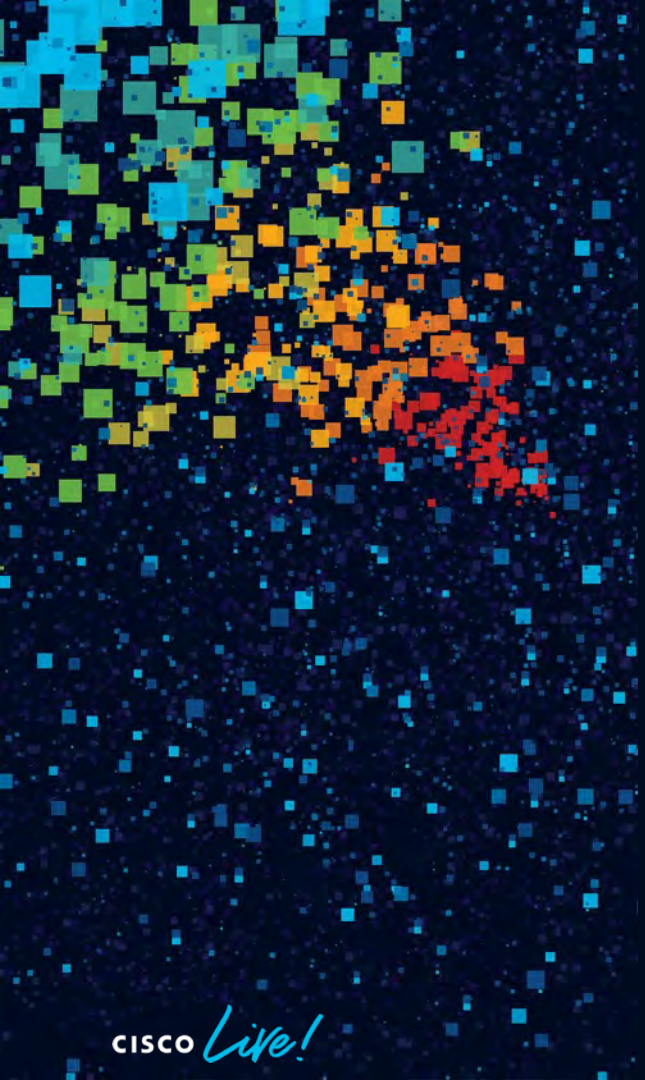
Coding 1002: Getting Started with Python

Matt DeNapoli – DevNet Developer Advocate
@theDeNap



Agenda

- Why Python
- Using the Python Interpreter
- Python Basics
- Python Conditionals and Loops
- Python Scripts and Execution
- Conclusion

Start Now Challenge – http://cs.co/startnowchallenge

DEVNET

GET
POST
PUT
DELETE

{REST}

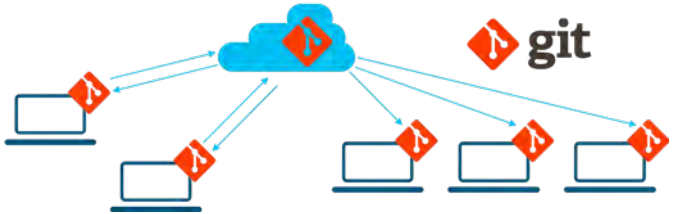
```
def generate_decks(num_decks):
    try:
        decks = requests.get(f"https://deckofcardsapi.com/api/deck/new/shuffle/?deck_count={str(num_decks)}")
        decks.raise_for_status()
        print(type(decks.text))
        print(decks.text)
        decks = decks.json()
        print(type(decks))
        print(decks)
        print(f"Success! {str(num_decks)} created!  The deckId is {decks['deck_id']}")

        return decks


        print(f'YOUR SCORE: {str(your_score)}\n')

        print("Now my turn!\n")


    except Exception as e:
        sys.exit(e)
```



Why Python

Python is...

- An interpreted language
  Do not need to compile the code
- Easy to understand
  Code reads like English without unnecessary syntax
- Easy to code
  Do powerful things with less code
- Simple
  Scripts are UTF-8 text files that can be edited in any text editor

Python...

- Maintains two stable versions
  - Python 2.x (deprecated Jan 2020)
  - Python 3.x (not backwards compatible)
- Includes pip
  - Installs packages and their dependencies
- Has a Python Interactive Shell
  - Useful for testing out code or libraries
- Has virtual environments
  - An isolated environment for installing and working with Python Packages

Why Python?

- Power and Flexibility
  Create & Work With: Shell Scripts, Back-end Web APIs, Front-end UIs, Databases, Machine Learning, etc.
- Platform Flexibility
  Run Your Code: Laptop, Server, VM, Container, Cloud, Cisco IOS Device
- Domain Applicability
  Established online community with open source and code sharing

Using the Python Interpreter

Know The 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
```

Start Now Challenge Hint!!!

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) $
```

Activating a Python Virtual Environment

Remember

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	`$ 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.
>>>
```

Python Basics

Basic Data Types

Python type()	Values (examples)
int	-128, 0, 42
float	-1.12, 0, 3.14159
bool	True, False
str	“Hello World”
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’>
```

Strings

- Are any Unicode text (Python 3.x)
- Can use ' ', " ", ''' ''' and """ """
- Combined single and double quotes to:
  - include quotes in a string
  - use apostrophes in a string
- Use triple quotes for multiline strings

```
>>> single_quote = 'This is my string.'
>>> double_quote = "This is my string."
>>> nested_quote = 'He said, "I love
Python".'
>>> triple_quote = '''I want my string
to be on
separate lines.'''
```

Working with Strings

String Operations

Concatenation: **+**

Multiplication: *****

Some Useful String Methods

Composition: **“{}”.format()**

Splitting: **“”.split()**

Joining: **“”.join()**

Length: **len(“”)**

```
>>> "One" + "Two"
'OneTwo'
>>>
>>> "Abc" * 3
'AbcAbcAbc'
>>>
>>> "Hi, my name is {}!".format("Chris")
'Hi, my name is Chris!'
>>>
>>> "Python is cool".split(" ")
['Python', 'is', 'cool']
>>>
>>> ",".join(['Bob', 'Sue', 'Joe'])
'Bob,Sue,Joe'
>>>
>>> len("a b c")
5
```

Variables

Start Now Challenge
Hint!!!

- Reserved memory to store values
- Every variable is an object
- Created with the = assignment operator
- Are type agnostic and can change type throughout its lifetime
- Variable Names
  - Cannot start with a number [0-9]
  - Cannot conflict with a language keyword
  - Can contain: [A-Za-z0-9_-]

```
>>> b = 7
>>> c = 3
>>> a = b + c
>>> a
10
>>>
>>> b = "Foo"
>>> c = "Bar"
>>> a = b + c
>>> a
'FooBar'
```

Variable Scope

Type	Where are they defined?	What is the scope?	Notes
`global/ module variable`	• In the main body of the Python script, outside of any function or class • In a function when the `global` keyword is used	• Throughout the whole module/file • Any file that imports the Python file • In every function	• Should be avoided when unnecessary • Global variables that are in all caps are often used to represent constants
`local variable`	• In a function • In a class	• In the function or class where it was defined	• When in a function, if a global and local variable has the same name, the local variable will be used • Function arguments are local variables

Variable Scope Example

```
>>> myVariable = "This is a global variable"
>>>
>>> def myFunction():
...     myLocalVariable = "This is a local variable"
...     myVariable = "This is a local variable with the same name as the global variable"
...     print(myLocalVariable)
...     print(myVariable)
...
>>>
>>> myFunction()
This is a local variable
This is a local variable with the same name as the global variable
>>>
>>> myVariable
This is a global variable
>>>
>>> myLocalVariable
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'myLocalVariable' is not defined
```

In Python, Everything is an Object!

- Objects are an encapsulation of variables and functions into a single entity
- Use the **.** *(dot)* syntax to access methods inside an object.
- View methods with **dir(*obj*)**
- Terminology

  When contained inside an object, we call…

  Variable → Attribute

  Function → Method

```
>>> a = 57
>>> a.bit_length()
6
>>>
>>> b = 3.5
>>> b.is_integer()
False
>>>
>>> "WhO wRoTe THIs?".lower()
'who wrote this?'
>>>
>>> dir(a)
['__abs__', '__add__', '__and__', '__bool__',
'__ceil__', '__class__', '__delattr__',
'__dir__', '__divmod__', '__doc__', '__eq__',
'__float__', '__floor__', '__floordiv__',
```

Advanced Data Types (Collections)

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)`

`dict`
(dictionary)
	• 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	Useful methods

list	`l = ['a', 1, 18.2]`
	`l2 = [53, 1, 67]`
	`l3 = [[1, 2], ['a', 'b']]`
		`>>> l[2]`
		`18.2`
			`>>> l[2] = 20.4`
			`>>> l`
			`[‘a’, 1, 20.4]`
				`>>> concat_l = l2 + l3`
				`>>> concat_l`
				`[53, 1, 67, [1, 2], ['a', 'b’]]`
				
				`>>> l2.sort()`
				`>>> l2`
				`[1, 53, 67]`

tuple	`t = ('a', 1, 18.2)`
	`t2 = (1, 3, 4)`
		`>>> t[0]`
		`'a'`
			You cannot update tuples after they have been created.	`>>> concat_t = t + t2`
				`>>> concat_t`
				`('a', 1, 18.2, 1, 3, 4)`

dict	`d = {"apples": 5,`
	`"pears": 2,`
	`"oranges": 9}`
	`d2 = {1: 15,`
	`5: 'grapes’,`
	`9: [1, 2, 3]}`
		`>>> d["oranges"]`
		`9`
		
		`>>> d.get("pears")`
		`2`
			`>>> d["pears"] = 6`
			`>>> d`
			`{'apples': 5, 'pears':`
			`2, 'oranges': 9}`
				`>>> d.items()`
				`dict_items([('apples', 5), ('pears',`
				`2), ('oranges', 9)])`
				
				`>>> d.keys()`
				`dict_keys(['apples', 'pears',`
				`'oranges’])`
				
				`>>> d.values()`
				`dict_values([5, 2, 9])`

Python Conditionals and Loops and Functions

Conditional Expressions

Syntax:

```
    operand operator operand
Ex:    x        <        10
```

- ✓ All expressions have minimum of one operand
- ✓ Operands are the objects that are manipulated
  - Can be a variable
  - Can be a value
- ✓ Evaluated to a Boolean

Comparison Operators:

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

Logical Operators

Syntax:

```
    expr1 operator expr2
Ex: x<10    and     x>0
```

- ✓ Logical operators join together expressions
- ✓ Evaluated to a Boolean
- ✓ Expressions are evaluated from left to right
- ✓ Can combine multiple logical operators

Logical Operators:

Logical Operator	expr1	expr2	Result
or	True	False	True
	False	True	True
	True	True	True
	False	False	False
and	True	False	False
	False	True	False
	True	True	True
	False	False	False
not	------	True	False
	------	False	True

Conditionals

Syntax:

```
if expression1:
    statements1…
elif expression2:
    statements2…
else:
    statements3…
```

- ✓ Indentation is important!
- ✓ 4 spaces indent recommended
- ✓ You can nest if statements
- ✓ `elif` and `else` statements are optional

```
>>> b = 5
>>> if b < 0:
...     print("b is less than zero")
... elif b == 0:
...     print("b is exactly zero")
... elif b > 2 and b < 7:
...     print("b is between three and six")
... else:
...     print("b is something else")
...
b is between three and six
```

Loops | For

Syntax:

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

- ✓ Loops are used when you want to do something many times.
- ✓ Iterate over a sequence or collection
- ✓ *Iterator* can be a list, tuple, dictionary, set, string, etc.

```
>>> names = ["Chris", "Dave", "Jay"]
>>> for name in names:
...     print(name)
...
Chris
Dave
Jay
```

Loops | While

Syntax:

```
while logical_expression:
    statements…
```

- ✓ Executes until the `logical_expression` is false
- ✓ Watch out for infinite loops
- ✓ Use `break` to exit the loop
- ✓ Use `continue` to skip the current block

```
>>> i = 0
>>> while i < 4:
...     print(i)
...     i += 1
...
0
1
2
3
>>>
>>> while True:
...     print("In an infinite loop.")
...
In an infinite loop.
In an infinite loop.
In an infinite loop.
In an infinite loop.
...
```

Functions | Don't Repeat Yourself

Syntax:

```
def func_name
(optional_args):
    statements…
    return value
```

- ✓ Modularize your code
  - Defining your own Functions
  - (optional) Receive arguments
  - (optional) Return a value
- ✓ `optional_args` are local variables

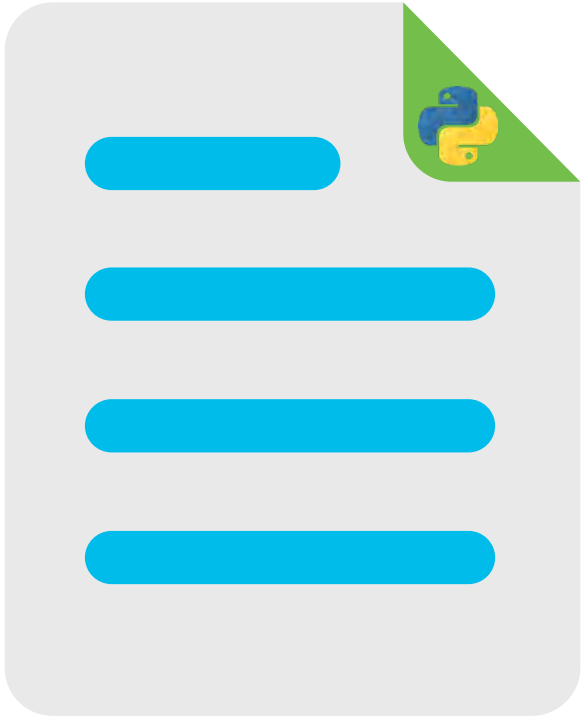
```
>>> def circumference(radius):
...     result = 2 * math.pi * radius
...     return result
...
>>> circumference(2)
12.566370614359172
>>>
>>> def say_hello():
...     print("Hello!")
...
>>> say_hello()
Hello!
```

Python Scripts and Execution

Python Scripts

- ✓ File extension: *.py
- ✓ Text files (UTF-8)
- ✓ Use any text editor

  Using a Python-aware editor will make your life easier
- ✓ No need to compile
- ✓ Executed with a Python Interpreter



Importing and Using Packages & Modules

Syntax:

```
import module

from module import func/const
```

- ✓ Import code into your script
  - Another module you wrote
  - Modules provided by companies
  - From other developers
- ✓ Can import the whole module or a specific function or constant

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

Basic I/O

Get Input with `input()`

- Pass it a prompt string
- Returns the user's input as a string
- Need to convert the returned string to the correct data type

Display Output with `print()`

- Can pass multiple values
- It will concatenate those values with separators in between (default = spaces)
- Adds 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
```

Python Script Example

```
#!/usr/bin/env python

# Imports
import random

# Module Constants and Global/Module Variables
FORTUNES = [
    "There is a good chance your code will work, eventually.",
    "I see Network DevOps in your future."
]

# Module Functions and Classes
def generate_fortune() -> str:
    return random.choice(FORTUNES)

def main():
    print(generate_fortune())

# Check to see if this file is the "__main__" script being executed
if __name__ == '__main__':
    main()
```

Executing Python Scripts

Syntax:

```
python file_name.py
python3 file_name.py
```

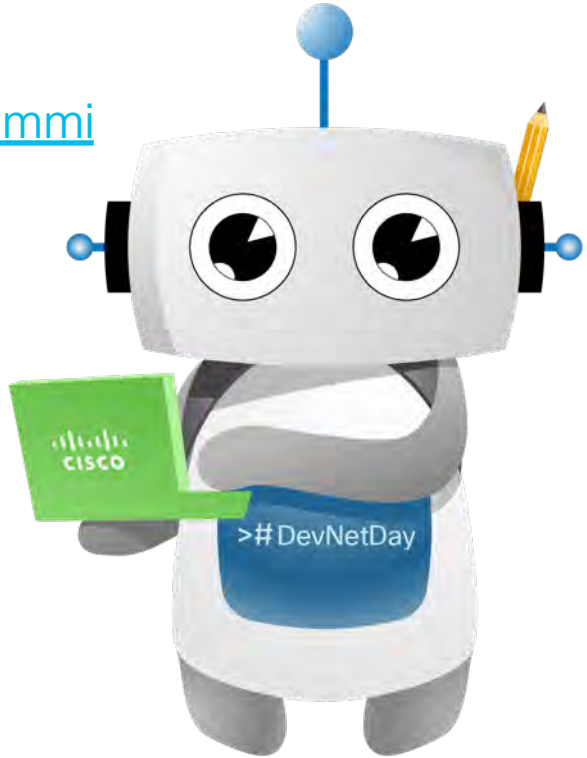
- ✓ Use the keyword `python` or **`python3`** to execute the script
- ✓ Scripts are executed from the Terminal
- ✓ The Python Interpreter starts from the first line and executes each statement in succession

```
(virtual_env)$ python fortune.py
I see Network DevOps in your future.
(virtual_env)$
(virtual_env)$ python fortune.py
I see Network DevOps in your future.
(virtual_env)$ deactivate
$
$ python3 fortune.py
$ There is a good chance your code will work,
eventually.
```

Explore More

- Programming Fundamentals: https://developer.cisco.com/learning/modules/programming-fundamentals

- Start Now Challenge: http://cs.co/startnowchallenge



Thank you

Possibilities

#CiscoLive | #DevNetDay