Cisco live!
February 20 - 24, 2017 · Berlin

cisco

Your Time Is Now

~~Monty~~
Python 101 for Network Engineers

Tom Taggart, Network Consulting Engineer

ttaggart@cisco.com

BRKSDN-1009

Cisco live!

The Trade Publications…

"…As software takes over the networking discipline, engineers who don't learn to code a general-purpose programming language will be left behind…"

Byers, K. (2014, August 11). *Programming: An Essential Skill For Network Engineers.* Retrieved from http://www.networkcomputing.com/.

What Does Wise Experience Tell Us?

“When you are finished changing, you're finished.”

Benjamin Franklin
1706-1790

The Research and Advisory Companies…

"… ISG Insights' new Automation Index™ indicates that IT service provider productivity is increasing sharply, leading to upwards of a 50 percent reduction in the number of resources required to support core IT operations services. …"

Jones, S. (2016, September 8). *Inaugural ISG Automation Index™ Shows IT Services Productivity Surging, Costs Plummeting* Retrieved from http://insights.isg-one.com/.

What Does Wise Experience Tell Us?

“…fortune favors the prepared mind.”

Louis Pasteur
1822-1895

Core Message & Goal Sheet

Your time is now, start today, in the planning and preparing of yourself to take a **leading role** in the network engineering evolution towards automation.

Teaching to the top

Top of the 50%

- **Have deep structural knowledge of concepts**
- Leverage this understanding to learn faster; **codes clearly**.
- Apply knowledge to solve new problems creatively
- **Has a vision, a timeline, and a plan to succeed**
- **Mentors**

Middle of the 50%

- **Have solid inflexible knowledge base**
- Realizing that memorization has its limits; **working to write clearer code.**
- Apply knowledge to solve redundant problems effectively
- **Working on a plan to succeed**

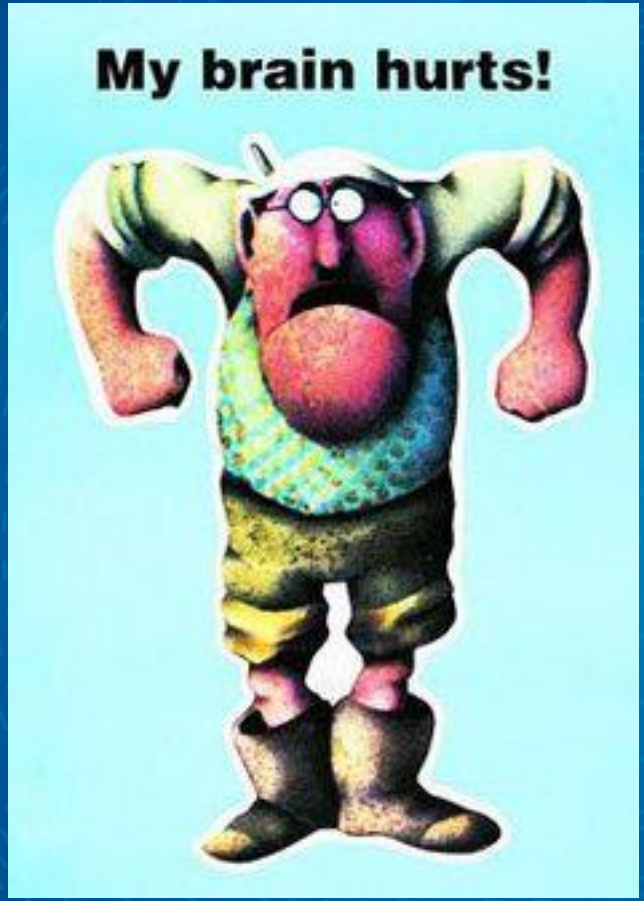
Bottom of the 50%

- Turning rote memorization into inflexible knowledge base
- Those with interest and aptitude just starting
- Learning use cases; writing unreadable code.
- Success plan yet to be developed
- Mentees

http://www.aft.org/periodical/american-educator/winter-2002/ask-cognitive-scientist

Agenda

- Housekeeping
- Foundational Ideas
- Core Object Data Types and Operations
- Procedural Statements and Syntax
- Transition to Functional Based Code
- Tips for Writing Good Code
- Documentation and Online Resources
- Conclusion
- Appendix: For Future Study



Housekeeping

Assumptions

- You want to learn Python and have a beginner or intermediate skill level
- You have Python installed on your laptop
- We'll use version 2.7 as our reference point
- We'll be creating a foundation for further study
  - This has been a long intense week
  - Relax, take in the information
- We'll show both contrived and production examples (in slide format)
  - Not all examples are cut-and-paste workable (snippets of code)
  - Might not be the most efficient or best approach (showing an idea)
- You are a passionately curious self-starter

Courtesies

- Please set mobile phones to vibrate or silent mode
- I am a little hard of hearing, so…
  - Please raise hand if you have an answer/question
  - If I ask you to repeat, rephrase, or talk slower please don't get flustered
- Please leverage the Spark message board for questions
  - Opened at start of this session
  - Will not monitor during this session but will start immediately after session
  - If a question is of larger scope, I'll have to bump into spark room discussion or schedule MTE time with you

Cisco Spark

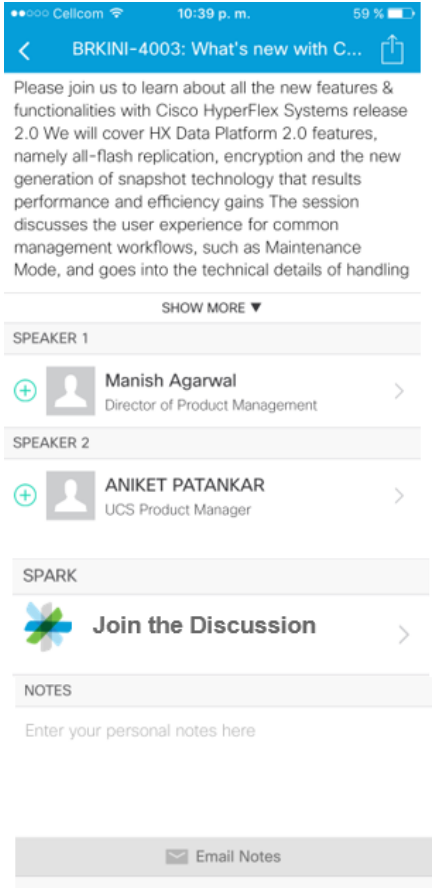
Ask Questions, Get Answers, Continue the Experience

Use Cisco Spark to communicate with the Speaker and fellow participants after the session

Download the Cisco Spark app from iTunes or Google Play

1. Go to the Cisco Live Berlin 2017 Mobile app
2. Find this session
3. Click the Spark button under Speakers in the session description
4. Enter the room, room name = BRKSDN-1009
5. Join the conversation!

The Spark Room will be open for 2 weeks after Cisco Live



Foundational Ideas

What is a scripting language?

"…Scripting languages are designed for "gluing" applications; they use typeless approaches to achieve a higher level of programming and more rapid application development…(this will be) more and more important for applications of the future."

Ousterhout, J.K. (1998, March). Scripting: Higher-Level Programming for the 21st Century. *IEEE COMPUTER*, 23-30.

Is Python a scripting language?

Python…

- Can be defined many ways depending on who you are and what you are using it for
- Can be considered a scripting language as network engineers often use it to “glue” systems together in the scripting sense.

Maybe better put,

- “…is a general-purpose programming language that blends procedural, functional, and object-oriented paradigms.”

Lutz, M. (2013). *Learning Python*. Sebastopol, CA: O’Reilly Media, Inc.

Python pros and cons…

Pros

- Free & Open source developed
- Portable
- Easy to use & learn
- Object-Oriented and functional
- Named after Monty Python

Cons

- Runs slower than lower level languages

Compared to other languages?

Your employer, job role, or customer may influence choice of language, but in general, Python is…

- More powerful than *TCL, Visual Basic,* and *JavaScript*
- More readable than *PERL*, *PHP*, and *Ruby*
- Easier to use than *Java, C, C++,* and *C#*
- More established than *GO*

Python program from an engineer's viewpoint

- A Python program is a text file with a series of Python statements in it, also called a module.
- The text file will end with the .py extension
  - E.g., *scp_files_to_server.py*
- Files can be created in a text processing application or an Integrated Development Environment, IDE
  - E.g., *notepad, vi, nano,* … or *IDLE, Komodo, PythonWin,* …
- The programs are then run through the Python interpreter, or program, installed on your machine.

What is an IDE?

- An **integrated development environment** (**IDE**) is a software application that provides comprehensive facilities to computer programmers for software development.

https://en.wikipedia.org/wiki/Integrated_development_environment

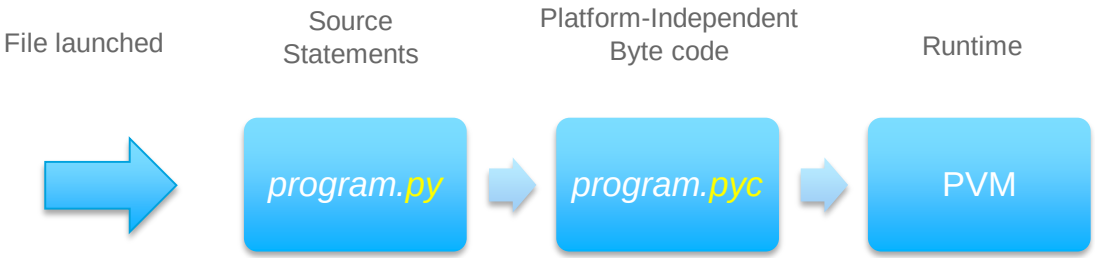
- How to pick one for yourself, reviews…

https://wiki.python.org/moin/IntegratedDevelopmentEnvironments

How do I launch a Python Program?

- From the command line or terminal (OS dependent)
  - `C:\code>` **myprogram.py**
  - `TTAGGART-M-M042:PythonScripts$` **./myprogram.py**
- Double click on icons
- From other programs
  - IDLE, IDE debug, Python script, Shell script, Applescript, Powershell, Mac Automator, Mac Launchd .plist files, Windows Task Scheduler, .bat files, …
- From the interactive shell
  - Used to both experiment and test program files on the fly

Python program from Interpreter's viewpoint



1. File is executed
2. Each statement is converted into a group of lower lever byte code instructions
   - Stored in *.pyc files
   - Done to speed up execution
   - Next time program is run, if no changes in source file, the .pyc file is run directly
3. Once file has been compiled into byte code it is handed off to Python Virtual Machine, PVM, for execution.
   - No make or build step, like in C
   - Byte code instructions iterated through at runtime
   - Byte code isn't binary machine code, it's a Python specific representation

Basic Python Installation

What's included

- Built-ins
  - Objects
    - Attributes
    - Methods
  - Functions
- Standard library
  - Already installed
  - Needs to be imported

- What's not included
  - 3rd party modules
    - Not installed with basic installation
    - Needs to be downloaded and installed

Core Object Data Types and Operations

Python Conceptual Hierarchy*

1. Programs are composed of modules (files with .py extension)
2. Modules contain statements
3. Statements contain expressions
4. Expressions create and process *objects*
   - Everything in Python is an Object
   - We're going to focus on the core built-in data objects

*(Lutz, 2013)

Core Built-in Data Objects

Type	Example
Numbers	`45.6, 7, Decimal(), Int()`
Strings	`'spam', u'Networking', 'Tom'`
Lists	`[[1, 2, 'three'], [1,2,3], 'spam', x]`
Dictionaries	`{'key1' : 1, 'key2' : {'Key3' : 'value3'}}`
Tuples	`((1, 2, 'three'), [1,2,3], 'spam', x)`
Files	`open('spreadsheet.xls', 'wb')`
Sets	`{'1', '2', '3'}, set('123')`
Other types	`Booleans, None`

Objects Ordered by Principle

	Immutable	Mutable
Atomic* Entity	Numbers	
Ordered Collection	Strings, Tuples	Lists
Unordered Collection	Sets	Dictionaries

Numbers

Social Media Moment?

Friendly competition with the other CiscoLives

I'm a lumberjack

and I'm OK

I sleep all night

and I work all day

He's a lumberjack

and he's OK

He sleeps all night

And he works all day

Four Number Types

Type	Description	Example
Integer, normal	Whole numbers with no fractional part; up to 32 bits precision.	5, Int('76'), -100, 666666666
Integer, Long*	Whole numbers with no fractional part; unlimited precision.	2**1000000, 65536L
Floating-point	Numbers with a fractional part expressed with either a decimal point or optional signed exponent.	3.14, 5.99e-10, 5E556, 3.410e+110
Complex	Numbers with a real and imaginary part; optional real part + imaginary part expressed with j.	6+4j, 9.35+2.0j, 5j

*- Python converts number to Long form when needed. No engineer interaction needed.

Built-in Functions and Standard Library Modules

Item	Description	Examples
Built-in Functions	Math based built-in functions	Abs(), bin(), bool(), float(), hex(), int(), len(), long(), max(), min(), oct(), ord(), pow(), round(), sum()
Standard Library Modules	Importable modules included in the standard library install.	math.py, random.py, fractions.py, decimal.py

Additional Number Types in Standard Library

Type	Description
Rational	Number with both a numerator and denominator.
Decimal	Floating-point number with user defined fixed precision.

Rational Examples

```
>>> from fractions import Fraction
>>> Fraction(16, -10)
Fraction(-8, 5)
>>> Fraction(123)
Fraction(123, 1)
>>> Fraction()
Fraction(0, 1)
>>> Fraction('3/7')
Fraction(3, 7)
>>> Fraction('-.125')
Fraction(-1, 8)
```

Decimal Examples

```
>>> from decimal import *
>>> getcontext().prec = 6
>>> Decimal(1) / Decimal(7)
Decimal('0.142857')
>>> getcontext().prec = 28
>>> Decimal(1) / Decimal(7)
Decimal('0.1428571428571428571428571429')
```

NOTE: Preciseness carries into arithmetic.

Other Types of Numbers

Type	Description	Example
More added via 3rd party extensions	Vectors, libraries, visualizations, plotting, dates …	Vectors.py, matplotlib.py, …

Different integer representations

For the integer 369…

Number System	Description	Example
Base-2	Binary number system	bin(369), 0b101110001
Base-8	Octal number system	oct(369), 0o561
Base-10	Decimal number system	Default numbering
Base-16	Hexidecimal number system	hex(369), 0x171

Mathematical Operators

- Subset of expressions…

Operator	Description	Example
X if Y else Z	X if Y is True, otherwise Z	status = ’OK’ if interfaceUp else ‘Problem’
X or Y	Logical OR (if X is False then evaluate Y)	if HW == 'NCS6000' or HW == ‘ASR1009’:
X and Y	Logical AND (If X is True evaluate Y)	if SW == ’5.2.5’ and HW == ‘NCS6008’:
X in Y, X not in Y	Membership in a group	if newBug in foundBugs:
X < Y, X <= Y, X > Y, X >= Y	Magnitude comparisons	if latency <= 10:
X == Y, X != Y	Evaluate Equality	if lineProtocol !== ‘up’:

https://docs.python.org/2/reference/expressions.html

Mathematical Operators

Operator	Expression	Example
X | Y	Bitwise OR*	Used to set a particular bit to 1
X ^ Y	Bitwise XOR*	Used to toggle bits
X & Y	Bitwise AND*	Used to find subnet network number
X + Y, X - Y	**Addition/Subtraction**	iterations = interations – 1 iterations -= 1
X * Y	**Multiplication**	installationDays = 3 * numberOfWeeks
X / Y, X // Y	True Division, Floor Division	5.0 / 3, 5.0 // 3 Floor division rounds down
X ** Y	Exponents	5 ** 2

* - https://en.wikipedia.org/wiki/Bitwise_operation

Mathematical Operators & Precedence

$$A * B + C * D = ?$$

- how does Python know what to do first?...Precedence
- Operators in previous 3 slides were ordered by precedence
- You can override default precedence by using parenthesis, X + (Y * Z)

https://docs.python.org/2/reference/expressions.html#operator-precedence

Strings

Strings

- An ordered collection of alpha-numeric characters
- Immutable sequences
  - Cannot be changed in place
  - Ordered left to right
- Encodings (Beyond Scope)
  - In Python 2.x, ASCII
  - Unicode for other language scripts
    - u’ආයුබෝවන්’
  - Raw to ignore escape sequences
    - r’C:\test\test\now\folder’
  - Binary, no encoding, directly to byte
    - b’Sp\xc4m’

String Basics

Assignment

Example	Description
S = ‘ ’	Empty string
S = ‘show run’, S = “show run”	Single or double quotes
S = ”router’s config”, S = ”router\’s config”	Mixing single/double quotes
S = ‘’’…*multiline...*””	Triple quoted multi-line block

String Basics

Manipulation

Example	Description
S = ‘show run’, S = S + ‘ning config’	Immutable, must be reassigned to make change
S1 + S2	Concatenate
S1 * 3	Repeat
S[i]	Indexing; find the position of a character
S[i:j]	Slicing; get a piece of the string

String Basics

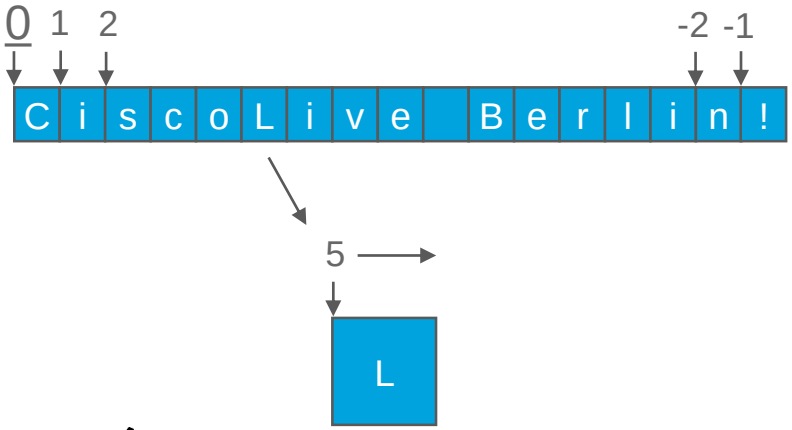
See Appendix: Formatting (Two Ways)

Example	Description
‘ip address %s %s’ % (addr, subnet)	Formatting expression
’ip address {0} {1}’.format(addr, subnet)	Formatting method

Indexing

On ordered sets (strings, lists, & tuples)

- Python numbering starts at 0
- Think in terms of “slicing” or “cutting” set apart
- Index number is cut line to the left of item
- Variable name + [cut line number]

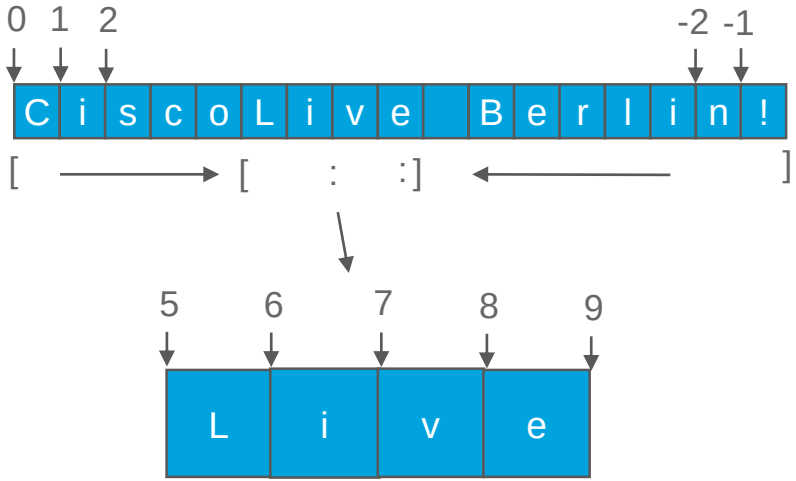



```
>>>
>>> string = 'CiscoLive Berlin!'
>>> string[5]
'L'
>>> string[-2]
'n'
>>>
```

Slicing

On ordered sets (strings, lists, & tuples)

- Find a substring inside a string
- Variable name + [startCutLine : endCutLine]
- Think of [] as start and end of substring




```
>>> string = 'CiscoLive Berlin!'
>>> string[:]  #same as string[0:0]
'CiscoLive Berlin!'
>>> string[5:9]
'Live'
>>> string[5:-1]
'Live Berlin'
```

String Methods

- Everything in Python is an object, even strings
- Holy Hand Grenade gift example
- At a conceptual level objects are:
  - Containers (objects) that have
  - Things (attributes) &
  - Instructions (Methods)
- All items an object holds are accessed with a “.”
- Attributes are accessed by attribute name, e.g., ObjectName.AttributeName
- Methods are accessed by method name + (), e.g., ObjectName.MethodName()

String Methods

```
>>>

>>> dir(S)

['__add__', '__class__', '__contains__', '__delattr__', '__doc__', '__eq__',
'__format__', '__ge__', '__getattribute__', '__getitem__', '__getnewargs__',
'__getslice__', '__gt__', '__hash__', '__init__', '__le__', '__len__',
'__lt__', '__mod__', '__mul__', '__ne__', '__new__', '__reduce__',
'__reduce_ex__', '__repr__', '__rmod__', '__rmul__', '__setattr__',
'__sizeof__', '__str__', '__subclasshook__', '_formatter_field_name_split',
'_formatter_parser', 'capitalize', 'center', 'count', 'decode', 'encode',
'endswith', 'expandtabs', 'find', 'format', 'index', 'isalnum', 'isalpha',
'isdigit', 'islower', 'isspace', 'istitle', 'isupper', 'join', 'ljust',
'lower', 'lstrip', 'partition', 'replace', 'rfind', 'rindex', 'rjust',
'rpartition', 'rsplit', 'rstrip', 'split', 'splitlines', 'startswith',
'strip', 'swapcase', 'title', 'translate', 'upper', 'zfill']

>>>
```

String Method Example

Input as screen scrape from router CLI

```
>>>ShowVersion = “””
Cisco IOS XR Software, Version 5.2.5.37I
Copyright (c) 2013-2014 by Cisco Systems, Inc.


Build Information:
 Built By     : ahoang
 Built On     : Mon Dec 14 02:53:00 PST 2015
 Build Host   : iox-lnx-010
 Workspace    : /auto/iox-lnx-010-san1/production/5.2.5.37I.SIT_IMAGE/all/workspace
 Version      : 5.2.5.37I
 Location     : /opt/cisco/XR/packages/


cisco NCS-6000 () processor
System uptime is 48 weeks, 21 hours, 22 minutes
“””
```

String Method Example

As stored in memory

```
>>> ShowVersion

'\nFri Jan  6 11:36:37.930 PST\n\nCisco IOS XR Software, Version
5.2.5.37I\nCopyright (c) 2013-2014 by Cisco Systems, Inc.\n\nBuild
Information:\n Built By     : ahoang\n Built On     : Mon Dec 14 02:53:00 PST
2015\n Build Host   : iox-lnx-010\n Workspace    : /auto/iox-lnx-010-
san1/production/5.2.5.37I.SIT_IMAGE/all/workspace\n Version      : 5.2.5.37I\n
Location     : /opt/cisco/XR/packages/\n\ncisco NCS-6000 () processor \nSystem
uptime is 48 weeks, 21 hours, 22 minutes\n'

>>>
```

String.count(*substring*) method

How many times a substring occurs in string

- Some methods take additional arguments in parenthesis
- Add a substring, one or more characters, argument to method
- Useful when wanting to know how many times a substring occurs in CLI output
- Can also search a slice of a string by adding a start and end argument

```
>>>
>>> ShowVersion.count('Version')
2
>>>
```

String.index(*substring*) method

Find location of substring in main string

- Add a substring, one or more characters, argument to method
- Useful when trying to locate where in CLI output a substring starts
- .find() and .index() do same thing

```
>>>
>>> ShowVersion.find('Version')
53
>>>
```

String.upper() & String.lower() methods

Normalizing data

- This method takes no arguments
- Add a substring, one or more characters, argument to method
- Useful when normalizing user input as you can’t predict how someone will type a word

```
>>> len('Version')
7
>>> text = ShowVersion[53: 53 + 7]
>>> text
'Version'
>>> text.upper()
'VERSION'
>>> text.lower()
'version'
```

String.split(*[seperator]*) method

Break a string into a list of words

- This method takes no or some arguments
- Use List() function if you want a list of characters
- Call method on string, pass in delimiter as argument

```
>>> text = 'This is SPAM!'
>>> text.split()
['This', 'is', 'SPAM!']
>>> text = 'SPAM!,SPAM!,SPAM!'
>>> text.split(',')
['SPAM!', 'SPAM!', 'SPAM!']
>>> text = 'SPAM!'
>>> list(text) #Creating list of characters
['S', 'P', 'A', 'M', '!']
```

String.join(*iterable*) method

- This method takes a collection of strings as an argument
- Add a delimiter argument to method
- Useful when wanting to join an iterable list of strings together into one string
- Reversed! Call method on delimiter, pass in ordered set as argument

```
>>> text = ['SPAM!', 'SPAM!', 'SPAM!']
>>> type(text)
<type 'list'>
>>> text = ','.join(text)
>>> text
'SPAM!,SPAM!,SPAM!'
```

Lists, Dictionaries, & Tuples

Lists

- An ordered collect of arbitrary objects (can access by index)
- Denoted by [ ]
- Iterable
- Mutable (can be changed in place unlike a string)
- Heterogeneous (you can put any type of object in the same list)
- Nestable (you can put lists in lists in lists)
- Use when you want the ability to modify collection of objects

List Basics

Assignment

```
>>>
>>> L = []     #Assign empty set
>>>
>>> L = [456, 'xyz', 3.14, [1,2,3], 'Cisco']
>>>
>>> L = list('CiscoLive')
['C', 'i', 's', 'c', 'o', 'L', 'i', 'v', 'e']
>>>
```

List Basics

Comprehension assignment

- Make a small list based from a larger collection of objects
- Evaluate all the entities in larger set by some discerning criteria
- If they meet the criteria, put a copy of them in the smaller set

List Creation, comprehensions

- Make a small list based from a larger collection of objects

```
>>>
>>> L = [456, 'xyz', 3.14, [1,2,3], 'Cisco']
>>>
>>> L2 = [X for X in L if type(X) == str]
>>> L2
['xyz', 'Cisco']
>>>
```

https://docs.python.org/2/tutorial/datastructures.html#list-comprehensions

Basic List Operations

```
>>> L = [456, 'xyz', 3.14, [1,2,3], 'Cisco']
>>> type(L)
<type 'list'>
>>> L[3]   #Indexing object in list
[1, 2, 3]
>>> L[3][1]  #Indexing object in nested list
2
>>> L[1:2]    #slicing objects from list
['xyz', 3.14, [1,2,3]]
>>> len(L)
5
```

List Methods

```
>>> dir(L)

['__add__', '__class__', '__contains__', '__delattr__', '__delitem__',
'__delslice__', '__doc__', '__eq__', '__format__', '__ge__',
'__getattribute__', '__getitem__', '__getslice__', '__gt__', '__hash__',
'__iadd__', '__imul__', '__init__', '__iter__', '__le__', '__len__', '__lt__',
'__mul__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__',
'__reversed__', '__rmul__', '__setattr__', '__setitem__', '__setslice__',
'__sizeof__', '__str__', '__subclasshook__', 'append', 'count', 'extend',
'index', 'insert', 'pop', 'remove', 'reverse', 'sort']

>>> [x for x in dir(L) if '_' not in x]

['append', 'count', 'extend', 'index', 'insert', 'pop', 'remove', 'reverse',
'sort']
```

List Append() & Extend()

Adding to the end of a list

- Use .append() to add an object to the end of a list

```
>>>
>>> L = [1, 2, 3]
>>> L.append([4, 5, 6])
>>> L
[1, 2, 3, [4, 5, 6]]
>>>
```

- Use .extend to combine two list collections

```
>>>
>>> L = [1, 2, 3]
>>> L.extend([4, 5, 6])
>>> L
[1, 2, 3, 4, 5, 6]
>>>
```

* Notice we just call method, not reassign variable!

List Index() & Insert()

Finding and adding an object at a particular index

- Use .index(object) to find and objects index

```
>>>
>>> L = [456, 'xyz', 3.14]
>>> L.index(3.14)
2
>>>
```

- Use .insert(index, object) to insert an object at a particular index

```
>>>
>>> L = [456, 'xyz', 3.14]
>>> L.insert(1, 'router5')
>>> L
[456, 'router5', 'xyz', 3.14]
>>>
```

List Pop() & Remove()

Removing objects from a list

- Use .pop(index) to remove obj at specified index
- Returns the popped value

```
>>> L
[456, 'router5', 'xyz', 3.14]
>>> L.pop(1)
'router5'
>>> L
[456, 'xyz', 3.14]
```

- Use .remove(object) to remove an object by name/value

```
>>> L
[456, 'router5', 'xyz', 3.14]
>>>
>>> L.remove('router5')
>>> L
[456, 'xyz', 3.14]
```

Dictionaries

- An unordered collection of objects
  - You cannot refer to objects by index!
  - Key values will be hashed for faster access
  - Used to make code more readable
- Denoted with { key : value, key : value, … }
- Mutable (can be changed in place)
- Heterogeneous (you can put any type of object in the same list)
- Nestable (you can put dictionary in dictionary in dictionary)
- Must create a dictionary before you can reference or add to it

Dictionary Basics

Assignment

```
>>> D = {}
>>> D = {'hostname' : 'router1', 'loop0' : '1.1.1.1'}
>>> D
{'hostname': 'router1', 'loop0': '1.1.1.1'}
>>>
>>> D = dict(hostname='router1', loop0='1.1.1.1')
>>> D
{'loop0': '1.1.1.1', 'hostname': 'router1'}
>>>
```

Dictionary Basics

Referencing values

```
>>> D = {'hostname' : 'router1', 'loop0' : '1.1.1.1'}
>>> 'hostname' in D
True
>>> D['hostname']
'router1'
```

Dictionary Basics

Adding values

- Add a key:value pair with *Dictionary[key] = value* expression

```
>>> D = {'hostname' : 'router1', 'loop0' : '1.1.1.1'}
>>> D['device'] = 'switch'
>>> D
{'device': 'switch', 'hostname': 'router1', 'loop0': '1.1.1.1'}
```

Dictionary Basics

Changing a value

- Same syntax as adding a value

```
>>> D = {'hostname' : 'router1', 'loop0' : '1.1.1.1'}
>>> D['hostname'] = 'switch1'
>>> D
{'device': 'switch', 'hostname': 'switch1', 'loop0': '1.1.1.1'}
>>>
>>> len(D)
3
>>>
```

Dictionary Methods

```
>>> [x for x in dir(D) if '_' not in x]

['clear', 'copy', 'fromkeys', 'get', 'has_key', 'items',
'iteritems', 'iterkeys', 'itervalues', 'keys', 'pop', 'popitem',
'setdefault', 'update', 'values', 'viewitems', 'viewkeys',
'viewvalues']

>>>
```

Dictionary Methods .pop(key) & .clear()

Removing items from dictionary

- Use .pop(key) to remove key
- Returns popped value

```
>>> D = {'hostname' : 'router1',
'loop0' : '1.1.1.1', 'device' :
'switch'}

>>> D.pop('hostname')

'router1'

>>> D

{'device': 'switch', 'loop0':
'1.1.1.1'}
```

- Use .clear() to remove all key:value pairs from dictionary

```
>>> D = {'hostname' : 'router1',
'loop0' : '1.1.1.1', 'device' :
'switch'}

>>> D.clear()

>>> D

{}

>>>
```

Dictionary Methods .has_key(key) & .keys()

Testing for and seeing keys

- Use .has_key(key) to test for key
- Returns boolean

```
>>> D = {'hostname' : 'router1',
'loop0' : '1.1.1.1', 'device' :
'switch'}

>>> D.has_key('interfaces')

False

>>> if 'loop0' in D:

...     print 'yes'

yes
```

- Use .keys() to see all keys in dictionary

```
>>>

>>> D.keys()

['device', 'hostname', 'loop0']

>>>
```

Dictionary Methods .values() & .items()

- Use .values() to get all values

```
>>> D = {'hostname' : 'router1',
'loop0' : '1.1.1.1', 'device' :
'switch'}

>>> D.values()

['switch', 'router1', '1.1.1.1']

>>>
```

- Use .items() to get all key:value pairs

```
>>> D = {'hostname' : 'router1',
'loop0' : '1.1.1.1', 'device' :
'switch'}

>>> D.items()

[('device', 'switch'), ('hostname',
'router1'), ('loop0', '1.1.1.1')]

>>>
```

Tuples

- An ordered collect of arbitrary objects (can access by offset)
- **Denoted by (,)**
- Iterable
- **Immutable (CANNOT be changed in place like a string)**
- Heterogeneous (you can put any type of object in the same tuple)
- Nestable (you can put tuples in tuples in tuples)
- **Use when you DO NOT want the ability to modify collection of objects**

Tuples

```
>>> T = ('Cisco')
>>> type(T)
<type 'str'>
>>> T = ('Cisco',)
>>> type(T)
<type 'tuple'>
>>> T
('Cisco',)
>>>
```

```
>>> T = ('CiscoLive', 'Berlin',
'!')
>>> type(T)
<type 'tuple'>
>>> T[0]
'CiscoLive'
>>> T[1:]
('Berlin', '!')
>>> Len(T)
3
```

Files

Files

See Appendix for further study

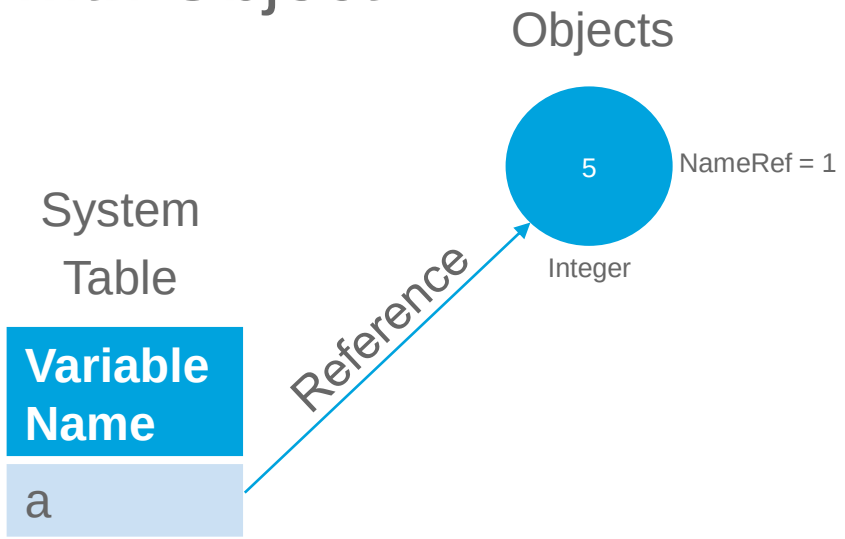
- Out of box you can work with text files
- Other file types, e.g., .xls or .pdf, require 3rd party modules
- When you want to write objects to file for later use, you have to serialize the data in memory into a stream of characters
  - Pickle
  - JSON

Dynamic Typing

- Unlike C, C++, and Java, you don't statically define variables up front in Python
- You do have to initialize them to update them
  - counter = 0 must be created before counter += 1
- Variables are created dynamically
  - Variable is just a name
  - That name lives in a system table with a link to an object
  - Type information is not tied with the variable name but the object itself
- Variable names are replaced with objects in an expression when run
- A variable must be assigned before referenced in an expression
- Be careful when assigning and modifying variables in your programs

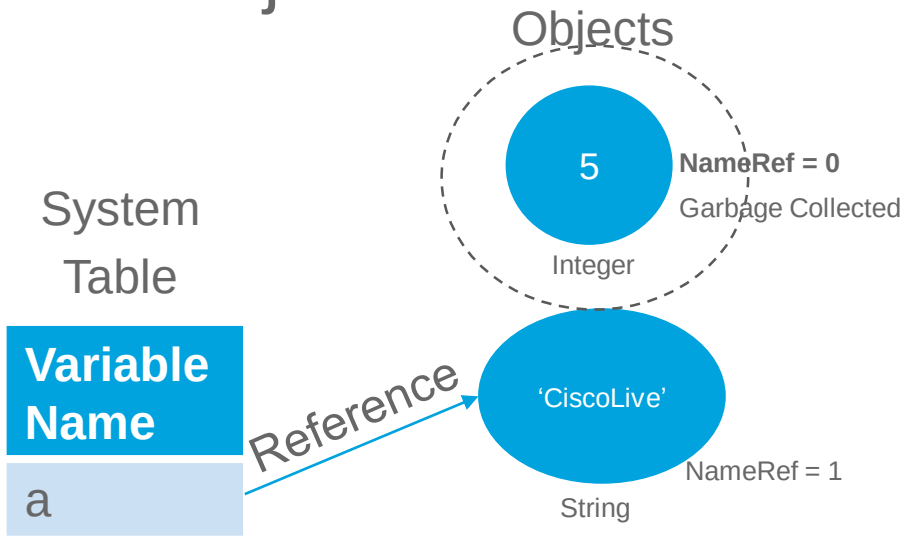
Dynamic Typing: Type lives with Object

```
>>>
>>> a = 5
>>> type(a)
<type 'int'>
```



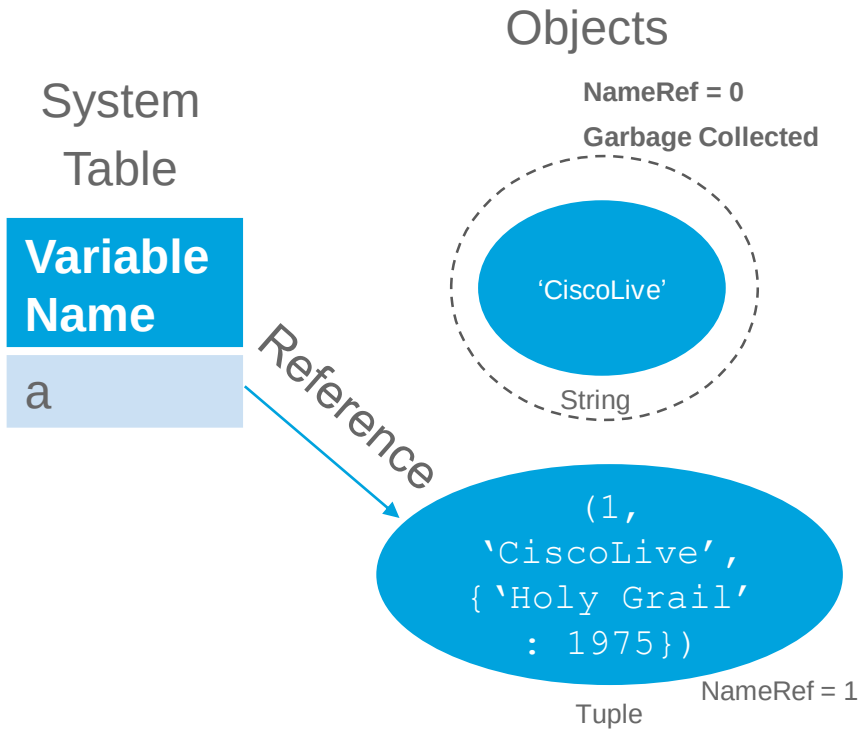
Dynamic Typing: Type lives with Object

```
>>>
>>> a = 5
>>> type(a)
<type 'int'>
>>> a = 'CiscoLive'
>>> type(a)
<type 'str'>
```



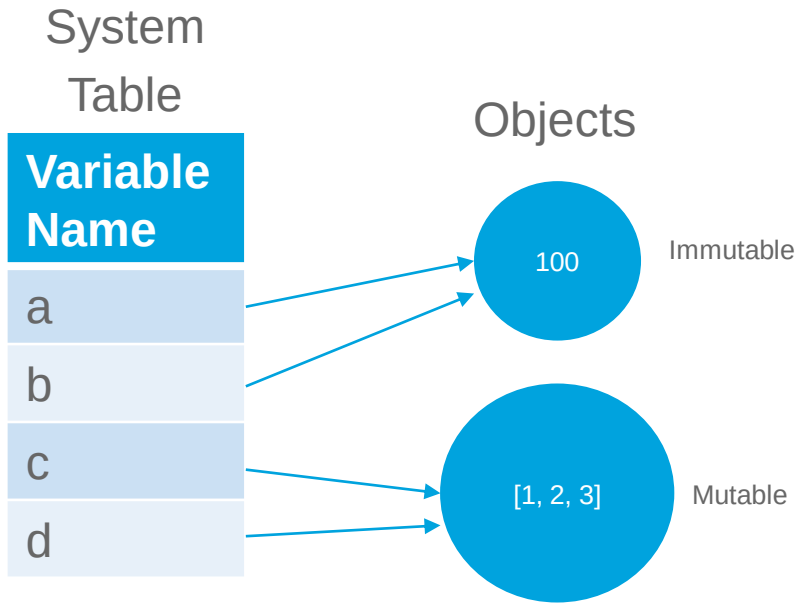
Dynamic Typing: Type lives with Object

```
>>>

>>> a = 5

>>> type(a)

<type 'int'>

>>> a = 'CiscoLive'

>>> type(a)

<type

>>> a = (1, 'CiscoLive', {'Holy
Grail' : 1975})

>>> type(a)

<type 'tuple'>
```



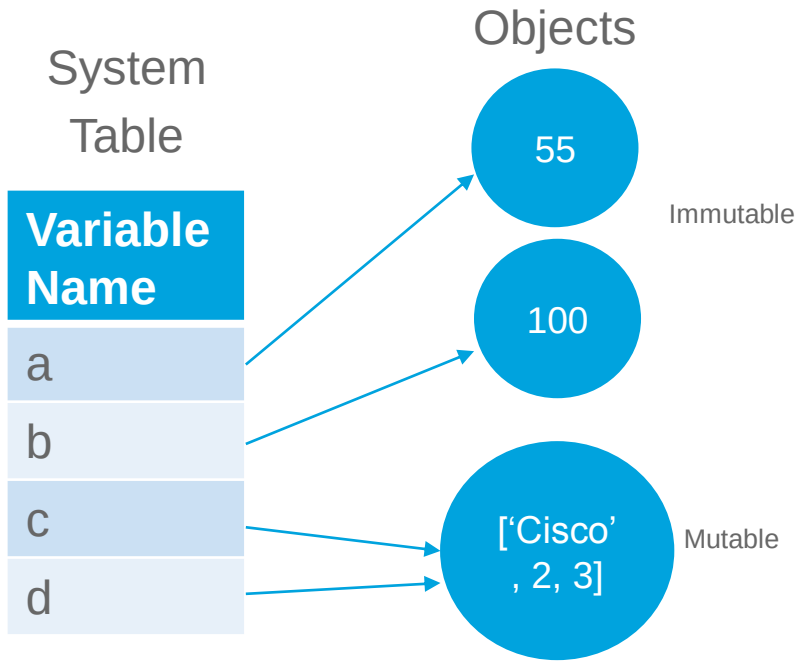
Dynamic Typing & Shared References

```
>>>
>>> a = 100
>>> b = a
>>> a == b  #checking value
True
>>> a is b  #checking sharing
True
>>> c = [1,2,3]
>>> d = c
```



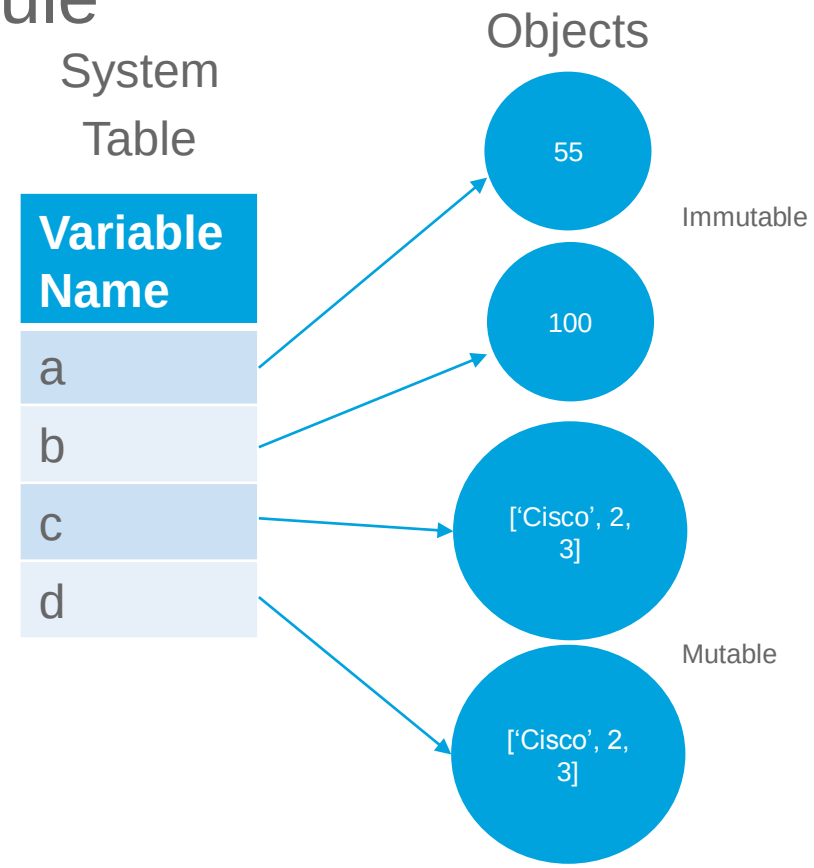
Dynamic Typing - Immutable vs Mutable Changes

```
>>> a = 55
>>> a == b  #checking value
False
>>> a is b  #checking sharing
False
>>> c[0] = 'Cisco' #Change int to str
>>> c
['Cisco', 2, 3]
>>> d
['Cisco', 2, 3]  #d Changes
```



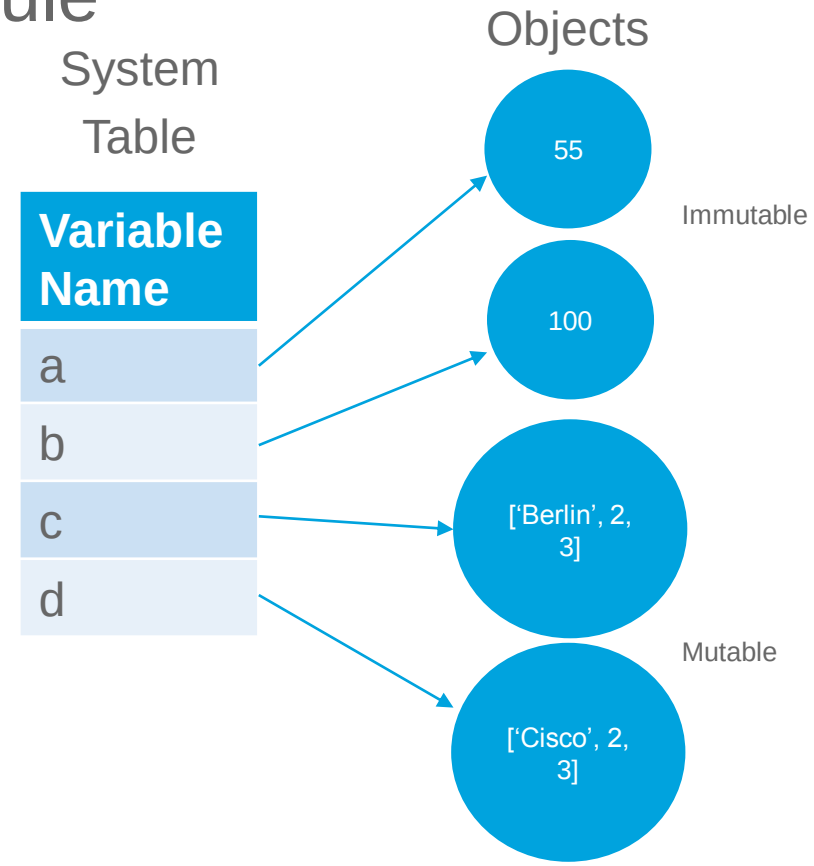
Dynamic Typing - Copy Module

```
>>> import copy
>>> c = ['Cisco', 2, 3]
>>> d = copy.copy(c)
>>> c
['Cisco', 2, 3]
>>> d
['Cisco', 2, 3]
```



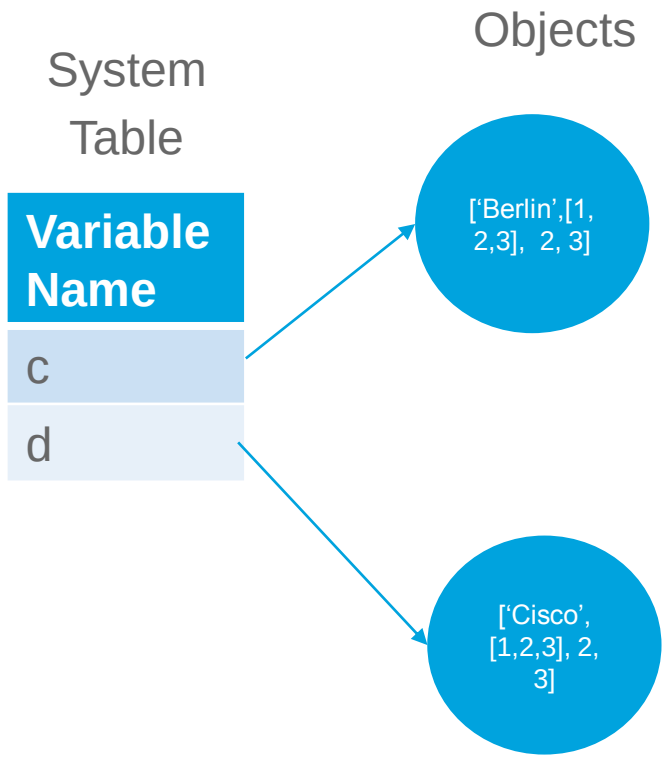
Dynamic Typing - Copy Module

```
>>> c[0] = 'Berlin'
>>> c
['Berlin', 2, 3]   #Expected change
>>> d
['Cisco', 2, 3]    #Now no change
```



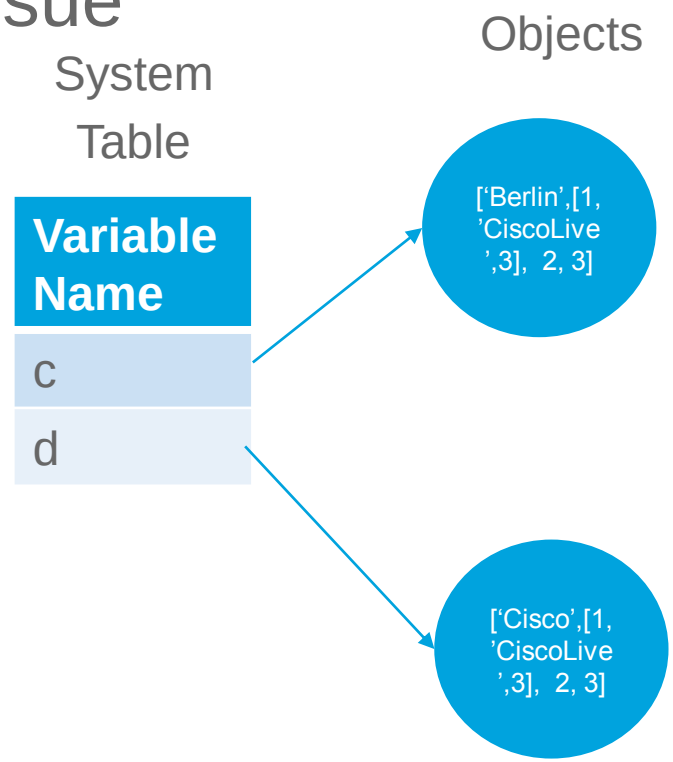
Dynamic Typing – Nesting

```
>>> c = ['Cisco', [1, 2, 3], 2, 3]
>>> d = copy.copy(c)
>>> c
['Cisco', [1, 2, 3], 2, 3]
>>> d
['Cisco', [1, 2, 3], 2, 3]
>>> c[0] = 'Berlin'   #same change
>>> c
['Berlin', [1, 2, 3], 2, 3] #expected
>>> d
['Cisco', [1, 2, 3], 2, 3] #no change
```



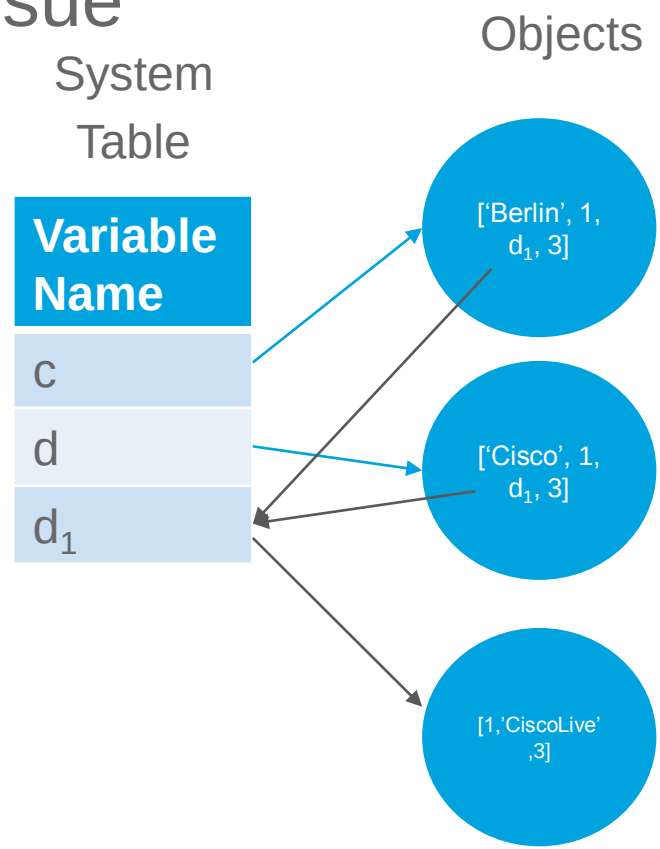
Dynamic Typing – Nesting Issue

```
>>> c

['Berlin', [1, 2, 3], 2, 3]

>>> d

['Cisco', [1, 2, 3], 2, 3]

>>> c[1][1] = 'CiscoLive'

>>> c

['Berlin', [1, 'CiscoLive', 3], 2,
3]

>>> d

['Cisco', [1, 'CiscoLive', 3], 2, 3]
```



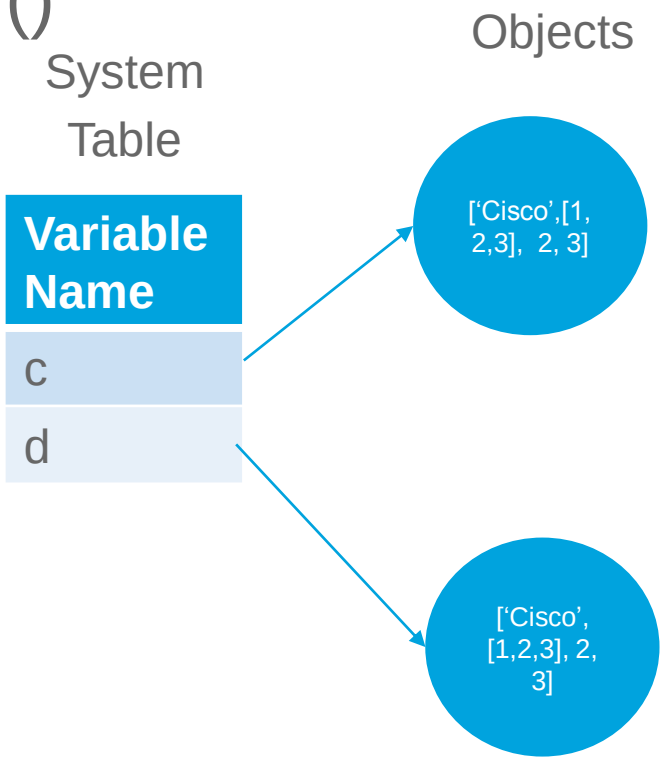
Dynamic Typing – Nesting Issue

```
>>> c
['Berlin', [1, 2, 3], 2, 3]
>>> d
['Cisco', [1, 2, 3], 2, 3]
>>> c[1][1] = 'CiscoLive'
>>> c
['Berlin', [1, 'CiscoLive', 3], 2, 3]
>>> d
['Cisco', [1, 'CiscoLive', 3], 2, 3]
```



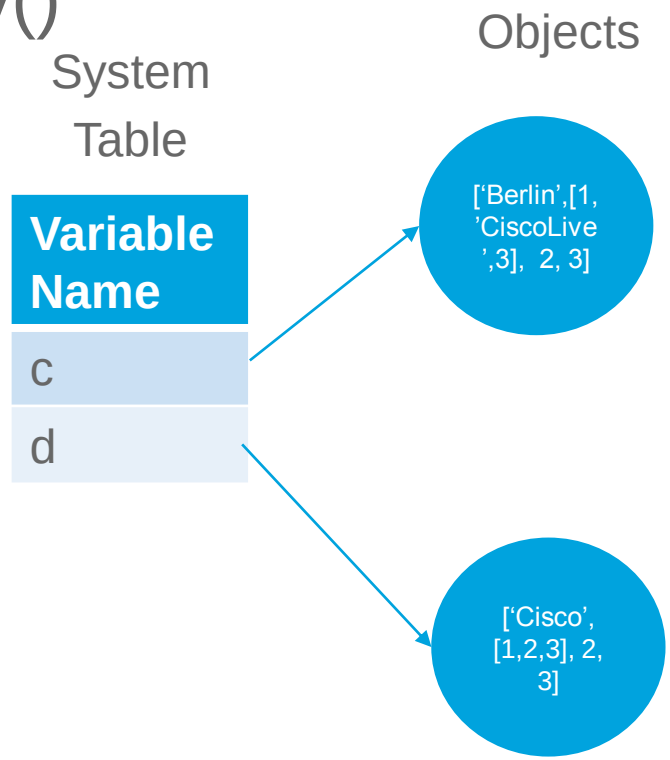
Dynamic Typing – .deepcopy()

```
>>> import copy
>>> c = ['Cisco', [1, 2, 3], 2, 3]
>>> d = copy.deepcopy(c)
>>> c
['Cisco', [1, 2, 3], 2, 3]
>>> d
['Cisco', [1, 2, 3], 2, 3]
```



Dynamic Typing – .deepcopy()

```
>>>

>>> c[0] = 'Berlin'

>>> c[1][1] = 'CiscoLive'

>>> c

['Berlin', [1, 'CiscoLive', 3], 2,
3]

>>> d

['Cisco', [1, 2, 3], 2, 3]

>>>
```



Polymorphism

- For now, before Object Oriented Programming…
- Basically that operators can be used on different object types

```
>>> 5 + 5
10
>>> L1 = [1, 2, 3]
>>> L2 = [4,5,6]
>>> L1 + L2
[1, 2, 3, 4, 5, 6]
>>> L1 * 3
[1, 2, 3, 1, 2, 3, 1, 2, 3]
>>> str1, str2 = 'Monty', ' Python'
>>> str1 + str2
'Monty Pyton'
```

```
>>>
>>> str1 + L1
Traceback (most recent call
last):
  File "<stdin>", line 1,
in <module>
TypeError: cannot
concatenate 'str' and
'list' objects
>>>
```

Procedural Statements & Syntax

Semantics

- Up to now we've been talking about Semantics, what things mean.
  - Grammatically speaking
    - A router is a noun
    - A noun has certain properties
    - Specific nouns, like a router, have certain characteristics
  - Programmatically
    - A string is an object
    - An object has a memory address
    - Specific objects, like Strings or Dictionaries, have certain properties

String

- Collection of Alpha Numeric characters
- Immutable
- Ordered
- Accessed by index

Dictionary

- Collection of Objects
- Mutable
- Unordered
- Accessed by key

Syntax

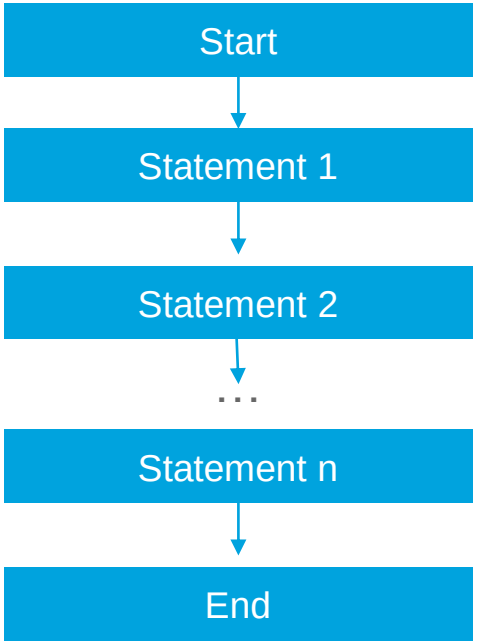
- We are going to move to Syntax now, how things fit together
  - Grammatically speaking
    - The grammar of a language
    - In English, *Noun-Verb-DirectObjectNoun* is syntax of sentence
    - Sentences end with periods
  - Programmatically speaking
    - Rules for how things fit together sequentially
    - Rules to come...

Assignment Syntax

Operation	Description
Hostname = ‘router1’	Basic variable assignment
A, B = [5, 6]	List assignment
A, B = (5, 6)	Tuple assignment
A, B, C, D, E, F = ‘router’	Sequence Assignment
A = function(), A = object.method()	Function, method assignment
Counter += 1	Counter = Counter + 1

Procedural Programming Diagram

- A Python script executes a series of statements, in order, from the start of the script to the end.



Procedural Programming

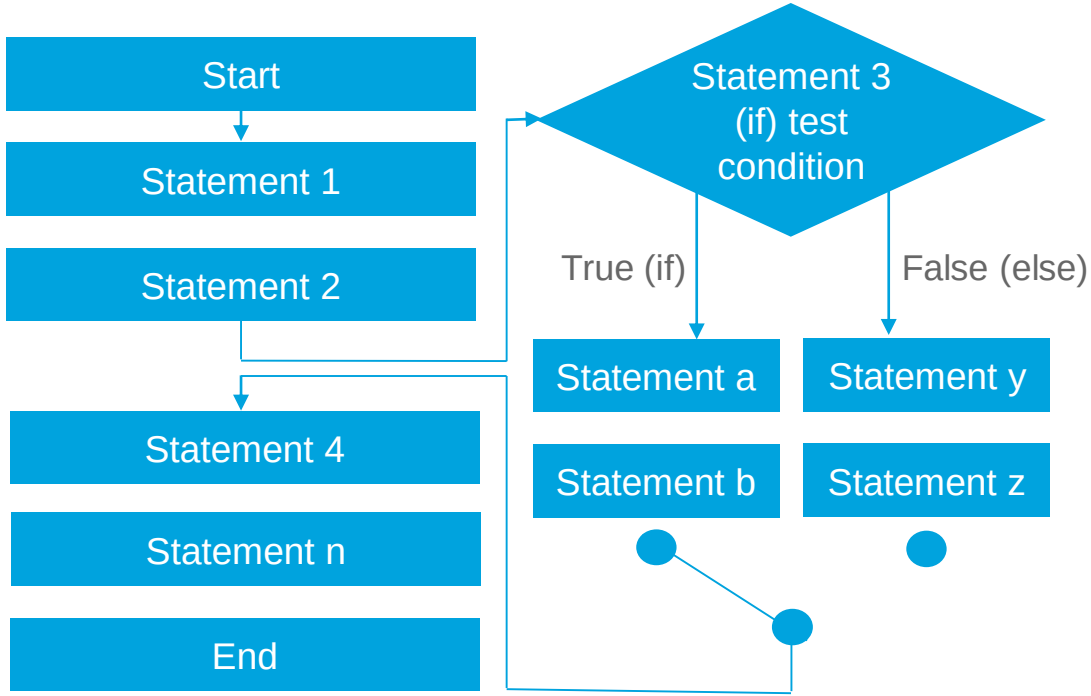
Problem statement: Find, if any, sensitive verbiage in a TAC case that would flag the case as higher severity than currently assigned.

```
…
logger.info('Searching for sensitive verbiage for TAC case %s...' % caseNumber
logger.info('Collecting casekwery webpage...')
r = requests.get(TACWebsite + '%s' %  caseNumber, auth=(username, password))
logger.info('Recieved casekwery webpage.')
logger.info('Checking for verbiage=%s' % verbiage)
verbiageCount = r.text.lower().count(verbiage)
logger.debug('verbiageCount=%i' % verbiageCount)
…
```

If-else statements

If-else statements, single branch Diagram

- Runs a sub-block of code once if condition exists
- Mark the position in list
- Test for a condition and branches one way or the other
- Runs a sub-block of code
- Returns to marked position and proceeds

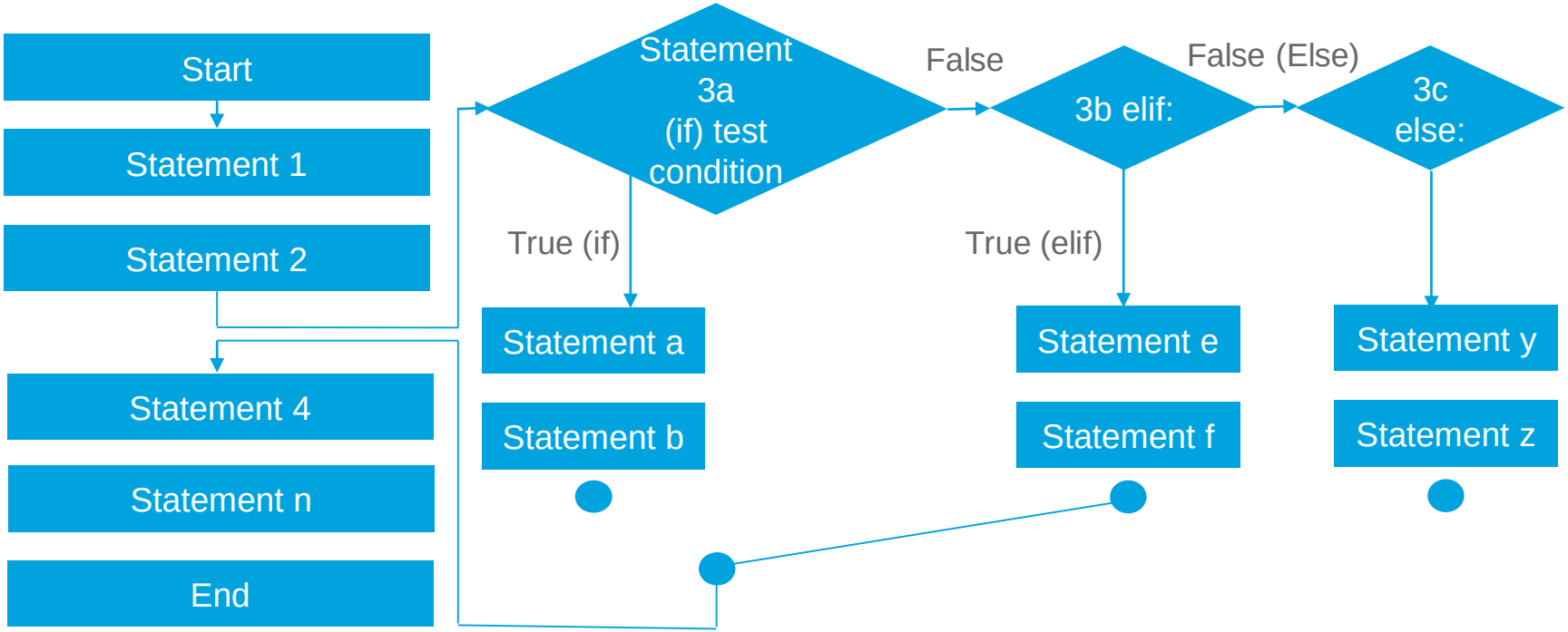


If-else statement

Single branch, Production Example

```
…
logger.info('Searching for sensitive verbiage for TAC case %s...' % caseNumber
logger.info('Collecting casekwery webpage...')
r = requests.get(TACWebsite + '%s' %  caseNumber, auth=(username, password))
logger.info('Recieved casekwery webpage.')
logger.info('Checking for verbiage=%s' % verbiage)
verbiageCount = r.text.lower().count(verbiage)
logger.debug('verbiageCount=%i' % verbiageCount)
if verbiageCount > 0:
    verbiageFound = True
else:
    verbiageFound = False
logger.debug('verbiage=%s, verbiageFound=%s' % (verbiage, verbiageFound))
…
```

If-else statements, multi-branch Diagram



If-else statements, multi-branch

Problem Statement: Extract a device's hostname, if present, out of a TAC case so that you can apply business logic to the notification process.

- In our example the hostname starts with "BER0"

If-else statements

Multi-branch Production Example

```
title = TACCaseDict[case].caseTitle      #TAC case title
title = title.lower()       #Normalizing the data
description = TACCaseDict[case].caseDescription        #TAC case description
description = description.lower()        #Normalizing the data
logger.info('tac case title=%s' % title)
if 'BER0' in title:
    logger.info('Extracting BER0 hostname from case title...')   #finding and slicing hostname
    hn_start = title.find('BER0')
    hn_end = title[hn_start:].find(' ')
    hn_end += hn_start
    logger.debug('hn_start=%s, hn_end=%s' % (hn_start, hn_end))
    TACCaseDict[case].hostname = title[hn_start:hn_end]
```

If-else statements
Multi-branch Production Example

```
elif 'BER0' in description:
    logger.info('Extracting BER0 hostname from case description...')
    hn_start = description.find(' BER 0')
    hn_end = description[hn_start:].find(' ')
    hn_end += hn_start
    logger.debug('hn_start=%s, hn_end=%s' % (hn_start, hn_end))
    TACCaseDict[case].hostname = description[hn_start:hn_end]
else:
    logger.debug('Setting hostname=False')
    TACCaseDict[case].hostname = False
```

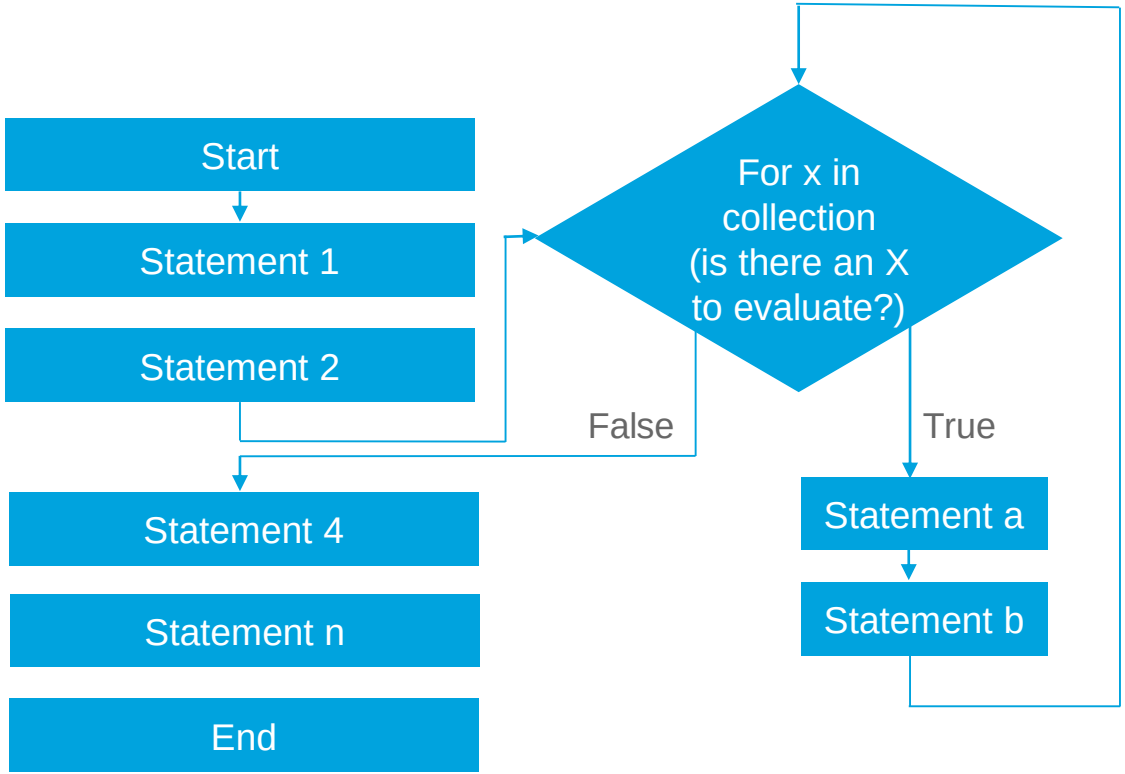
for loops

For loops

- **All loops run a certain number of times**
- **For loops conceptually deal with a collection of things**
- The certain number that defines how many times can be:
  - A integer/counter
  - A range(start, end) of numbers
  - A collection of objects
  - An index number in enumerate(a collection of objects)
- **Every object in the collection is an iteration of 1**

For loops Diagram

- Mark the position in list
- Iterates through collection of objects
- If there is an iteration to be run, runs a sub-block of code
- Returns to marked position and proceeds



For Loop Production Example

Problem Statement: Need to poll TAC cases at a set interval and create a Python dictionary of new TAC cases. Take the resulting Python dictionary and give the ability to sort and create new dictionaries , by severity, for further processing at a later time.

- TACCaseDict is a dictionary of all TAC case objects
- TAC case number is the key in the dictionary
- _return_filter is a vairable specifying what type of case you want to sort, i.e., S1 or S2
- TACCaseDict.severity is an attribute containing case severity level
- s1_TACCaseDict is new dictionary for all S1 cases
- s2_TACCaseDict is new dictionary for all S2 cases

For Loop Production Example

```
if _return_filter == 's1':     #tell program to check for S1 cases
   s1_TACCaseDict = {}
   logger.info('Building Dict for s1 cases...')
   for TACCase in TACCaseDict:
      if TACCaseDict[TACCase].severity[0:2] == 'S1':
         s1_TACCaseDict[TACCase] = TACCaseDict[TACCase]
elif _return_filter == 's2':       #or telling program to check for S2 cases
   s2_TACCaseDict = {}
   logger.info('Building Dict for s2 cases...')
   for TACCase in TACCaseDict:
      if TACCaseDict[TACCase].severity[0:2] == 'S2':
         s2_TACCaseDict[TACCase] = TACCaseDict[TACCase]
```

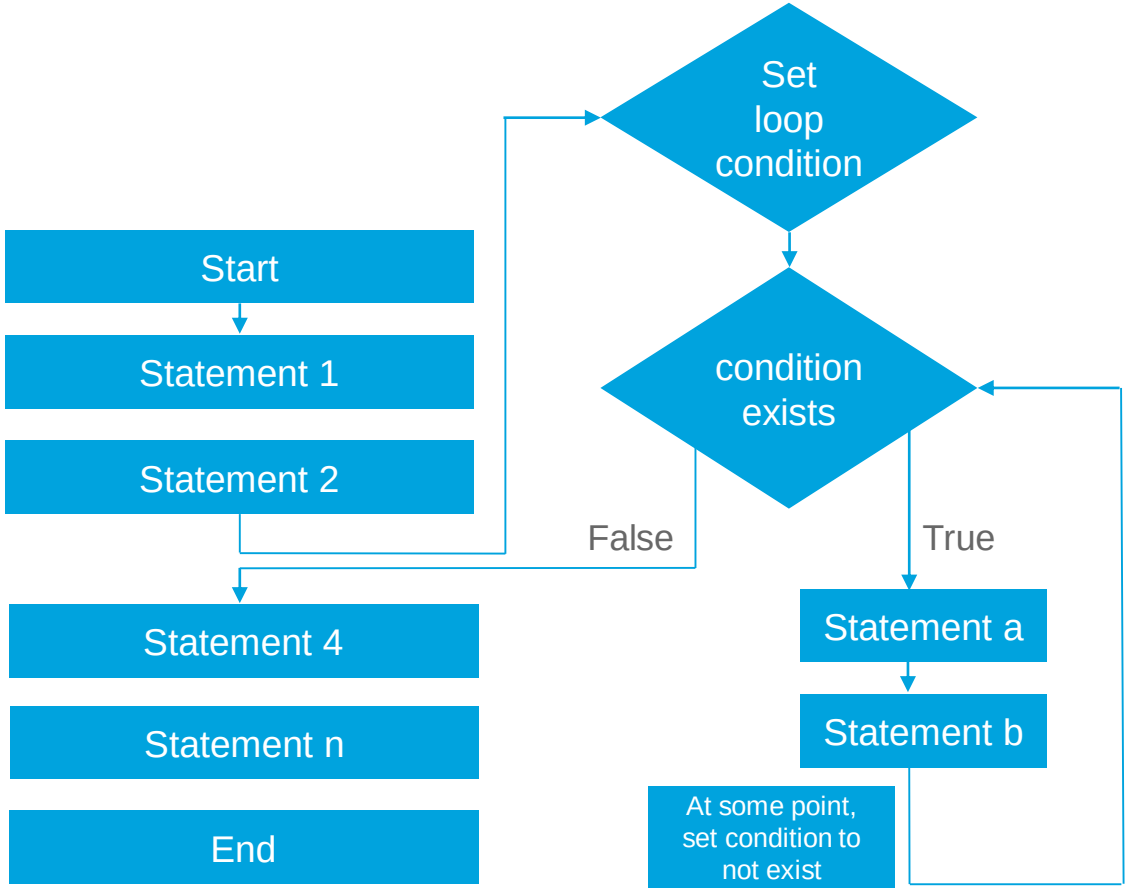
while loops

While loops

- **All loops run a certain number of times**
- **While loops conceptually deal with a condition**
  - The loop runs while the loop condition exists
  - You set the condition before entering the loop
  - Condition is changed inside subblock of loop code at some point
- Loop will run forever if condition is not changed
  - Program hangs

While loops

- Mark the position in list
- Set loop condition to exist
- If there is an iteration to be run, runs a sub-block of code
- At some point, set loop condition to not exist
- Returns to marked position and proceeds



While Loop Production Example

Problem Statement: Need to monitor a S1 TAC case for one hour for the initial cisco.webex.com troubleshooting support call URL to text to larger team.

- Personal Conf Room URL – Looking for this in this example
  - https://cisco.webex.com/join/ttaggart
  - https://cisco.webex.com/join/ttaggart

- Scheduled Webex URL Link – Not Looking for this
  - https://cisco.webex.com/ciscosales/j.php?MTID=m34bf69290f8564f81b4699d6c999994a

While Loop Production Example

```
def find_webex_url(caseNumber):
   TACWebsite = 'http://www-tac.cisco.com/Teams/ks/c3/casekwery.php?Case='
   logger.info('Searching for initial Webex URL for TAC case %s...' % caseNumber)
   initialWebexFound = False  #Setting While Loop Condition
   response = False
   minutes = 0
```

While Loop Production Example

```
while initialWebexFound == False:  #Calling While Loop
   if minutes == 60:  #Checking to see if we've been checking for hour
      logger.info('Script has reached its timeout  Aborting program.')
      response = {'url': None, 'reason' : 'timed_out'}
      break
   logger.info('Collecting casekwery webpage...')
   r = requests.get(TACWebsite  + str(caseNumber), auth=(user, password))
   logger.info('Recieved casekwery webpage.')
   webexURLStart = r.text.find('https://cisco.webex.com') #Checking for URL
   logger.debug('webexURLStart=%i' % webexURLStart)
```

While Loop Production Example

```
     if webexURLStart != -1:  #-1 means string not found.
        webexURLEnd = r.text[webexURLStart:].find("’")
        webexURLEnd += webexURLStart
        webexURL = r.text[webexURLStart: webexURLEnd]
        response = {'url': webexURL}
        initialWebexFound = True   #Resetting While Loop Condition
    else:
        time.sleep(60)
    minutes += 1
logger.debug('returning %s' % response)
return response
```

Transition to Function Structured Code

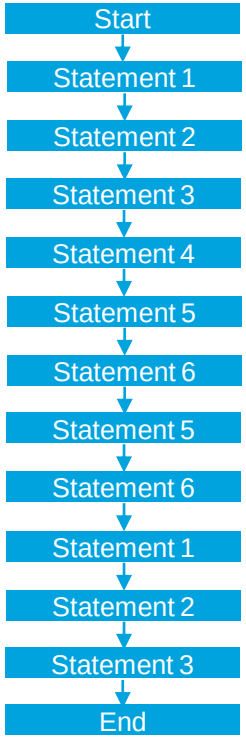
Transition to Functions

See appendix for further study (name space & arguments)

- **We've talked about syntax and semantics, now onto structure**
- **Your evolution in writing code will go from procedural, to function based, to object-oriented, to a mixture of all styles**
- A function is an object that groups lines of code together for reuse
- Small amount of new syntax
- But, several big new ideas about programming
- **We are at limits of a 101 class**

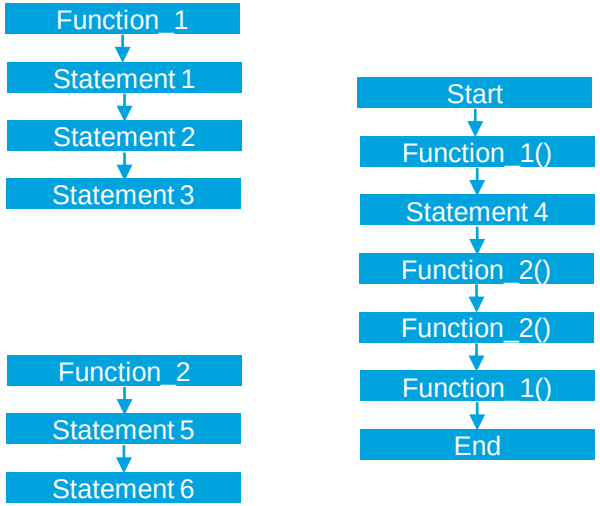
Long Redundant Procedural Code

- Code is long list of statements
- Certain statements repeat
- This makes refactoring difficult
  - Code harder to read and follow
  - Same change multiple places
- Need to make structure simpler
- Need to be able to fix redundant code in one place



Function-ized Code

- Pull redundant lines into functions
- Functions create an object, containing executable code, and assigns it a name
- Call functions by name + ()
- Pass in (*arguments)* if needed
- Benefits
  - Maximizes code reuse
  - Minimizes Redundancy
  - Breaks big program into smaller one, procedural decomposition – allows for TDD



Tips for Writing Good Code

Tips for Good Coding

- How you write code is just as important, if not more, than the fact you do write code
  - https://www.python.org/doc/humor/
  - `>>> import this`
  - Read PEP8 – Style Guide, https://www.python.org/dev/peps/pep-0008/
- Get consensus on how you indent and stick to it as group (tabs or spaces)
- Learn GIT & use it correctly
  - https://www.codecademy.com/learn/learn-git
  - https://git-scm.com/documentation
  - Branch alot

Tips for Good Coding

- Embrace refactoring, refactoring is good
- Follow framework to write code
  - Collect requirements and document problem
  - Write rough algorithms & document data structures
  - Write prototype code
  - Analyze work
  - Rewrite for clarity, speed, and ease of reading
- Log and document your code very, very, very well!
  - https://docs.python.org/2/library/logging.html
  - https://docs.python.org/2/howto/logging.html
- Use key:value pairs in your logging to leverage outside tooling systems

Tips for Good Coding

- Timestamp your work for benchmarking and future enhancements
- Keep track of variable names
- Write very readable code, you'll have to come back to it long after you write it
- Investigate Test Driven Development model, TDD
- Master regular expressions

Documentation & Resources

Further Study

- Official Tutorial
  - https://docs.python.org/3/tutorial/index.html
- Official Documentation
  - https://docs.python.org/2/
  - https://docs.python.org/3/
- Python Enhancement Proposals
  - https://www.python.org/dev/peps/
  - https://www.python.org/dev/peps/pep-0008/
- Intro Books
  - http://learnpythonthehardway.org/book/
  - Head First Programming, David Griffiths, Paul Barry

Further Study

- Online Classes
  - **https://www.codecademy.com/tracks/python**
  - **https://www.edx.org/course?search_query=python**
  - https://www.coursera.org/courses?query=python
  - https://www.udacity.com/course/intro-to-computer-science--cs101
  - https://developers.google.com/edu/python/
- Deeper Books
  - **Learning Python, Mark Lutz**
  - Programming Python, Mark Lutz
  - Python Cookbook, David Beazley, Brian K. Jones
  - Python 3 Object-oriented Programming, Dusty Phillips
  - https://automatetheboringstuff.com

Conclusion

Your Time is Now

“There is a tide in the affairs of men.
Which, taken at the flood, leads on to fortune;
…
On such a full sea are we now afloat…”

Julius Caesar, Act-IV, Scene-III, Lines 218-224

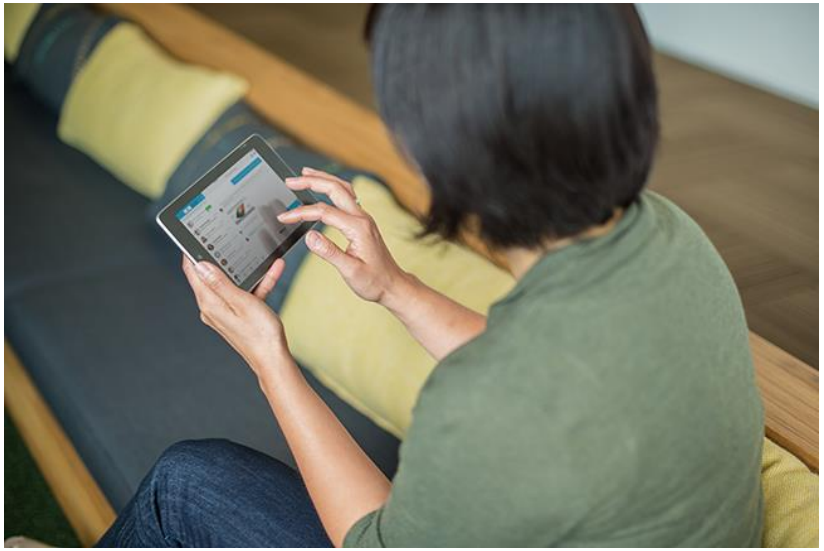
What will your position be?

“Whatever the mind of a man can conceive and believe it can achieve.”

Napoleon Hill
1883-1970

Complete Your Online Session Evaluation

- Please complete your Online Session Evaluations after each session
- Complete 4 Session Evaluations & the Overall Conference Evaluation (available from Thursday) to receive your Cisco Live T-shirt
- All surveys can be completed via the Cisco Live Mobile App or the Communication Stations



Don’t forget: Cisco Live sessions will be available for viewing on-demand after the event at CiscoLive.com/Online

Continue Your Education

- Demos in the Cisco campus
- Walk-in Self-Paced Labs
  - LABACI-1011 & LABACI-1012
- Lunch & Learn
- Meet the Engineer 1:1 meetings
- Related sessions
  - LTRCRT-2225, BRKCLD-1003, DEVNET-1001, DEVNET-1080, DEVNET-1060, DEVNET-1212, DEVNET-1219, DEVNET-1002, DEVNET-2041, DEVNET-1068, DEVNET-1082, LABACI-1012, DEVNET-1042, DEVNET-1040, & DEVNET-1041

Thank You

Cisco live!
February 20 - 24, 2017 · Berlin

cisco

Your Time Is Now

Appendix
For further study

Strings

String Basics

Metacharacters and escape sequences

- Escape Sequences are special characters
- Python will do something with these characters by default
  - `S = 'Algoryhme by Radia Perlman\nI think that I shall never see\nA graph more lovely than a tree.\n…'`
- Raw strings suppress Escapes
  - S = “C:\test\new.txt” vs S = r”C:\test\new.txt”

Escape	Description
\n	Newline
\t	Horizontal tab
\\, \’, \”, …	Print next character

String Expression Formatting

Use Case 1 - Substitution

- Used to insert one or multiple variables into a string
  - Used instead of concatenation or when you don't know variables
  - %typecode within string
  - String & tuple of substitutions separated by % sign
- `print 'TAC case %s has %d bug(s) identified.' % (TACcaseNumber, 1)`

Type Code (subset)	Description
s	String
d	decimal
f	Floating-point decimal

String Expression Formatting

Use Case 2 – Format a Floating Point Number

- Used to format the conversion of numbers to strings (and at times with stings)
- *%[(keyname)][flags][width][**.precision**]typecode*
  - **number of digits after a decimal**

```
>>> t = 1.23456789
>>> print ' this is a test of %.2f as a number' % t
 this is a test of 1.23 as a number
```

https://docs.python.org/2/library/stdtypes.html#string-formatting

String Method Formatting

Use Case 1 - Substitution

- Formats a string based on a method call
  - Variables in string defined by { }
  - Can be referenced by index, keyword, both of the former, or relative position

```
>>> inventory = 'Device type is {0}, hostname is {1}, loopback0 is {2}'
>>> inventory.format('switch', 'switch10', '1.1.1.1')
'Device type is switch, hostname is switch10, loopback0 is 1.1.1.1'
>>>
>>>
>>> inventory = 'Device type is {device}, hostname is {hostname}, loopback0 is {ip}'
>>> inventory.format(device='switch', hostname='switch10', ip='1.1.1.1')
'Device type is switch, hostname is switch10, loopback0 is 1.1.1.1'
```

String Method Formatting

Use case 2 – Formatting a Floating Point Number

- Advanced formatting can alter way string appears
- *{fieldname component !conversionflag **:formatspec**}*
- *[[fill]align][sign][#][0][width][,]**[.precision]**[typecode]*

```
>>>
>>> s = 'TAC case {0} took {1:.2f} days to find root cause.'
>>> s.format('688601432', 1.23456)
'TAC case 688601432 took 1.23 days to find root cause.'
>>>
```

https://docs.python.org/2/library/string.html#formatstrings

ShowVersion Remembered

AS Displayed (CLI Output)

```
…
Build Information:
 Built By      : ahoang
 Built On      : Mon Dec 14 02:53:00
PST 2015
 Build Host    : iox-lnx-010
 Workspace     : /auto/iox-lnx-010-
san1/production/5.2.5.37I.SIT_IMAGE/
all/workspace
 Version       : 5.2.5.37I
 Location      :
…
```

AS Stored (as a string)

```
…\n Workspace     : /auto/iox-lnx-
010-
san1/production/5.2.5.37I.SIT_IMAGE/
all/workspace\n Version       :
5.2.5.37I\n Location      :
/opt/cisco/XR/packages…
```

String.find(*substring*) method

- This method takes a substring as an argument
- Useful when wanting to find a particular substring within a string

```
>>> VersionString = 'Version      : '

>>> len(VersionString)

15

>>> ShowVersion.find(VersionString)

319

>>> ShowVersion.find('\n', 319 +
len(VersionString))

343

>>> Version = ShowVersion[319 +
len(VersionString):343]

>>> Version

'5.2.5.37I'
```

Lists, Dictionaries, & Tuples

List Methods

Using a comprehension to see methods without underscores

```
>>> dir(L)

['__add__', '__class__', '__contains__', '__delattr__', '__delitem__',
'__delslice__', '__doc__', '__eq__', '__format__', '__ge__',
'__getattribute__', '__getitem__', '__getslice__', '__gt__', '__hash__',
'__iadd__', '__imul__', '__init__', '__iter__', '__le__', '__len__', '__lt__',
'__mul__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__',
'__reversed__', '__rmul__', '__setattr__', '__setitem__', '__setslice__',
'__sizeof__', '__str__', '__subclasshook__', 'append', 'count', 'extend',
'index', 'insert', 'pop', 'remove', 'reverse', 'sort']

>>> [x for x in dir(L) if '_' not it x]

['append', 'count', 'extend', 'index', 'insert', 'pop', 'remove', 'reverse',
'sort']
```

Dictionary Creation from Zip()

- Use zip(collection1, collection2) to “zip” keys to values

```
>>>
>>> KeyList = ('one', 'two', 'three')
>>> ValueList = (1, 2, 3)
>>> D = dict(zip(KeyList, ValueList))
>>> D
{'three': 3, 'two': 2, 'one': 1}
>>>
```

Dictionary Comprehensions

- Use zip(collection1, collection2) to “zip” keys to values

```
>>>
>>> range(5)
[0, 1, 2, 3, 4]
>>> D = {X: X ^ 2 for X in range(5)}
>>> D
{0: 2, 1: 3, 2: 0, 3: 1, 4: 6}
>>>
```

https://www.python.org/dev/peps/pep-0274/

Named Tuples & Ordered Dictionaries

- What if you want a immutable dictionary like structure to make code readability easier?...Named Tuples
- Named tuples allow a dictionary like naming of the tuple values
- What if you want your dictionary items to be deterministic in order? Ordered Dictionaries
- Ordered Dictionaries remember order in which keys are added
- Must import *collections* module for both structures

https://docs.python.org/2/library/collections.html#collections.namedtuple

Files

Files

- How you work with file really depends on what you are doing and type of file
- Files can be opened as objects
- You can then work with the objects to read and write data
- Many types supported, some have additional modules
  - *csv*
  - *openpyxl*
- If on windows use r'', or raw, text strings when opening
  - `file = Open(r'C:\test\next\file.txt', 'r')`
- Don't forget to close file
  - `file.close()`

Files Open(), .write(string), & .close()

```
>>> file = open(r'lumberjack.txt', 'w')

>>> file.write("I'm a lumberjack and I'm OK\nI sleep all night\nand I work all
day\n")

>>> file.close()
```

Files .read()

```
>>> file = open(r'lumberjack.txt', 'r')
>>> string = file.read()
>>> file.close()
>>> print string
I'm a lumberjack and I'm OK
I sleep all night
and I work all day
>>>
```

Common File Operations

Operation	Description
`File = open(r'C:\test\test.txt', 'w')`	Create an file for output; 'w' means write
`File = open(r'C:\test\test.txt', 'r')`	Create a file for input; 'r' means read
`String = file.read()`	Read entire file into string
`String = file.readline()`	Read next line (including \n) into string
`List = file.readlines()`	Read all lines of file into a list
`File.write('Router1 is %s') % 'down'`	Write a string to a file
`File.writelines(['one\n', 'two\n', 'three\n')`	Write a list of strings to a file
`File.close()`	Close a file

Saving Object to File

- What if you want to save a Python object to a file?
- This is called object serialization
- Used to store data for later or to transmit data to another system/application
- Two popular options
  - Pickle
  - JSON

Pickle

- Python Module
- Designed for Python
- Creates a Pickle Jar, a file, to store contents
- Can include an object's methods in pickle jar
- Used when only one Python script will use saved data or transfering data between Python applications

https://docs.python.org/2/library/pickle.html

```
import pickle

RouterDict = { "router1":
"NCS6008", "router2": "ASR9001"
}

pickle.dump(RouterDict, open(
"routers.pkl", "wb" ) )

$ ls routers.pkl

routers.pkl

RouterDict2 = pickle.load(
open( "routers.pkl", "rb" ) )

RouterDict2

{ "router1": "NCS6008",
"router2": "ASR9001" }
```

JSON

- JavaScript Object Notation
- More generalized
- Stored as text file
- Cannot include Python code (methods)
- Used when sending receiving data from APIs or between Python and non-Python applications

https://docs.python.org/2/library/json.html

```
>>> import json

>>> D = {'hostname' : 'router1',
'loop0' : '1.1.1.1', 'device' :
'NCS6008'}

>>> file = open(r'object', 'w')

>>> file.write(json.dumps(D))

>>> file.close()

>>> file = open(r'object', 'r')

>>> D2 = file.read()

>>> D2 = json.loads(D2)

>>> D2

{u'device': u'NCS6008', u'hostname':
u'router1', u'loop0': u'1.1.1.1'}
```

Procedural Statements & Syntax

Semantics vs Syntax

- Syntactically & semantically incorrect
  - Grammatically
    - ?am eating rocket my
  - Programmatically
    - ’router’**D.keys()
- Syntactically correct and semantically incorrect
  - Grammatically
    - Rocket am eating sadness.
  - Programmatically
    - A = 5
    - B = ‘Cisco’
    - A + B = C

Semantics vs Syntax

- Syntactically incorrect and semantically correct
  - Grammatically
    - Eats the horse oats.
  - Programmatically
    - A = 5
    - B = 6
    - A B + = C
- Syntactically & semantically correct
  - Grammatically
    - The horse eats oats.
  - Programmatically
    - A, B = 5, 6
    - A + B = C

Basic Python Syntax

- All scripts in *.py text file
- No declaration of variables, dynamic typing
- No ; for end of line, end of line is end of statement
- No {} for blocks of code, indentation denotes blocks of code
- Compound statements have following format:

```
Header line:
    Nested statement block
```

- Can use ; to put multiple statements on one line

Assignment Syntax

- Assignments create variable names, references, and objects
- Variables must be assigned before being referenced
- Variable names must start with underscore or letter
- Special cases
  - _name, single underscore start, won't import with `from module import *`
  - __name, double underscore start, "mangled" name to make name unique
  - __name__, double underscore at start and end, system names

Indentation

- Indentation marks the start of a sub-block of code
- End-of-indentation marks the end of a sub-block of code
- Sub-blocks can be nested
- Standardize on whether spaces of tabs will be used

```
Main Block
Header-Line:
    Sub-block1
    Header-Line:
        Sub-block2
    Sub-block1
Main Block
```

Transition to Function Structured Code

Function Syntax

def *function_name*(*arguments*):

statement(s)

return *object*

def *function_name*(*arguments*):

statement(s)

yield *object*

```
def runtime(startTimeStamp, entity, units, divisor=1):
    endTimeStamp = datetime.datetime.now()
    runtime = _endTimeStamp - _startTimeStamp
    logger.info('It took %s %s to complete.' % (entity, runtime))
    if _divisor >1:
        logger.info('Average time for %i %s(s) was %s.' % (divisor, units,
                                                           (runtime/divisor)))
    return runtime
```

Variable Scope/Namespace

Finding a Person

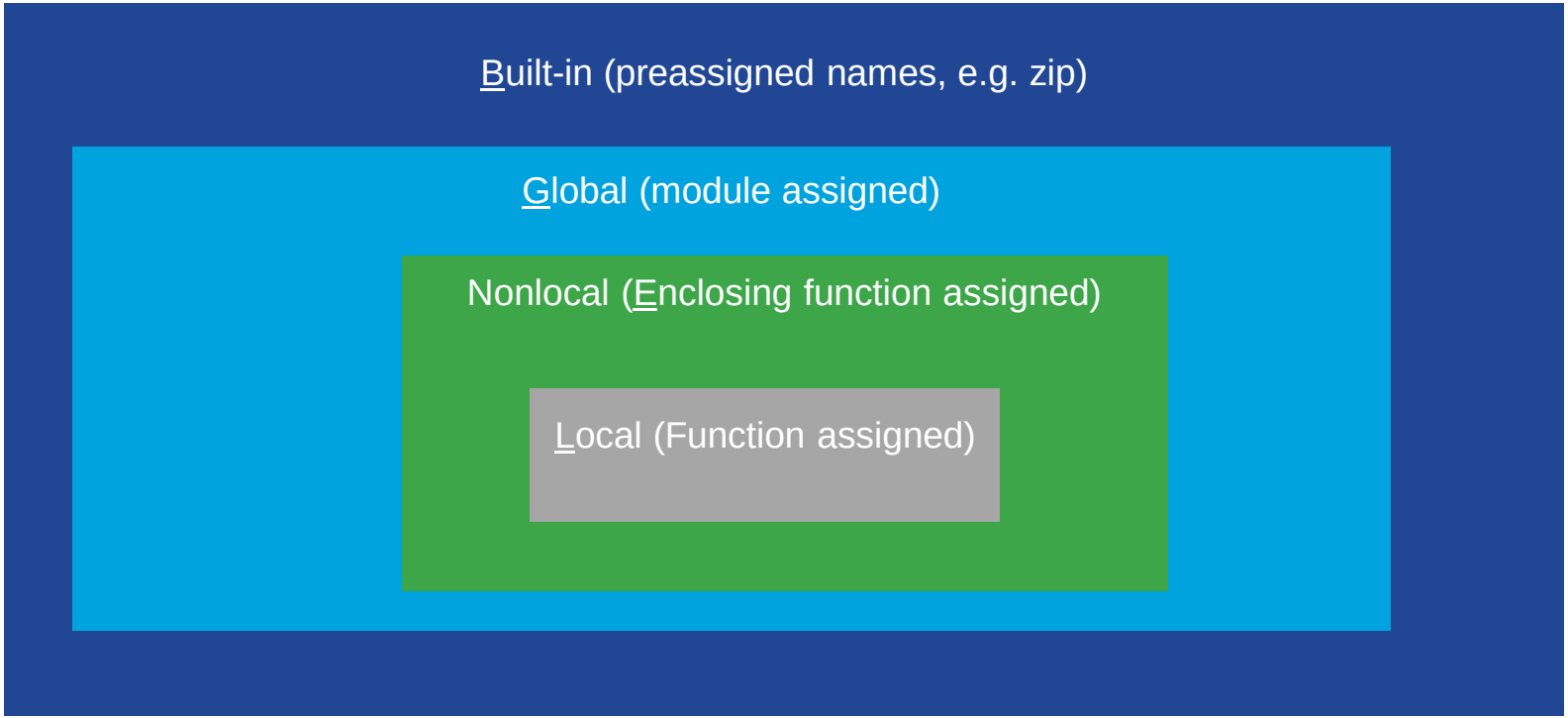
- The people example
  - People reside in houses
  - Houses reside in neighborhoods
  - Neighborhoods reside in cities
  - Cites reside in countries
- From housemate perspective, If you wanted to find their housemate
  - First you'd look in their house, if they weren't there…
  - You'd look in their neighborhood, if they weren't there…
  - You'd look in their city, if they weren't there…
  - You'd look in their country

Variable Scope/Namespace

Finding a Variable Name

- Remember variables create names and the names point to objects
- Just like people live in houses, names live in namespaces
- The name example
  - Local names reside in function namespace
  - Nonlocal names reside in enclosing function namespace
  - Global names reside in module namespace
  - Built-in names reside in Python source code namespace
- From function standpoint, if you wanted to find a particular name
  - First you'd look in the function/Local namespace, if it wasn't there…
  - You'd look in enclosing function/NonLocal namespace, if it wasn't there…
  - You'd look in their city, if it wasn't there…
  - You'd look in their country

Namespace Diagram



Function Scope Points

- Module is global scope
  - All variables so far have been global
- Names in a function are local to that function only
  - This alleviates name collisions
- Names in a Function are local unless declared global or nonlocal
  - You can use higher namespaces in read-only fashion
  - If you want to modify higher namespace variable you have to declare it global or nonlocal
- Name resolution moves up to find a name, not down!
  - LEGB rule
  - Local-Enclosing-Global-BuiltIn

Namespace Example

```
>>>
>>> cisco = 'global variable'
>>> cisco
'global variable'
>>> def func1():
...     cisco = 'local variable'
...     print cisco
...
>>> func1()
```

Namespace Example

```
>>> def func1():
...     cisco = 'local variable'
...     print cisco
...
>>> func1()
local variable
>>> cisco
```

Namespace Example

```
>>> def func1():
...     cisco = 'local variable'
...     print cisco
...
>>> func1()
local variable
>>> cisco
'global variable'
>>>
```

Namespace Example

```
>>>
>>> cisco = 'global variable'
>>> cisco
'global variable'
>>> def func1():
...     global cisco
...     cisco = 'local variable'
...     print cisco
...
>>> func1()
```

Namespace Example

```
>>> def func1():
...     global cisco
...     cisco = 'local variable'
...     print cisco
...
>>> func1()
local variable
>>> cisco
```

Namespace Example

```
>>> def func1():
...     global cisco
...     cisco = 'local variable'
...     print cisco
...
>>> func1()
local variable
>>> cisco
'local variable'
>>>
```

Namespace Example

```
argument = 'Today is a good day!'
number = 5
word = 'cisco'


def func1(argument):
 if argument == 1: :
    print 'Hello World!'


def func2(argument):
    if argument == 1:
    print 'Goodbye World!'
```

Namespace Example

```
print argument

func1(argument)

func1(number)

func2(word)

func1(1)
```

Function(*arguments*) Basics

- Arguments are passed into function inside of parenthesis
- Can be read into function by position
- Can be read into function by keyword
  - Key=value format
  - Position not important
- In Key=value, value can be default value if no argument is passed in
- If both positional and keyword arguments are passed in, positional must come first

Cisco live!
February 20 - 24, 2017 · Berlin

cisco

Your Time Is Now