Python

How You Can Use and Write Python Programs
(Part I)

Intro

I’m a Systems Programmer with IT/NSS.

- I’ve been programming over 30 years
- Recent convert to Python
- Needed to use Python because Google APIs are written for Python
- I’m new to Python. You can probably stump me
- …. unless I can use Google.

What is Python?

Python is an object oriented interpreted language with a large number of features

- … that you can program in a “C”-like manner, skipping object oriented components
- … many of the more confusing features can be skipped.
- … Python can be converted to a “C” program and compiled.
- … Python can be converted to a Java program

What is Python?

When Python modules are converted, the results are stored as .pyc files.

Modules are an advanced topic. I mention it in case you install a package. You will want write access so that the .pyc files can be created.

Python will still run if the .pyc files can not be created. Just a little more slowly.

What is Python?

So, it is a little hard to put Python in a box. It can be used for many things.

Its role in high performance computing is to allow simple use of optimized libraries that implement mathematical routines in an efficient manner.

Python is reasonably fast, but it is not a number crunching language by itself. Extra packages are required such as NumPy.

What is Python

Python comes with many operating systems or can easily be installed.

- It is native to MAC OS/X
- It is likely installed on any Linux system, or can be easily added using package management commands
- The Python website contains Windows distributions on the main site: https://www.python.org/downloads/windows/ Googling “python windows” is as hard as it gets.

This means you can work in Python right on your desktop or laptop.

Python Resources:

- Python - http://python.org/
- PEPs - https://www.python.org/dev/peps/

(PEPs are Python Enhancement Proposals but are often used to document various Python issues.)

PEP 8 is required reading if you plan to write Python programs. Less so if you just need to read or make minor changes.

- The last slide contains more on help. Left to end so you will remember!

Python Resources: Zen

```
>>> import this
```

The Zen of Python, by Tim Peters

- Beautiful is better than ugly.
- Explicit is better than implicit.
- Simple is better than complex.
- Complex is better than complicated.
- Flat is better than nested.
- Sparse is better than dense.
- Readability counts.
- Special cases aren't special enough to break the rules.
- Although practicality beats purity.
- Errors should never pass silently.
- Unless explicitly silenced.
- In the face of ambiguity, refuse the temptation to guess.
- There should be one-- and preferably only one --obvious way to do it.
- Although that way may not be obvious at first unless you're Dutch.
- Now is better than never.
- Although never is often better than *right* now.
- If the implementation is hard to explain, it's a bad idea.
- If the implementation is easy to explain, it may be a good idea.
- Namespaces are one honking great idea -- let's do more of those!

Python Source Code

Python looks pretty normal EXCEPT

It is the only language that I know where indent levels actually created the block structure. For instance:

In (Modern) C, the classic “Hello, World!” Program is:

```
#include <stdio.h>
int main( int argc, char ** argv )
{
     int i;
     printf( "Hello, World!\n" );
     for( i = 0; i < argc; i++ ) {
          printf( "    Arg %d: %s\n", i, argv[i] );
    }

    return( 0 );
}
<save>
doc => gcc a.c
doc => ./a.out  a
Hello, World!
    Arg 0: ./a.out
    Arg 1: a
```

In Python:

```
#!/usr/bin/python
import sys

print( "Hello World" )
for (i, arg ) in enumerate( sys.argv ):
    print( "    Arg %d: %s" % (i, arg) )
<save>
doc => ./a.py 12 bc
Hello World
    Arg 0: ./a.py
    Arg 1: 12
    Arg 2: bc
```

Python Source Code

Important Points

- You need to either have the "`#!/usr/bin/python`" line at the start of the script or just type "*python a.py*" each time you want to run the script.

  … The operating system needs to know to run Python
- You will probably need to make your script executable by using a command like this:

chmod a+x a.py

Python Source Code

In Python, a “main” routine is not needed. Python executes your script top to bottom.

Python Source Code

```
for (i, arg ) in enumerate( sys.argv ):
    print( "    Arg %d: %s" % (i, arg) )
```

The indentation is super important.

- In C, I could have written the entire program on one line.
- In Python, I could too. But, no one does that.
- In Python, the indent prior to `print` is what associates the `print` with the `for`.

Most (all?) Python statements that start a block of code end with a colon (:)

Python Source Code

- So, indenting is important.
- Line ends matter

So, what to do for lines that extend? Afterall, we KNOW we are supposed to keep lines to 79 characters or less.

Lines automatically continue if there is an open parenthesis, brace or bracket.

-
  ```
  MyLongClassName.ThatHasALong.RoutineName(
      Argument1, Argument2 )
  ```

-
  ```
  a = [ ‘List item1’,
            ‘List item2’ ]
  ```

-
  ```
  a = { ‘key1’: ‘value1,
            ‘key2’: ‘value2’ }
  ```

You can also end a line with a “\”, but one of the above forms is far preferred.

Python Source Code

- So, indenting is important.
- Line ends matter

Remember, you can always just add in parens if you need to continue the line

```
if (ThisLongVariable > SomeOtherLongVariable &&
     AndThisOneToo < TheLastOne):
     print( “It’s good.” )
```

VS

```
if ThisLongVariable > SomeOtherLongVariable && AndThisOneToo < TheLastOne:
    print( “Yuck!” )
```

Python Source Code

- This simple fact that lines can be continued when any brace like pattern is open is what finally made me love Python.
- Once you get used to not typing braces and semicolons, you will not want to go back!
- And very, very rarely should you use the line continuation character “\” In over 50,000 lines, I have never used it.

Python: Unicode

If you have not dealt with supporting Unicode on your terminal emulator, now is good time to do so:

```
$ python
>>> a=u'\u2284'
>>> print a
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
UnicodeEncodeError: 'ascii' codec can't encode character u'\u2284' in position 0: ordinal
not in range(128)
$ export LANG=en_US.UTF-8
$ python
>>> a=u'\u2284'
>>> print a
⊄
```

So, just set LANG, and make your terminal emulator handle UTF8 and be done with it. Dealing with the `UnicodeEncodeError` will waste endless amounts of time.

Python Editing

Because editing in Python is based on indents

… and indents of level 8 - a full tab stop - are rather long

… and indents of level 2 are horrid to read

… and only certain people use indents of five (you know who you are!)

Setting up your editor to convert a tab key to four spaces is quite useful.

Python Editing

In my ~/.exrc file for vi, I have

```
set shiftwidth=4
set tabstop=8
set expandtab
set softtabstop=4
```

This causes any tab key press to be converted to align to column divisible by 4, and to use spaces. This will not convert existing tabs to align on four, though. Converting code in that manner, unless it is yours, can cause anger!

There are many clever ways to do this so it only works for Python files, for instance. Google your options...

Python Editing

What about Emacs?

….I’m sure Emacs can handle four character “tabs”.

Python Editing

Converting existing code aligned to four columns, but with a mix of tabs and spaces can be fixed by:

```
#!/bin/bash

for f in `find . -name ”*.py”`; do
    expand ${f} > ${f}-new.py
    mv ${f}-new.py ${f}
done
```

(Fix the double quotes if you copy and paste that)

Python is Objects!

- Every “thing” in Python is an Object.
- Names in Python refer to an Object.
- Names have no type.
- Names have no specific value.

Python is Objects!

- Every “thing” in Python is an Object.

What does this mean?

Is ‘’1’’ an object?

… yes!

```
a = 1
```

This statement means exactly that name “a” refers to the object “1”. The object “1” is of type “int”

Python is Objects!

- Every “thing” in Python is an Object.

What does this mean?

Is ‘’1’’ an object?

… yes!

```
a = 1
```

The constructor for type ‘’int’’ was automatically called, and the value 1 was passed.

Python is Objects!

- Every “thing” in Python is an Object.

What does this mean?

Is “1” an object?

… yes!

```
a = 1
```

Is it vital that the concept of “a” referring to the object “1” become the way you think. “a” IS NOT OF TYPE “int”. “a” refers to an object of type “int”.

Python is Objects!

- Every “thing” in Python is an Object.

What does this mean?

Is ‘’1’’ an object?

… yes!

```
a = int(1)
```

Python is Objects!

- Every “thing” in Python is an Object.
- Names in Python refer to an Object.
- Names have no type.
- Names have no specific value.

So, now we see that Names refer to objects.

- Objects have a type
- Objects have a value

Python is Objects!

Some standard string assignments. For each of these, the name “a” refers to a different Object.

- `a = “A string”`
- `a = 1.1`
- `a = ‘Single quotes can be used same as doubles’`
- `a = ‘Standard escapes\ncan be used and printed’`
- `a = r”””A raw string with no escape processing\nNope”””`
- `a = “””Three double quotes start a multi-line document string.”””`
- `a = “””SELECT * FROM mytable””” # useful for writing multi-line SQL strings.`

-
  ```
  a = ( “Adjacent strings “ “concatentate.  Nice for “
      “writing long lines and not line wrapping.” )
  ```

Python: A few Special Objects

- `None` - means the name has no object - or perhaps the default object of `None`.
- `True` - Boolean “true”
- `False` - Boolean “false”

Python Operators

Python supports standard operators.

I am going to be very brief here because you can look these up online if need be.

- `+,-,*,/` etc work as expected.
- Assignment: `=`
- Quick forms: `+=, -=, *=, /=` etc.
- C style `++` and `--` not supported.
- `and`, `or`, `not` are the logical operators (NOT `&&, ||`, etc)

Python Standard Objects

- Integers: `1`
- Floats: `1.1, 1e10`
- Strings: `“string”, ‘string’, “””Doc String”””, r”””Raw String”””`
- Lists: `[ 1, ‘a’, 2, [ 1, 2 ] ]`
- Sets: `{1,2,3,3}` (`set([1, 2, 3])`, of course)
- Dicts: `{ ‘1’: 1, 1: 1, ‘A long key’: 2 }`
- Tuples: `(1,)`

Python Standard Objects

Objects have an important property:

Mutability

Mutability means “Can we change the object”

Of course we can change what object a name refers to. But, can we change the actual object?

Python Standard Objects

Objects have an important property:

Mutability

I think we can all agree that changing the object “1” to be a “2” would be a bad thing.

All of a sudden, all objects of value “1” would change to a “2”?

Python Standard Objects

Objects have an important property:

Mutability

Remember, changing the value of the object IS NOT:

```
a=1
a=2
```

That just changes which object “a” refers to. A “1” or a “2”.

Python Standard Objects

- Integers: `1 –` **Immutable**
- Floats: `1.1, 1e10 –` **Immutable**
- Strings: `“string”, ‘string’, “””Doc String”””, r”””Raw String””” –` **Immutable**
- Lists: `[ 1, ‘a’, 2, [ 1, 2 ] ] –` **Mutable**
- Sets: `{1,2,3,3}` (`set([1, 2, 3])`, of course) – **Mutable**
- Dicts: `{ ‘1’: 1, 1: 1, ‘A long key’: 2 } –` **Mutable**
- Tuples: `(1,) –` **Immutable**

Python Standard Objects

- Integers, floats, complex: `1` - **Immutable**

Let’s just skip discussing changing all “1” to a “2”. I think we can agree that would be a terrible feature. Unless you are a systems programmer and it is April the 1st. Yes, yes we might.

`int` can be arbitrarily long.

```
>>> a=int(1111111111111111111111111111111)
>>> a
1111111111111111111111111111111L
```

Python Standard Objects

- Strings: `“aaa” -` **Immutable**

It’s interesting that you can not change a string in Python. You can cut pieces out and make a new string, but you can not change a string object. I am going to use “slices” to show this. Just go with it…

```
>>> a="1234"
>>> a[0:1] = "a"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support
item assignment
```

Python Standard Objects

- Strings: `“aaa”` – **Immutable**

```
>>> a="1234"
>>> a="b"+a[1:]
>>> a
'b234'
```

So, that took the constant string “`b`” plus the characters in the object referenced by name a after the first character and made a new string object and assigned that object to the name “`a`”. Pedantic, aren’t I?

Python Standard Objects

- Strings: `“aaa”` - **Immutable**

Strings allow access to characters via indexes.

```
>>> a="abcdefg"
>>> a[:1]
'a'
>>> a[-1:]
'g'
>>> a[-2:-1]
'f'
```

This technique is called “taking a slice.” Slices will be covered as a separate topic.

Strings are also “iterable” - another topic we will cover.

Python Standard Objects

- Tuples: `(1,2,3,4) -` **Immutable**

Tuples are a kind of cool feature. They can be referenced like lists (arrays), but can not be changed. Slices also work on tuples.

That they are immutable gives them the advantage that they can be used as keys in `dict`s.

Tuples are often used as a way of passing multiple values in a single argument.

We will see them in string formatting and databases. Probably math libraries.

Python Standard Objects

- Tuples: `(1,2,3,4)` - **Immutable**

```
>>> a = ( 'mike', {} )
>>> a[1][ 'gender' ] = 'M'
>>> print a
('mike', {'gender': 'M'})
```

Just playing. Kinda neat. Tuples can not change, but objects in the tuple can.

Tuples are also sliceable and iterable. We will cover this later on.

Python Standard Objects

- Tuples: `(1,2,3,4)` - **Immutable**

```
>>> a = ( 'mike', {} )
>>> a[1][ 'gender' ] = 'M'
>>> print a
('mike', {'gender': 'M'})
>>> b={}
>>> b[a] = 1
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'dict'
```

But, still can not have mutable as a hash key!

Python Standard Objects

- Lists: `[ 0, 1, ‘a’, “a string”, 4, … ] -` **Mutable**

Lists are the basic “array” feature that all languages have.

- Basic constructor: `a=[]` or `list()`
- Lists have a length. You can not assign beyond the length.

```
>>> a=[]
>>> len(a)
0
>>> a[0]=1
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list assignment index out of range
```

- `a=[ 0 for i in xrange( 0, 10 ) ]`

```
a[5]=”A test”
```

Python Standard Objects

- Lists: `[ 0, 1, ‘a’, “a string”, 4, … ] -` **Mutable**

Lists are the basic “array” feature that all languages have.

- Some other ways to work with arrays:

```
>>> a=[]
>>> a.append(1)
>>> len(a)
1
>>> a.extend( [2,3] )  # extend does not return a result!
>>> a
[ 1, 2, 3 ]
```

- We will return to “comprehensions” later on (`[... for … xrange…]`)
- Zero based indexing, of course.
- Groups of elements can be manipulated via slices.

Python Standard Objects

- dicts: `{ ‘a key’: ‘a value’ }` – **Mutable**

Dicts are often called “hashes” in other languages.

- Items can be indexed by just about anything.
- EXCEPT: the index or key object must be **Immutable.**
- The associated value can be either mutable or immutable.

Python Standard Objects

- dicts: `{ ‘a key’: ‘a value’ }` - **Mutable**

Dicts are often called “hashes” in other languages.

- The values can be other dicts or lists or ints or function references or...

```
>>> a=dict()
>>> b={}
>>> a,b
({}, {})
>>> a[ 'a key' ] = 1
>>> b[ 'The big key' ] = a
>>> b
{'The big key': {'a key': 1}}
```

Python Standard Objects

- dicts: `{ ‘a key’: ‘a value’ }` - **Mutable**

Dicts are often called “hashes” in other languages.

- The values can be other dicts or lists or ints or function references or...

```
>>> a = { 'a': 1,
... 'b': 2,
... 'c': 3 }
>>> a
{'a': 1, 'c': 3, 'b': 2}
```

Python Standard Objects

As you might have noticed, `strings`, `tuples`, `lists`, and `dicts` can all use the `[...]` to select items from the object. Additionally, if the object is mutable, the brackets are used to select what portion of the object to change or add.

Does this make life confusing? What TYPE OF OBJECT DO I HAVE???

Python Standard Objects

As you might have noticed, `strings`, `tuples`, `lists`, and `dicts` can all use the `[...]` to select items from the object. Additionally, if the object is mutable, the brackets are used to select what portion of the object to change or add.

Does this make life confusing? What TYPE OF OBJECT DO I HAVE???

Answer: Why do you care? If whatever your code is doing would work equally well on a list or a dict, why does it matter what object was passed?

And if does matter, then the code will fail.

Generally speaking, document what you expect, and just fail if that is not what was passed.

A Word About Booleans

We have seen True and False are valid booleans.

In Python, it is correct and a common form to do boolean tests based on empty objects or 0/1.

```
>>> a = {}
>>> if not a:
...    print( ‘a is empty’ )
a is empty
```

Many languages do not encourage this style. Python does.

A Word About Booleans

Note that an empty object is not the same as `None`.

```
>>> a = {}
>>> if not a:
...     print( 'a is empty' )
...
a is empty
>>> if a is None:
...     print( 'a is empty' )
...
>>> a=None
>>> if a is None:
...     print( 'a is none' )
...
a is none
```

A Word About Booleans

- Use `is None` if you need to check for a name without a useful object
- Use a boolean test without a comparison to check for emptiness or 0/1.
- Do not use `=` with `None`.
- Avoid comparing directly to `True` or `False`.

Python, oh Python

A brief pause here.

We have seen that: `a=1` creates a name “a” that refers to the object integer one.

Many other statements in Python are just fancy ways of assigning specialized objects to names.

Python could have a lot fewer statement types if the equals sign was used for these assignments. Of course, the code would be unreadable.

So, let’s push on...

Python Functions!

Now, we are getting somewhere. Functions are the basic building blocks of any language.

```
#!/usr/bin/python
def a( a1, b1 ):
    """
    What this function Does.

    Args:
        a1 - What is a1 for?
        b1 - What is b2 for?
    Returns:
        What the return values is/are.
    """

    return(a1+b1)

print( "Running function a(2,3): %s" % a(2,3) )
5
```

Python Functions!

Now, we are getting somewhere. Functions are the basic building blocks of any language.

```
>>> def a( a1, b1 ):
...     """
...     Doc string.
...     """
...     return(a1+b1)
...
>>> a(2,3)
5
```

⇐ `def` defines a function with arguments `a1` and `b1`

⇐ A documentation string to help other programmers.

Python Functions!

Now, we are getting somewhere. Functions are the basic building blocks of any language.

```
>>> def a( a1, b1 ):
...     """
...     Doc string.
...     """
...     return(a1+b1)
...
>>> a(2,3)
5
```

⇐ `def` defines a function with arguments `a1` and `b1`

⇐ A documentation string to help other programmers.

⇐ The code that executes. Returns a simple sum of two objects that can be added.

Of course, code that is part of the routine is indented under the "`def`."

Python Functions!

Now, we are getting somewhere. Functions are the basic building blocks of any language.

```
>>> def a( a1, b1 ):
...     """
...     Doc string.
...     """
...     return(a1+b1)
...
>>> a(2,3)
5
```

⇐ `def` defines a function with arguments `a1` and `b1`

⇐ A documentation string to help other programmers.

⇐ The code that executes. Returns a simple sum of two objects that can be added.

⇐ Here, we call the function with two arguments

Python Functions!

Now, we are getting somewhere. Functions are the basic building blocks of any language.

```
>>> def a( a1, b1 ):
...     """
...     Doc string.
...     """
...     return(a1+b1)
...
>>>a("Huh"," neat")
'Huh neat'
```

⇐ `def` defines a function with arguments `a1` and `b1`

⇐ A documentation string to help other programmers.

⇐ The code that executes. Returns a simple sum of two objects that can be added.

⇐ Works for any object type that supports the "+" operator

Python Functions

OK, now that we have seen them, they are not that surprising.

What does `def` do?

- `def` assigns a python code object to a name.
- `def` does not process the code object during assignment.
- The code object does have to syntactically correct, of course.

Python Functions

A big source of confusion is: for arguments passed to a function, when will the object I pass be changed by the function.

```
a=1
b(a)
```

What is the value of “`a`” after the call? Could the function have changed “`a`”?

Python Functions

A big source of confusion is: for arguments passed to a function, when will the object I pass be changed by the function.

```
a=1
b(a)
```

Let’s go back and think this through. The call to function `b` is just going to assign the object referenced by name “`a`” to the first argument.

AND: `int`s are **IMMUTABLE**!

Therefore: the object referenced by name “`a`” can not change.

Python Functions

A big source of confusion is: for arguments passed to a function, when will the object I pass be changed by the function.

```
a=”My string”
b(a)
```

The same holds true for a string object passed to a function. Recall that strings are immutable.

Python Functions

A big source of confusion is: for arguments passed to a funtion, when can the object I pass be changed by the function.

```
a={ ‘my key’: 1, ‘key two’: “value 2” }
b(a)
```

Here, the name “`a`” refers to a `dict`. `Dict`s are mutable. Can the call to routine “`b`” change the object referred to by the name “`a`”?

You bet! Because `dicts` are mutable.

Python Functions

A big source of confusion is: for arguments passed to a function, when can the object I pass be changed by the function.

```
a=(1,)
b(a)
```

Here, the name “`a`” refers to a `tuple`. `Tuples` are immutable. Can the call to routine “`b`” change the object referred to by the name “`a`”?

You bet NOT! Because `tuples` are immutable.

Python Functions

A big source of confusion is: for arguments passed to a function, when can the object I pass be changed by the function.

```
a=(1,)
b(a)
```

So, a programmer with experience in other languages would ask “Is Python pass by value or pass by reference?”

I would say “pass by reference”, but it does not matter. What does matter is if the object is mutable or immutable.

Python Functions

A big source of confusion is: for arguments passed to a function, when can the object I pass be changed by the function.

```
a={‘a’: 1 }
b(a)
```

It is important to understand that the object passed via name “`a`” will remain bound to name “`a`” after the call. Whether the object can change depends entirely on if it is mutable or not. The dict referred to by “`a`” can change during the call, but “`a`” will still refer to the same dict after the call.

Python Functions

So, is Python call by reference or call by value?

Answer: Python is better than that and does not need to answer such mundane questions.

But, I hope you have a solid grasp of when arguments passed to a function can be changed by the caller.

Python Functions

Some quick notes:

- A routine can be called with named arguments. Referring the prior example:

```
a( b1=2, a1=3)
```

- Keyword arguments can make code with many arguments clearer.
- A routine can be defined with default values:

```
def a( a1=”Why, Hello”, b1=”There” )
```

- Generally, a function call must supply values for all expected arguments
- … unless the function definition supplies a default.
- `def( a1=None, b1=None)` can be a useful way to define a function with lots of optional arguments with no particular default.

Python Functions

Some quick notes:

- The call formats `a( *args )` and `a( **kwargs )` are special ways to pass lists of arguments constructed at runtime, and dicts of keyword arguments constructed at runtime.
- Can be combined as `a( *args, **kwargs )`.
- Further discussion later on. But, I thought I would mention it because it confused the heck out of me at first and it is a hard thing to “Google”.

Python Statements

```
simple_stmt ::=  expression_stmt
               | assert_stmt
               | assignment_stmt
               | augmented_assignment_stmt
               | pass_stmt
               | del_stmt
               | print_stmt
               | return_stmt
               | yield_stmt
               | raise_stmt
               | break_stmt
               | continue_stmt
               | import_stmt
               | global_stmt
               | exec_stmt
```

Oh, cool, cut-n-paste from the python web pages works!

Python Statements

It would be pointless to just do slide after slide of expression statements.

- We've seen simple assignment: `a = 1`
- Print is in fact a statement in Python 2.7, not a function call. Best to write in Python 3.3 format and use it as a function call. At top of each module, do:

  ```
  from __future__ import print_function
  ```

  This makes the 2.7 print act like the 3.0 function. Saves you the trouble when porting to 3.0.

Python Statements

It would be pointless to just do slide after slide of expression statements.

- `return` returns values when a function ends.
- `return` can return multiple values which are returned as a `tuple`.
- Recall that you can access `tuple` values using simple 0 based indexing.
- Slices, which we have not directly discussed also work.

Python Statements

It would be pointless to just do slide after slide of expression statements.

- `del` deletes the name, and hence a reference to the object.
- If there are no other references to the object, then the object itself is deleted.
- We have not covered garbage collection - this is what happens to objects that have no references. They can't be used, so they are removed from memory.

Python Statements

It would be pointless to just do slide after slide of expression statements.

- Sometimes an expression is required, but you have nothing to say. Just as in Bridge, when you have nothing to say, say “`pass`”.

Python Statements

It would be pointless to just do slide after slide of expression statements.

- `break` ends the current compound statement. Usually breaks out of a loop.
- `continue` returns to the top of the compound statement. Goes to the top of the current loop.
- Augmented assignments are just the `a+=1` sort of thing we've seen before.

The remaining simple statements are either beyond the scope of this talk or will be discussed in context.

Python Compound Statements:

```
compound_stmt ::=  if_stmt
                 | while_stmt
                 | for_stmt
                 | try_stmt
                 | with_stmt
                 | funcdef
                 | classdef
                 | decorated
suite        ::=  stmt_list NEWLINE | NEWLINE INDENT statement+ DEDENT
statement    ::=  stmt_list NEWLINE | compound_stmt
stmt_list    ::=  simple_stmt (";" simple_stmt)* [";"]
```

We will concentrate on if, while and for. Try is part of exceptions, which will be covered, and we have seen function definitions already. Classes and decorators? In four hours?

Python `if` Statement

If `<something>` is true, then do the statements indented under the `if` statement, `elif` check something else, `else` do this.

`<something>` is just an expression that can yield a `boolean` result. We've seen before how empty objects yield False, 0 is `False`, `None` is False, `False` is not False (just kidding… who's paying attention), etc.

Python `if` Statement

```
from __future__ import print_function

a = {}
b = [ 0, 1 ]

if a or b:
    print( “This will print because b is True” )

if a:
    print( “Not print” )
elif b:
    print( “Will print” )
else:
    print( “Will not be reached” )
```

Python `if` Statement

```
from __future__ import print_function

import secret_module

a = {}
b = [ 0, 1 ]

if (secret_module.does_something_with_the arg( a ) or
        secret_module.does_something_with_the_arg( b )):
    print( “This will print because b is True” )
```

Note that we wrapped the long conditional by using parenthesis to cause line continuation. Note the from future statement for print, and note the import statement that gets us “secret_module”

Python `for` Statement

```
for <variable> in <iterator>:
    statement-block
```

So, that is pretty simple looking. But, it is very powerful.

- *statement-block* is an indented group of statements. Either complex or simple.
- <*variable*> is used to receive the output of the iterator. *variable* is not actually accurate but suffices for now.
- <*iterator*> takes several chapters in a book to explain. We will start simple.

Python `for` Statement

```
for <variable> in <iterator>:
    statement-block
```

An *<iterator>* is anything that can be enumerated.

- Items in a `list` can be enumerated.
- Characters in a `string` can be enumerated.
- It makes the most sense to provide the keys from a `dict` if we are going to enumerate a `dict`.
- You can write your own iterators.
- There are lots of builtin iterators or generators (which can be used as iterators).

Python `for` Statement

```
for <variable> in <iterator>:
    statement-block
```

```
>>> for i in xrange( 0, 10 ):
...     print i
...
0
1
...
8
9
```

Note: there is a `range` function, too. Avoid it unless using V3 Python.

Python `for` Statement

`for` *<variable>* `in` *<iterator>*`:`
    statement-block

```
>>> for i in xrange( 0, 10 ):
...     print i
...
0
```

What is `xrange` doing? It is creating a `list` of integers starting from 0 for 10. But, it only adds an element to the output `list` when “asked” to by the `for` loop execute. The *iteration protocol* is an advanced topic.

Python `for` Statement

`for` *<variable>* `in` *<iterator>*`:`
    statement-block

```
>>> for i in xrange( 0, 1000000000000):    # Generates integers as needed
...     break                              # Only one int needed.
...
>>>                                        # Ran very quickly

>>> for i in range( 0, 1000000000000):     # Generates all ints in a list first
...     break
...
^C^Z                                       # Will run computer out of memory?
[1]+  Stopped                  python      # Certainly takes a while to run!
doc => kill %1                             # Don't run on a shared system!
```

Python `for` Statement

`for` *<variable>* `in` *<iterator>*`:`
    statement-block

- `xrange` is one of those optimizations that was obviously required even though it sort of breaks the beauty of the language.
- If you program in V2.7 Python, there are simple tools to convert V2.7 code to V3.x. They will convert `xrange` to `range`.
- `range` in V3.x Python generates results as needed, like V2.7 `xrange`.

Python `for` Statement

```
for <variable> in <iterator>:
    statement-block
```

```
>>> a=[1,2,3]
>>> for i in a:
...     print i
...
1
2
3
```

Simple enough, right? But what if you need the indexes of the items in `a`?

Python `for` Statement

`for` *<variable>* `in` *<iterator>*`:`
    statement-block

```
>>> for (i,v) in enumerate(a):
...     print "Item %d: %s" % ( i,v)
...
Item 0: 1
Item 1: 2
Item 2: 3
```

So, now we see "<*variable*>" is really "<*target-list*>". `Enumerate` creates a `tuple` with the *index* and the *value*, and unpacks to the <*target-list*>. And I snuck in print formatting!

Python `for` Statement

`for` *<target-list>* `in` *<iterator>*`:`
    statement-block

```
>>> for (i,v) in enumerate(a):
...     print "Item %d: %s" % ( i,v)
...
Item 0: 1
Item 1: 2
Item 2: 3
```

People often think *iterator functions* can return “any number of values.” That is not accurate. A function returns a tuple, whose values are unpacked and assigned to those listed in the `for`. Error occurs if improper number of targets listed.

Python `for` Statement

`for` *<target-list>* `in` *<iterator>*`:`
    statement-block

```
for (i,v) in enumerate(my_list):
    v2 = associated_list[i]
    target_list.append( v+v2 )
```

This example shows how to get past the "<*iterator*>" concept and to be able to do traditional indexed access on "arrays" (lists in Python)

Python `for` Statement

`for` *<target-list>* `in` *<iterator>*`:`
    statement-block

```
target_list=[ 0 for i in xrange(0,len(my_list)) ]
for (i,v) in enumerate(my_list):
    target_list[i] = v+associated_list[i]
```

The first statement is called a “*comprehension*”. It created a list of zeros that is the same length as *my_list*. I wonder if this code is faster than the previous slide? I would experiment if the lists are long or called very often. Otherwise, I would use the first form. It is less cluttered.

Python `for` Statement

```
for <target-list> in <iterator>:
    statement-block
```

```
my_dict = { ‘k1’: 1, ‘k2’: 2 }
for (k,v) in my_dict.items():
    print( “Key = %s, value = %s’ % ( k, v ) )
```

- For the sake of completeness, this is one way to iterate over a dict.
- Iteration of a dict directly returns the keys.
- It possible to get the values directly

Python `for` Statement

```
for <target-list> in <iterator>:
    statement-block
```

Do not forget that:

- `continue` returns to the top of the loop and executes the *<iterator>*.
- `break` ends the loop and starts execution after the *statement-block*.

Python `while` Statement

```
while <boolean>:
   statement-block
```

- This is much simpler. Runs until the *boolean* is `False`.
- I am going to skip examples here.

Python `else` on Iteration Statements

```
while <boolean>:
    statement-block
else:
    statement-block
```

The `else` executes if the loop terminates **without** using `break`.

```
>>> while False:
...     break
... else:
...     print 'else'
...
else
```

Python `else` on Iteration Statements

```
for i in xrange( 0, 10 ):
    if i == 5:
        break
else:
    print 'else'
```

- The `else` executes if the loop terminates **without** using `break`.
- We can see pretty clearly that the `else` **will not** execute in this example.

Python `else` on Iteration Statements

```
for i in xrange( 0, 10 ):
    if i == 5:
        break
else:
    print ‘else’
```

The `else` portion of the statement can often eliminate the need for flag variables in the loop that describe how the loop exited.

Python `else` on Iteration Statements

```
for v in my_list:
    if v == looking_for:
        <code for finding>
        break
else:
    <code when not found>
```

vs

```
found = False
for v in my_list:
    if v == looking_for:
        found = True
if found:
    <code for finding>
else:
    <code when not found>
```

Python `else` on Iteration Statements

```
for v in my_list:
    if v == looking_for:
        <code for finding>
else:
    <code when not found>
```

- Watch the indent level to see what statement the else is associated with.
- In this case, it is clearly associated with the for.
- Keep the code blocks pretty short so the indent can be easily read.
- Use functions to keep the code short.

Python: Exceptions

In Python, we often try to do things, and they do not work out. And that is OK because there is a try statement block:

```
try:
     a=1/0
except ZeroDivisionError:
     a=float('nan')
>>> a
nan
>>> b=1*a
>>> b
nan
```

Python: Exceptions

- Exceptions are going to be covered in part 2 of this discussion.
- Unlike other languages, they are a standard part of the control flow and are not at all slow.
- Other languages will start an entire new copy of the interpreter or other similar heavyweight operations. Not so in Python.

Help!

- Google
- The Python website is great. Note that you can select the version to get appropriate help.
- Within Python

```
>>> help( ‘for’ )
The "for" statement
*******************
The "for" statement is used to iterate over the elements of a sequence
(such as a string, tuple or list) or other iterable object:

   for_stmt ::= "for" target_list "in" expression_list ":" suite
                ["else" ":" suite]

The expression list is evaluated once; it should yield an iterable
```

Help!

This is so cool! I was thinking about “help” and an idea occurred to me.

“Could I put a help command in the middle of a block of code to see what the object associated with the name is and since it is a method, what are the arguments?”

I had never considered this until I wrote these slide. Check it out!

Help!

```
params = {}
userid = params[ 'userId' ] = (
    Mailbox( cmdparse.defdomain ).MakeMailbox( cmdparse.user ) )
g = gmail( userid )
help( g.auth.users().messages().insert )  ⇐ What arguments does insert take?
params[ 'internalDateSource' ] = 'dateHeader'
```

When I ran the program:

Help!

```
    g = gmail( userid )
    help( g.auth.users().messages().insert )
```
⇐ What arguments does insert take?

When I ran the program:

```
[mike@googleapps cmds]$ ../../ga --prod -vv
ga> user email migrate mike --mbox=a
Help on method method in module googleapiclient.discovery:

method(self, **kwargs) method of googleapiclient.discovery.Resource instance
    Directly inserts a message into only this user's mailbox similar to IMAP APPEND,
bypassing most scanning and classification. Does not send a message.

    Args:
      userId: string, The user's email address. The special value me can be used to
indicate the authenticated user. (required)
      body: object, The request body.
        The object takes the form of:

    { # An email message……….
```

Help!

- That’s huge because often you will not really know what object type you have. I am not at all sure what the module path is to variable “g”, and I wrote that program.
- In fact, the Google API dynamically creates the classes based on JSON that the program gets from Google at run time.
- There is NO SOURCE CODE to read! But, when the Google API created the class, it documented it from the JSON.
- (I spent quite some time trying to figure out something the other day. Knowing that I could do “help” like that would have been great.)

Help!

```
>>> help()
Welcome to Python 2.7!  This is the online help utility.

If this is your first time using Python, you should definitely check out
the tutorial on the Internet at http://docs.python.org/2.7/tutorial/.

Enter the name of any module, keyword, or topic to get help on writing
Python programs and using Python modules.  To quit this help utility and
return to the interpreter, just type "quit".

To get a list of available modules, keywords, or topics, type "modules",
"keywords", or "topics".  Each module also comes with a one-line summary
of what it does; to list the modules whose summaries contain a given word
such as "spam", type "modules spam".

help>
```

⇐ **A help command! I forgot about that**.

Subtlies

Commas in tuples. Particularly in db calls. `c.execute( sql, ( arg ) )` fails.

Just remember that a comma after a single value is needed to create a tuple.

```
c.execute(sql,(arg,))
                  ^  ⇐ That comma
```

Python is pretty free of strange syntactical requirements. But, the trailing comma in a single element tuple is one that exists.

Unicode!

Learn to use it, learn to love it, and miss the annoying exceptions that you will get otherwise.

We covered this earlier, but in case you forgot, get set up for unicode.

Python will throw exceptions at you if you attempt to print a unicode string but do not have LANG set up appropriately.

Very annoying to process 5,000 names only to find out the 5,001 name has an unusual accent that has an ordinal value > either 127 or 255, depending on LANG.

Tracebacks

```
[mike@googleapps ~]$ /usr/local/src/ga-py/ga
Traceback (most recent call last):
  File "/usr/local/src/ga-py/ga", line 42, in <module>
    import ga_app.cmds.interactive
  File "/usr/local/src/ga-py/ga_app/cmds/interactive.py", line 8, in <module>
    import ga_app.argp.maincmd
  File "/usr/local/src/ga-py/ga_app/argp/__init__.py", line 14, in <module>
    import ga_app.argp.user
  File "/usr/local/src/ga-py/ga_app/argp/user.py", line 8, in <module>
    1/0
ZeroDivisionError: integer division or modulo by zero
```

Tracebacks are really useful. They show the execution process from the start to the error, provide the source file and the line number where each function call occured. The last part will give the exact that caused the error and the error class. Often times, just look at the of to see what failed.

The End!

I hope this is 1.5 hours from when I started. Really, really do!