Background & Overview
Beginnings

Developed by Guido Van Rossum, BDFL, in 1990

(Guido is a Monty Python fan.)

Cornerstones:

- simple, general-purpose, high level, object-oriented
Qualities

- Dynamically typed
- Compilation step rarely fails
- Considered by many to be a scripting language
- Interpreted
- Interactive
- Object-optional
Style

- Useful where Perl is useful, but simpler syntax & semantics
- Class style typical of dynamic languages
  - dynamic creation of slots
  - classes exist at run time
- Ruby, JavaScript
Runs Everywhere

Python available for

- Linux
- MacOS
- Solaris
- Windows
- ...

Compiled binaries are cross-platform compatible
Python available for

- Linux
- MacOS
- Solaris
- Windows
- ...

Compiled binaries are cross-platform compatible

But watch out for different releases
Development Environments

- **IDLE**
  - comes with Python (from python.org)
- Eclipse plug-in available
At the very basic statement level, Python can look a lot like C++ or Java

```python
name = "James"
sys.stdout.write( "Hey, " + name + ",&n
", pi = " + str(math.pi) )
```
But, very importantly, that code snippet can (almost) work all by itself

% python
>>> import sys, math
>>> name = "James"
>>> sys.stdout.write( "Hey, " + name + ", pi = " + str(math.pi) + "\n" )
Hey, James, pi = 3.14159265359
%
But it can be even simpler.

```python
% python
>>> "Hello, world!"
'Hello, world!'
>>> name = "James"
>>> name
'James'
>>> n = 22.0 / 7
>>> n
3.1428571428571428
>>> if n < 3: print( "smaller" )
...
>>> 
```
The large Python standard library contains almost any common utility one might want.

Python interfaces well with other languages, due largely to its C base.

Variations: CPython (fast), Jython (Java), Python.NET
Syntax

and a bit of semantics
As seen, expressions, including function calls and assignment statements look a lot like C.

Python programs are a list of definitions:

- variable assignments (=), functions (def), classes (class)

Different semantics

- Definitions are executed.
- A definition creates an object
Operators

Arithmetic
- + - * / // % **

Relational
- == != < > <= >=

Logical
- and or not

Bitwise
- & | ~
Equality, by default, is reference-based, but most classes redefine it to be value-based.
Operators

Arithmetic
+ - * / // % **

Relational
== != < > <= >=

Logical
and or not

Bitwise
& | ~

Equality, by default, is reference-based, but most classes redefine it to be value-based.

Newsflash: "a < b < c" works!
Arrays, called lists (type `list`), are accessed with brackets.

Negative indices work right (-1) to left (-length)

```python
>>> a = [ 0, 1, 1, 2, 3, 5, 8, 13, 21, 33 ]
>>> a[ 6 ]
8
>>> a[ 9 ] = 34
>>> a[ 9 ]
34
>>> a[ -2 ]
21
```
Lists versus Tuples

Tuples (type `tuple`) are built just as lists are, but with parentheses.

Tuples are immutable.

```python
>>> a = ( 0, 1, 1, 2, 3, 5, 8, 13, 21, 33 )
>>> a[ 6 ]
8
>>> a[ 9 ] = 34
TypeError: 'tuple' object does not support item assignment
```
Dictionaries

Dictionaries (type `dict`) are hash tables.

```python
>>> d1 = {}
>>> d1[ "a" ] = 4
>>> d1[ 4.3 ] = 'four point 3'
>>> d1
{'a': 4, 4.3: 'four point 3'}
>>> d2 = {'a': 4, 4.3: 'four point 3'}
>>> d1 == d2
True
>>> d3 = dict( ( ( 'a', 4 ), ( 4.3, 'four point 3' ) ) )
>>> d3
{'a': 4, 4.3: 'four point 3'}
>>> d3 == d1
True
```
Dictionaries (type `dict`) are hash tables.

```python
>>> d1 = {}
>>> d1[ "a" ] = 4
>>> d1[ 4.3 ] = 'four point 3'
>>> d1
{'a': 4, 4.3: 'four point 3'}
>>> d2 = { 'a': 4, 4.3: 'four point 3' }
>>> d1 == d2
True
>>> d3 = dict( ( ( 'a', 4 ), ( 4.3, 'four point 3' ) ) )
>>> d3
{'a': 4, 4.3: 'four point 3'}
>>> d3 == d1
True
```
String Syntax

String literals: single or double quotes
- no character type

Strings are indexable.

Strings are immutable.

```python
>>> a="0112358ABD"
>>> a
'0112358ABD'
>>> a[6]
'8'
>>> a[9]='C'
TypeError: 'str' object does not support item assignment
```
To control program flow:

- `if`*/elif*/`else`
- `while`*
- `for` - like a `foreach`

*Parentheses not required for conditions

No case statement
Because Python is dynamically typed,

- Function parameters have no declared types.
- Functions do not declare a return type.

There is a `return` statement.

- If no `return` statement is executed by a called function, it returns `None`.
Programmers Must Indent!
if height < MINIMUM:
    print( "Sorry, not allowed." )
if height < MINIMUM:
    print( "Sorry, not allowed." )

if height < MINIMUM:
    print( "Sorry, not allowed." )
else:
    print( "Go ahead!" )
if height < MINIMUM:
    print( "Sorry, not allowed." )

if height < MINIMUM:
    print( "Sorry, not allowed." )
else:
    print( "Go ahead!" )

height = float( input( "Height? " ) )
while height > 0:
    if height < MINIMUM:
        print( "Sorry, not allowed." )
    else:
        print( "Go ahead!" )
    height = float( input( "Height? " ) )
if height < MINIMUM:
    print( "Sorry, not allowed." )
else:
    print( "Go ahead!" )

if height < MINIMUM:
    print( "Sorry, not allowed." )
else:
    print( "Go ahead!" )
def admit( numRiders ):
    count = 0
    height = float( input( "Height? " ) )
    while count < numRiders:
        if height < MINIMUM:
            print( "Sorry, not allowed." )
        else:
            print( "Go ahead!" )
            count += 1
        height = float( input( "Height? " ) )
height = float( input( "Height? " ) )
while height > 0:
    if height < MINIMUM:
        print( "Sorry, not allowed." )
    else:
        print( "Go ahead!" )
    height = float( input( "Height? " ) )
Rule: Compound Statements

- All compound statements have at least one header line and one body, a sequence of statements.
- The header must end in a colon (:).
- The body must be uniformly indented.
- A one-statement body may be appended to the header.
Three ways to execute Python:

(a) Type statements directly into the interpreter.
(b) Provide a module (Python code file) name on the command line to execute that module.
(c) Use (a) or (b) and *import* other modules.
**import module**

- Module name: file name, without `.py`, `.pyc`, `.pyo`
- Reads and executes the named module, but keeps it in a separate namespace.
  - idempotent!
- `from module import list`
  - Merges `module` def.'s into importer's namespace.
import module

Module name: file name, without .py, .pyc, .pyo

Reads and executes the named module, but keeps it in a separate namespace.

idempotent!

from module import list

Merges module def.'s into importer's namespace.
# Demo
# James Heliotis, November 2009

__doc__ = "Demonstration module"

var = "I am a variable"

def what():
    "Prints this module's name."
    print( "My name is", __name__ )

what()
# Demo
# James Heliotis, November 2009

__doc__ = "Demonstration module"

var = "I am a variable"

```python
def what():
    "Prints this module's name."
    print( "My name is", __name__ )

what()
```

$ python3 demo.py
My name is __main__
# Demo
# James Heliotis, November 2009

__doc__ = "Demonstration module"

var = "I am a variable"

def what():
    "Prints this module's name."
    print( "My name is", __name__ )

what()
Using Modules

---

`demo.py`

```python
# Demo
# James Heliotis, November 2009
__doc__ = "Demonstration module"

var = "I am a variable"

def what():
    "Prints this module's name."
    print( "My name is", __name__ )

what()
```

$ python3 demo.py
My name is __main__

$ python3
>>> from demo import *
My name is demo
>>> var
'I am a variable'
>>> what()
My name is demo
>>> help(demo)
Help on module demo:

NAME
demo - Demonstration module

FILE
/Users/jeh/Source/Python/demo.py

FUNCTIONS
what()
    "Prints this module's name."

DATA
var = 'I am a variable'
# Demo
# James Heliotis, November 2009

__doc__ = "Demonstration module"

var = "I am a variable"

```python
def what():
    "Prints this module's name."
    print( "My name is", __name__ )

what()
```

```bash
$ pydoc3 demo
Help on module demo:

NAME
demo - Demonstration module

FILE
/Users/jeh/Source/Python/demo.py

FUNCTIONS
    what()
        Prints this module's name.

DATA
    var = 'I am a variable'
```
```python
__doc__ = "Demonstration module"

var = "I am a variable"

def what():
    "Prints this module's name."
    print( "My name is", __name__ )

what()

$ pydoc3 -w demo
My name is demo
wrote demo.html
```
# Demo
# James Heliotis, November 2009

__doc__ = "Demonstration module"

var = "I am a variable"

def what():
    "Prints this module's name."
    print( "My name is", __name__ )

what()

$ pydoc3 -w demo
My name is demo
wrote demo.html
As in Java, a Python package is just a directory containing modules and other packages.

If directory a contains directory b, and directory b contains file stuff.py,

That file’s code is referenced via a.b.stuff.
**Java vs. Python**

**Java**

-no import

Must use fully qualified name

`p1.p2.foo`

import `p1.p2.foo`

`foo` works on its own.

import `p1.p2.*`

Anything in `p1.p2` can be mentioned unqualified.

**Python**

-no import

Item cannot be used.

import `p1.p2`

Must use fully qualified name

`p1.p2.foo`

from `p1.p2` import `foo`

`foo` works on its own.

from `p1.p2` import *

Anything in `p1.p2` can be mentioned unqualified.
Python defines many variables, functions (methods) and object attributes whose names begin \textit{and} end in two underscores.

These items have special connections to the compiler.
Special Variable Examples

__slots__ is a class variable holding list of attribute names so that no other ones can be created in that class’s objects.

If $x+y$ is seen in source code, the compiler generates a call to $x.__add__(y)$.

__name__ is a global variable naming the "current" module.

__doc__ is an attribute of a module containing its documentation string (see earlier demo example).
len( x ) invokes x.__len__().

str( x ) invokes x.__str__().

iter( x ) invokes x.__iter__().

x[ i ] invokes x.__getitem__().

x.y invokes x.__getattr__( "y" ).
Object Orientation
class Point( object ):  
    __slots__ = [ "x", "y" ]  
    def __init__( self, xCoord, yCoord ):  
        self.x = xCoord  
        self.y = yCoord  
    def getX( self ):  
        return self.x  
    def getY( self ):  
        return self.y  
    def distFromOrigin( self ):  
        return ( self.x**2 + self.y**2 ) ** 0.5  
    def __str__( self ):  
        return "Point( " + str( self.x ) + ", " + str( self.y ) + " )"

p = Point( 3, 4 )
print( p.getX() )
print( str( p ) )
print( p.distFromOrigin() )
Classes in Python

- **Parent class(es)**
- **Multiple inheritance allowed**
- Attributes via `__slots__` list (optional)
- Methods: explicit `this/self` first argument
- All features are public.
Construction

1. Client calls class like a function.
   
   ```python
   p = Pet( "cat", "Chester", 5 )
   ```

2. Object gets created, sans slots.

3. Object reference is prepended to the arg. list, and the method `__init__` is called.
   
   ```python
   def __init__(self,kind,name,age):
   ```
Classes are Objects, Too.

```python
>>> x = [1, 2, 3]
>>> x.__class__
<class 'list'>
>>> x.__class__.__name__
'list'
>>> x.__class__.__class__
<class 'type'>
>>> help(type)
class type(object)
|  type(object) -> the object's type
|  type(name, bases, dict) -> a new type
|
|  Methods defined here:
|
|  mro(...)
|    mro() -> list
|    return a type's method resolution order
|
|  __new__ = <built-in method __new__ of type object at 0x32e600>
|    T.__new__(S, ...) -> a new object with type S, a subtype of T
|
|  __prepare__ = <built-in method __prepare__ of type object at 0x32e600>
|    __prepare__() -> dict
|    used to create the namespace for the class statement
```
Because Python allows multiple inheritance, it is usually best to refer to superclasses by name, à la C++. 
from point import Point

class ColoredPoint( Point ):

    __slots__ = [ "c" ]

    def __init__( self, xCoord, yCoord, color ):
        Point.__init__( self, xCoord, yCoord )
        self.c = color

    def __str__( self ):
        return Point.__str__( self ) + \
             '[' + self.c + ']

    def getColor( self ):
        return self.c

cp = ColoredPoint( 8, 10, "red" )
print( cp.getX() )
print( str( cp ) )
print( cp.distFromOrigin() )
print( cp.getColor() )
Exceptions
Python exceptions work in the model of a stack-upward propagation. Started by executing a `raise` statement. Only instances of descendants of `BaseException` may be raised. Caught in an `except` block below a `try` block. `except` has a single argument type (variable optional).
class MyError( Exception ):
    def __init__( self, value ):
        self.value = value
    def __str__( self ):
        return str( self.value )

try:
    raise MyError( 2 * 2 )
except MyError as e:
    print( 'My exception occurred, value:', e )
raise MyError( 'oops!' )

My exception occurred, value: 4
Traceback (most recent call last):
  File "exception1.py", line 12, in <module>
    raise MyError( 'oops!' )
__main__.MyError: 'oops!'
class MyError( Exception ):
    def __init__(self, value):
        self.value = value
    def __str__(self):
        return str(self.value)

class MyOtherError( Exception ):
    def __init__(self, value):
        self.value = value
    def __str__(self):
        return str(self.value)

exceptions = ( MyError, MyOtherError, Exception )

for i in range( len( exceptions ) ):
    try:
        raise exceptions[ i ]( i )
    except MyError as e:
        print( "(A) A MyError occurred, value: 0" )
    except Exception as e:
        print( "(B) A Exception occurred, value: 2" )

(A) A MyError occurred, value: 0
(B) A MyOtherError occurred, value: 1
(B) A Exception occurred, value: 2
Slicing

Once can make sub-lists and sub-tuples using *slices*.

- \( a = [ 0, 2, 4, 6, 8 ] \)
- \( a2 = a[2] \)
- \( a468 = a[2 : 5] \) # or \( a[2 :] \)
- \( a048 = a[0 : 5 : 2] \)
Other Operations

Concatenation

\[ [1, 3] + [2, 4] \rightarrow [1, 3, 2, 4] \]

Duplication

\[ (0, 1) \times 4 \rightarrow (0, 1, 0, 1, 0, 1, 0, 1) \]

\[ [\text{None}] \times \text{SIZE} \]

creates an array of known size
Sequences and Iterators

An iterable is any object that responds to __iter__() by returning an iterator.

An iterator is any object that responds to __next__() and throws a StopIteration exception if nothing is left.

A sequence is an iterable that supports integer indexing using __getitem__(i) and __len__()
Sequences and Iterators

An *iterable* is any object that responds to `__iter__()` by returning an iterator.

An *iterator* is any object that responds to `__next__()` and throws a `StopIteration` exception if nothing is left.

A *sequence* is an iterable that supports integer indexing using `__getitem__(i)` and `__len__()`. 
How for loops work

```python
>>> nums = [1,2,3]
>>> i = nums.__iter__() # Get the iterator from the iterable.
>>> try:
...   while True:
...     print( i.__next__() )
... except StopIteration:
...   pass
... pass
1
2
3
```
Making One’s Own Iterator

class range2( object ):  
    def __init__( self, n, max=None, step=None ):  
        if step == None:  
            self.step = 1  
        else:  
            self.step = step  
        if max == None:  
            self.stop = n  
            self.current = 0  
        else:  
            self.stop = max  
            self.current = n  
    def __iter__( self ):  
        return self  
    def __next__( self ):  
        result = self.current  
        if result >= self.stop:  
            raise StopIteration()  
        else:  
            self.current += self.step  
        return result

>>> r = range2( 1, 7, 2 )
>>> i = iter( r )
>>> try:
...   while True:
...     print( next( i ) )
... except StopIteration:
...   pass
...1
3
5

>>> r = range2( 1, 7, 2 )
>>> for n in r: print( n )
...
1
3
5

>>> r = range2( 1, 7, 2 )
>>> list( r )
[1, 3, 5]

>>>

Wednesday, December 2, 2009
Generators

A short-cut for building an iterator.

Write the loop yourself!

def range3( n, max=None, step=None ):
    if step == None: step = 1
    if max == None:
        stop = n
        current = 0
    else:
        stop = max
        current = n
    while current < stop:
        yield current
        current += step

for i in range3( 3 ): print( i )
A built-in function to apply a function to an iterable

```python
>>> nums = [1, 2, 3]
>>> def double(x): return 2 * x
... >>> num2 = map(double, nums)
>>> num2
<map object at 0x82b758c>
>>> list(num2)
[2, 4, 6]
```
A built-in function that uses a predicate to prune an iterable

```python
>>> def cutDF( grade ): return grade != "D" and grade != "F"
... 
>>> type( f )
<class 'filter'>
>>> list( f )
[ 'B', 'A', 'B', 'B', 'A', 'C', 'B' ]
>>> list( f )
[]
>>> 
Wednesday, December 2, 2009
```
Reduce

A formerly built-in function to apply a function to an iterable

```python
>>> from functools import reduce
>>> reduce( max, ( 1, 3, 5, 4, 2 ) )
5
>>> reduce( min, ( 1, 3, 5, 4, 2 ), 0 )
0
>>> def subtract(x,y): return x-y
...  
>>> reduce( subtract, [ 4, 2, 1 ] )
1
>>> 
```
List Comprehensions

Shorthand notation for generating lists

Mapping section is required.

Filtering section is optional.

```python
>>> [ x**2 for x in range(11) ]
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100]

>>> [ ( x, x**2 ) for x in range( 5 ) ]
[(0, 0), (1, 1), (2, 4), (3, 9), (4, 16)]

>>> [ x**2 for x in range(11) if x % 2 == 0 ]
[0, 4, 16, 36, 64, 100]
```
def mystery(lst):
    if lst == []:  # what does this do???
        return []
    else:
        return
            \\
        mystery([x for x in lst[1:] if x < lst[0]]) + \\
        lst[0:1] + \\
        mystery([x for x in lst[1:] if x >= lst[0]])
Exercise

What does this do???

def mystery(lst):
    if lst == []:
        return []
    else:
        return\
        mystery([x for x in lst[1:] if x < lst[0]]) +\
        lst[0:1] +\
        mystery([x for x in lst[1:] if x >= lst[0]])

>>> mystery([1,3,5,7,9,10,8,6,4,3,2,1])
[1, 1, 2, 3, 3, 4, 5, 6, 7, 8, 9, 10]

>>>
Exercise

Write a function that takes two iterables (for-each-able) and creates a single list of bituples:

```
combine( [1,2,3] , (5,6) )
```

--> 

```
[(1,5),(2,6),(3,None)]
```
Other Important Features
Tuples as Parameter Lists

A single asterisk before the last parameter says that it may be a tuple of argument values, like varargs or Java's "..".

```python
def product( *nums ):
    result = 1
    for x in nums: result *= x
    return result

print( product( 5, 6, 7 ) )
print( product( 8, 9 ) )
print( product( 10 ) )
print( product() )
numList = ( 2, 3, 4 )
print( product( *numList ) )
print( product( numList ) )
```

210
72
10
1
24
(2, 3, 4)
Keyword-based Parameter Lists

Python allows naming arguments in a call, as long as the unnamed ones appear first.

```python
def draw( obj=None, color="black", background="white", shadow=False ):
    print( "Drawing " + str(obj) + ", color=" + color + ", bkgd=" + background + ", end = " )
    if not shadow:
        print( "no ", end = " )
    print( "shadow." )

draw()
draw( "OBJ1" )
draw( "OBJ2", color = "purple" )
draw( "OBJ1", shadow = True )
draw( "OBJ1", background = "green", color = "red" )
```

Drawing None, color=black, bkgd=white, no shadow.
Drawing OBJ1, color=black, bkgd=white, no shadow.
Drawing OBJ2, color=purple, bkgd=white, no shadow.
Drawing OBJ1, color=black, bkgd=white, shadow.
Drawing OBJ1, color=red, bkgd=green, no shadow.
Keyword-based Parameter Lists

Python allows naming arguments in a call, as long as the unnamed ones appear first.

def draw( obj=None, color="black", background="white", shadow=False ):
    print( "Drawing " + str(obj) + ", color=", color + "", bkgd=" + background + ", ", end = "")
    if not shadow:
        print( "no ", end = "")
    print( "shadow." )

draw()
draw( "OBJ1" )
draw( "OBJ2", color = "purple" )
draw( "OBJ1", shadow = True )
draw( "OBJ1", background = "green", color = "red" )

Drawing None, color=black, bkgd=white, no shadow.
Drawing OBJ1, color=black, bkgd=white, no shadow.
Drawing OBJ2, color=purple, bkgd=white, no shadow.
Drawing OBJ1, color=black, bkgd=white, shadow.
Drawing OBJ1, color=red, bkgd=green, no shadow.
Dictionary-based Parameter

Argument names can also be made to correspond to the keys in a single dictionary parameter.

def demo( **params ):
    print( "first =", params[ 'first' ] )
    print( "second =", params[ 'second' ] )

demo( first="#1", second="#2" )
args = { "second": 2, "first": 1 }
demo( **args )

first = #1
second = #2
first = 1
second = 2
Python supports the functional programming closure concept.

```python
def getNumberPrinter( start ):
    def printParameter():
        print( start )
        return printParameter

a = getNumberPrinter( 1 )
z = getNumberPrinter( 10 )
a()
a()
a()
z()
```

1
1
10
Lambda Functions

Anonymous functions

In Python, the body is limited to a single expression.

```python
>>> ( lambda x, y: ( x + y ) / 2 )( 3, 10 )
6.5
>>> tuple( map( lambda n: n**3, range(5) ) )
(0, 1, 8, 27, 64)
>>> reduce( lambda x, y: x * y, [ 3, 2, 8, 20 ] )
960
>>> 
```
Variable Binding in Python

Remember that all variable assignments, including function definitions, are executed at run time.

All names in Python are variables: module names, function names, class names, ....

No variable’s value is hard-wired during compilation.
def log( f ):
    def logAndRun( *args ):
        print( f.__name__, "called with args", args )
        return f( *args )
    return logAndRun

>>> def hi( name ): print( "Hello, " + name + '!' )
...  
>>> hi( "Jim" )
Hello, Jim!
>>> hi = log( hi )
>>> hi( "Jim" )
hi called with args ('Jim',)
Hello, Jim!
def log(f):
    def logAndRun(*args):
        print(f.__name__, "called with args", args)
        return f(*args)
    return logAndRun

@log
def hi(name):
    print("Hello, " + name + '!')

>>> hi("Jim")
hi called with args ('Jim',)
Hello, Jim!
def ensureNonNegative( f ):
    def runAndCheck( *args ):
        result = f( *args )
        if result < 0: raise Exception( "Negative result?" )
        return result
    return runAndCheck

@ensureNonNegative
def locate( lst, val ):
    for i in range( len( lst ) ):
        if lst[ i ] == val: return i
    return -1

>>> nums = range( 0, 11, 2 )
>>> locate( nums, 2 )  
1
>>> locate( nums, 6 )  
3
>>> locate( nums, 7 )  
Exception: Negative result?
>>>
def checkResult(expected):
    def applyToFn(f):
        def runAndCheck(*args):
            result = f(*args)
            if result != expected:
                raise Exception("Test failed!")
            return result
        return runAndCheck
    return applyToFn

>>> @checkResult(8)
... def halve(n):
...     return n // 2
... return n // 2
>>> halve(16)
8
>>> halve(14)
Exception: Test failed!
>>>

(Not Necessarily Useful...)

Wednesday, December 2, 2009
Want More Information?

python.org
Contact Information

James Heliotis

jeh@cs.rit.edu

http://www.cs.rit.edu/~jeh
I dunno...
Dynamic typing?
Whitespace?
Come join us!
Programming is fun again!
It's a whole new world up here!
But how are you flying?

I just typed
import antigravity
That's it?

...I also sampled
everything in the medicine cabinet
for comparison.

But I think this is the Python.

I learned it last night! Everything is so simple!
Hello world is just print "Hello, world!"