Beyond Blocks: Python Session #1

Goals

- Quick introduction to Python
  - *Not* a tutorial or "how to"
  - Hope is that you'll want to learn (more)
- Advantages over higher level languages
- Challenges of programming syntax
  - Hope is that "foreign" syntax becomes less intimidating and more approachable

Beyond Blocks: Python #1

Installation: Mac Check

- Open Terminal
- Type "`python`" and hit `return`
  - (without the quotes)
- Type "`print 'hello world'`" `return`

```
print 'hello world'
```

- The result should be:

```
>>> print 'hello world'
hello world
```

Beyond Blocks: Python #1

Installation: Windows Check

- Get Python to "`print`" something with these instructions:

  http://docs.python.org/faq/windows.html

(You only have to get to the "Many people use the interactive mode as a convenient yet highly programmable calculator" paragraph)

Beyond Blocks: Python #1

Installation: More Information

- Computer Science Circles : Run Python at Home

  cemclinux1.math.uwaterloo.ca/~cscircles/wordpress/run-at-home/

Beyond Blocks: Python #1

Installation: Version Check

```
[/Users/headcrash]> python -V
Python 2.7.2
```

We'll be talking about version 2.7.2 in here, although version 3.2.2 is the "latest" (as of today).

If curious, there's more version info at:
http://docs.python.org/whatsnew/index.html

Beyond Blocks: Python #1

Installation: Version Check

```
[/Users/headcrash]> python -V
Python 2.7.2
```

We'll be talking about version 2.7.2 in here, although version 3.2.2 is the "latest" (as of today).

If curious, there's more version info at:
http://docs.python.org/whatsnew/index.html

Beyond Blocks: Python #1

Why used “text based” programming?

<Fibonacci demo>

Beyond Blocks: Python #1

Compiled vs. Interpreted

- Compiled
  - *Usually* much faster
  - Ability to edit and save program.
  - Mac: `python [filename.py]` in Terminal.
  - Win: `python [filename.py]` on command line (or run under Idle)

Beyond Blocks: Python #1

Compiled vs. Interpreted

- Interpreted
  - Ability to try out commands interactively.
  - Mac: (just) `python` in Terminal.
  - Win: (just) `python` on command line or launch the Idle executable

BYOB ⟷ Python

BYOB ⟷ Python

Variables


```
>>> var = 0
>>>
```

BYOB ⟷ Python

Variables


```
>>> var=1
>>> var
1
```

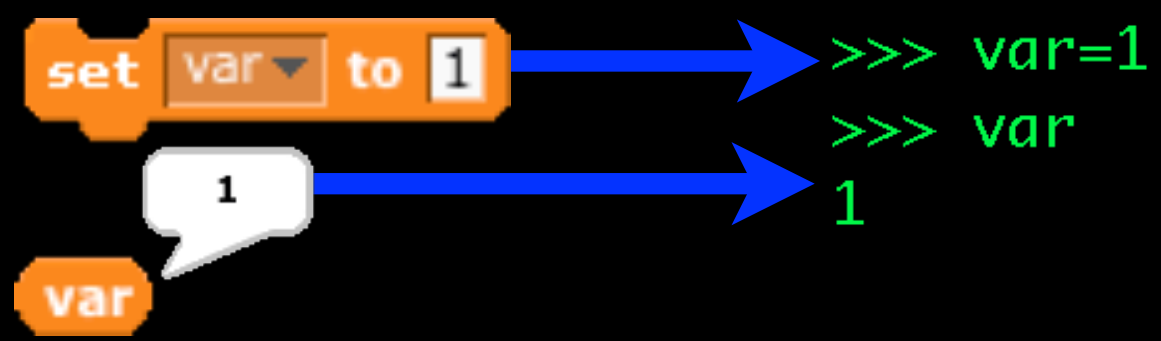
BYOB ⟷ Python

Variables


```
>>> var=1
>>> var
1
```

BYOB ⟷ Python

Variables



BYOB ⟷ Python

Variables


```
>>> var=1
>>> var
1
```

NOTE:
Assignment doesn't "evaluate" to anything, so nothing is printed!

BYOB ⟷ Python

Variables


```
>>> var = var + 1
```

BYOB ⟷ Python

Variables


```
>>> print var
1
```

BYOB ⟷ Python

Operators


```
>>> 1+1
2
>>> 2-1
1
>>> 2*2
4
>>> 6/2
3
```

BYOB ⟷ Python

Types

- Everything in Python has an internal "type"
- Types are determined *dynamically*
  - x = 1
  - x now has the type "int":
    - (short for "integer")

```
>>> x=1
>>> type(x)
<type 'int'>
>>>
```

We'll talk about this "script" (or function) later...

BYOB ⟷ Python

Types: bool

- ‘bool’ is short for boolean
- ‘bool’s can have two values:
  - True → true → True
  - False → false → False

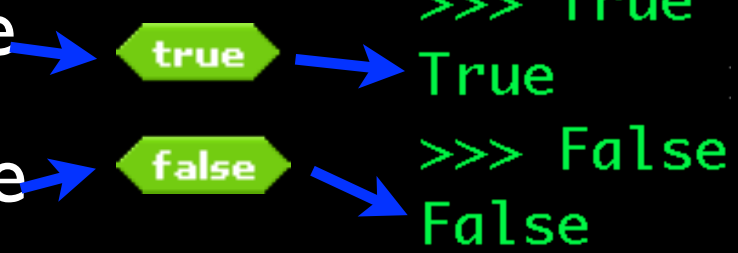
```
>>> True
True
>>> False
False
```

```
>>> type(True)
<type 'bool'>
```

BYOB ⟷ Python

Types: bool

- ‘bool’ is short for boolean
- ‘bool’s can have two values:
  - True
  - False




```
>>> True
True
>>> False
False
```

```
>>> type(True)
<type 'bool'>
```

NOTE: Upper case is important!

BYOB ⟷ Python

Types: function type()

This function returns the type that Python has assigned the *identifier*.

```
>>> type(True)
<type 'bool'>
>>> type(1)
<type 'int'>
>>> type(1.0)
<type 'float'>
>>> type("blah!")
<type 'str'>
```

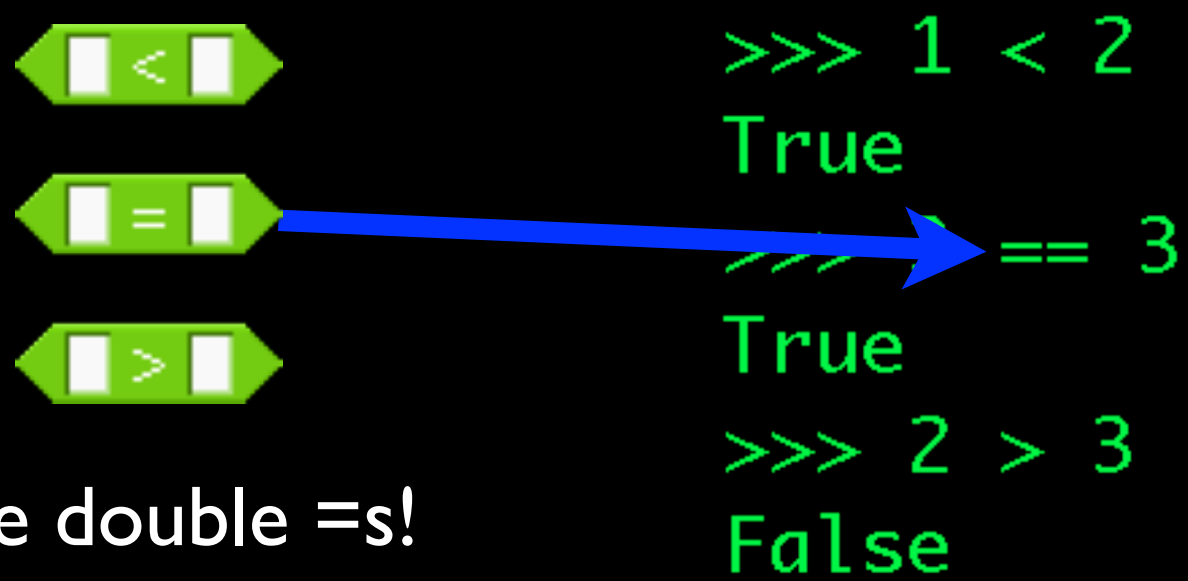
BYOB ⟷ Python

Operators


```
>>> 1 < 2
True
>>> 3 == 3
True
>>> 2 > 3
False
```

BYOB ⟷ Python

Operators



- Note the double =s!
- = means ***assign***, == means ***compare***
- Very common source of bugs!

BYOB ⟷ Python

Operators


```
>>> 3 % 2
1
>>> 12345 % 678
141
```

BYOB ⟷ Python

Sidebar: Division (integer vs. real/float)


```
>>> 5 / 6
0
>>> 5.0 / 6.0
0.8333333333333337
>>> 5.0 // 6.0
0.0
```

BYOB ⟷ Python

Sidebar: Division (integer vs. real/float)


```
>>> 5 / 6
0
>>> 5.0 / 6.0
0.8333333333333337
>>> 5.0 // 6.0
0.0
```

Same operator, "/," but
output type depends on input types!

BYOB ⟷ Python

Sidebar: Division (integer vs. real/float)


```
>>> 5 / 6
0
>>> 5.0 / 6.0
0.8333333333333337
>>> 5.0 // 6.0
0.0
```

Same operator, "/," but
output type depends on input types!

BYOB ⟷ Python

Sidebar: Division (integer vs. real/float)


```
>>> 5 / 6
0
>>> 5.0 / 6.0
0.8333333333333337
>>> 5.0 // 6.0
0.0
```

"Force" integer division

BYOB ⟷ Python

Sidebar: Exponent

BYOB has e^x and 10^x,
but Python can do <u>any</u> base & exponent!

```
>>> 2**8
256
>>> 2**10
1024
>>> 2**100
1267650600228229401496703205376L
```

BYOB ⟷ Python

Sidebar: Exponent

What's that "L?"

```
>>> 2**8
256
>>> 2**10
1024
>>> 2**100
1267650600228229401496703205376L
```

BYOB ⟷ Python

Sidebar: Exponent

```
>>> 2**8
256
>>> 2**10
1024
>>> 2**100
1267650600228229401496703205376L
>>> type(2**100)
<type 'long'>
```

BYOB ⟷ Python

Sidebar: Exponent

```
>>> 2**8
256
>>> 2**10
1024
>>> 2**100
1267650600228229401496703205376L
>>> type(2**100)
<type 'long'>
```

Just (for now) means: "a really big integer."

BYOB ⟷ Python

Operators


```
>>> True and False
False
>>> True and True
True
>>> True or False
True
>>> not True
False
>>> not False
True
```

BYOB ⟷ Python

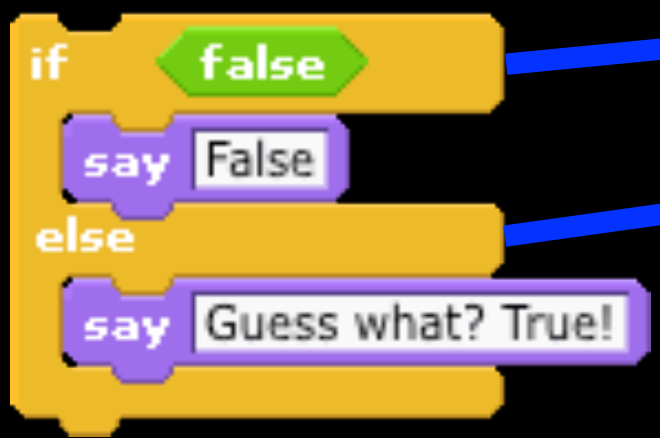
Conditionals


```
>>> if (True):
...     print "True"
...
True
>>> if (False):
...     print "False"
... else:
...     print "Guess what? True!"
...
Guess what? True!
```

BYOB ⟷ Python

Conditionals


```
>>> if (True):
...     print "True"
...
True
```




```
>>> if (False):
...     print "False"
... else:
...     print "Guess what? True!"
...
Guess what? True!
```

BYOB ⟷ Python

Conditionals

```
>>> if (True):
...     print "True"
...
True
>>> if (False):
...     print "False"
... else:
...     print "Guess what? True!"
...
Guess what? True!
```

Notice the colon and indentation *syntax*!

BYOB ⟷ Python

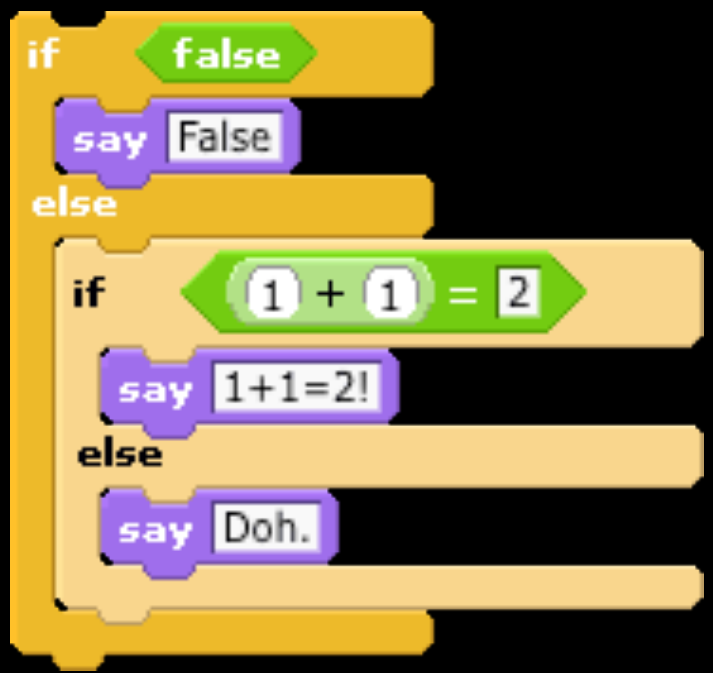
Conditionals

```
>>> if (True):
...     print "True"
...
True
>>> if (False):
...     print "False"
... else:
...     print "Guess what? True!"
...
Guess what? True!
```

Notice the colon and indentation *syntax*!

BYOB ⟷ Python

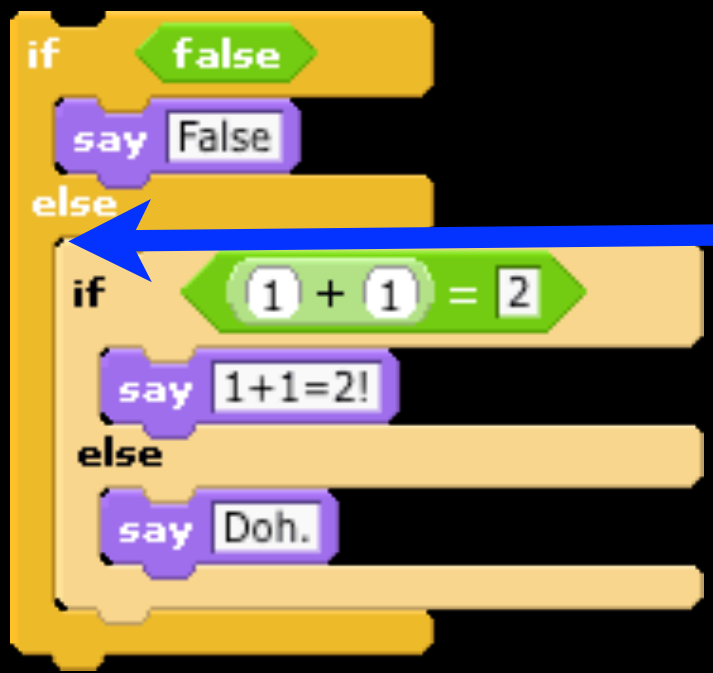
Conditionals




```
>>> if (False):
...     print "False"
... elif (1+1==2):
...     print "1+1==2!"
... else:
...     print "Doh."
...
1+1==2!
```

BYOB ⟷ Python

Conditionals




```
>>> if (False):
...     print "False"
... elif (1+1==2):
...     print "1+1==2!"
... else:
...     print "Doh."
...
1+1==2!
```

BYOB ⟷ Python

Loops


```
>>> var = 0
>>> while(True):
...     print var
...     var = var + 1
...
0
1
2
3
4
5
6
7
8
9
```

BYOB ⟷ Python

Loops


```
>>> var = 0
>>> while(True):
...     print var
...     var = var + 1
...
0
1
2
3
4
5
6
7
8
9
```

BYOB ⟷ Python

Loops

```
>>> var = 0
>>> while(True):
...     print var
...     var = var + 1
...
0
1
2
3
4
5
6
7
8
9
```

Note the indentation (again)!

BYOB ⟷ Python

Loops


```
>>> var = 0
>>> while( var < 5 ):
...     print var
...     var = var + 1
...
0
1
2
3
4
```

BYOB ⟷ Python

Loops


```
>>> var = 0
>>> while( var < 5 ):
...     print var
...     var = var + 1
...
0
1
2
3
4
```

BYOB ⟷ Python

More Loops

BYOB ⟷ Python

Moar [sic] Loops

There isn't really an exact equivalent of this in Python...

We'll talk more about this in Session #2...

BYOB ⟷ Python

Functions: Calling

- Calling functions (the *syntax*) looks like this:

```
>>> func(1,2,3)
```

- Equivalent to creating & running a BYOB block:

BYOB ⟷ Python

Functions: Calling

- Calling functions (the *syntax*) looks like this:

```
>>> func(1,2,3)
```

- Equivalent to creating & running a BYOB block:

BYOB ⟷ Python

Functions: Calling

- Calling functions (the *syntax*) looks like this:

```
>>> func(1,2,3)
```

- Equivalent to creating & running a BYOB block:

BYOB ⟷ Python

Functions: Calling

- Calling functions (the *syntax*) looks like this:

```
>>> func(1,2,3)
```

- Equivalent to creating & running a BYOB block:

BYOB ⟷ Python

Functions : Defining

```
>>> def func(arg1,arg2,arg3):
...     pass
...     pass
...
>>>
```

Keyword: DEF

BYOB ⟷ Python

Functions : Defining

```
>>> def func(arg1,arg2,arg3):
...     pass
...     pass
...
>>>
```

Name of the function

BYOB ⟷ Python

Functions : Defining

```
>>> def func(arg1,arg2,arg3):
...     pass
...     pass
...
>>>
```

"Arguments," or inputs to the function

BYOB ⟷ Python

Functions : Defining

```
>>> def func(arg1,arg2,arg3):
...     pass
...     pass
...
>>>
```

Indentation: the key to "scope."

We'll talk about "scope" later...

BYOB ⟷ Python

Functions : Defining

```
>>> def func(arg1,arg2,arg3):
...     pass
...     pass
...
>>>
```

pass: Python's "placeholder" or NOP

NOP: short for "NO OPeration"

(or do nothing...)

BYOB ⟷ Python

Functions : Defining

```
>>> def func(arg1,arg2,arg3):
...     pass
...     pass
...
>>>
```

pass: Python's "placeholder" or NOP

NOP: short for "NO OPeration"

Functions *must* have a body!

BYOB ⟷ Python

Functions : Defining

```
>>> def func(arg1,arg2,arg3):
...     pass
...     pass
...
>>>
```

Hitting Return/Enter (on an empty line) "closes" (finishes) the definition.

BYOB ⟷ Python

Sidebar: Keywords

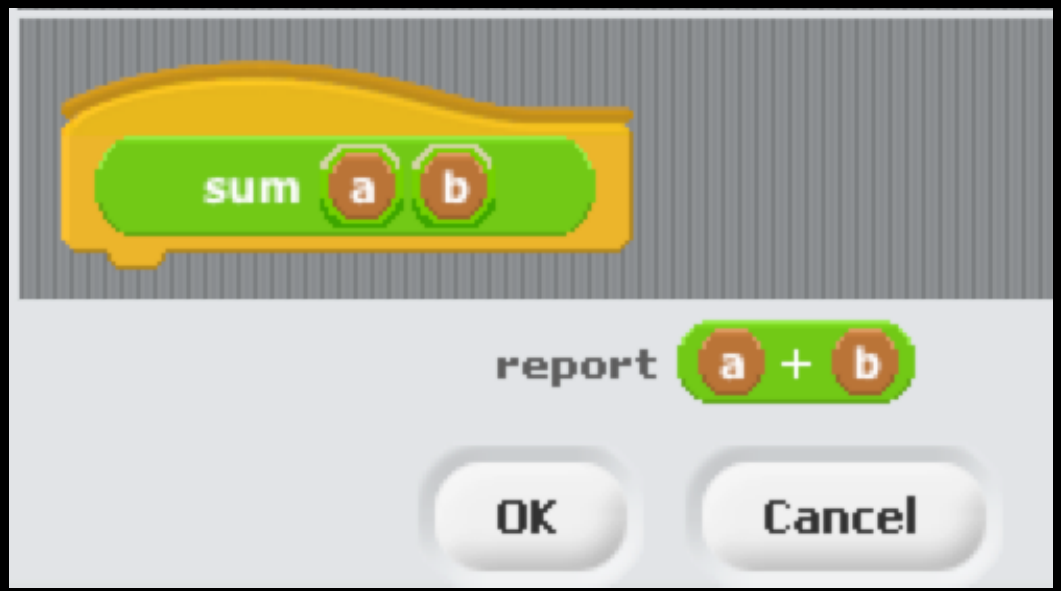
```
and        del        from       not        while
as         elif       global     or         with
assert     else       if         pass       yield
break      except     import     print
class      exec       in         raise
continue   finally    is         return
def        for        lambda     try
```

- Words reserved by Python
  - List at: docs.python.org/reference/lexical_analysis.html

BYOB ⟷ Python

Functions : Returning Values

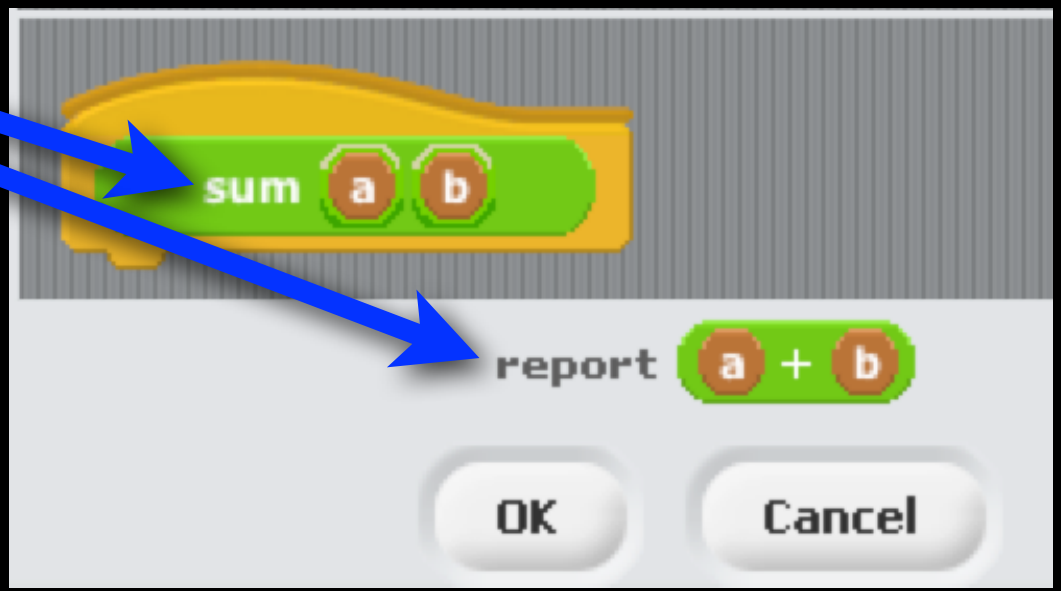
```
>>> def sum(a,b):
...     return (a+b)
...
>>> c=sum(5,7)
>>> print c
12
```



BYOB ⟷ Python

Functions : Returning Values

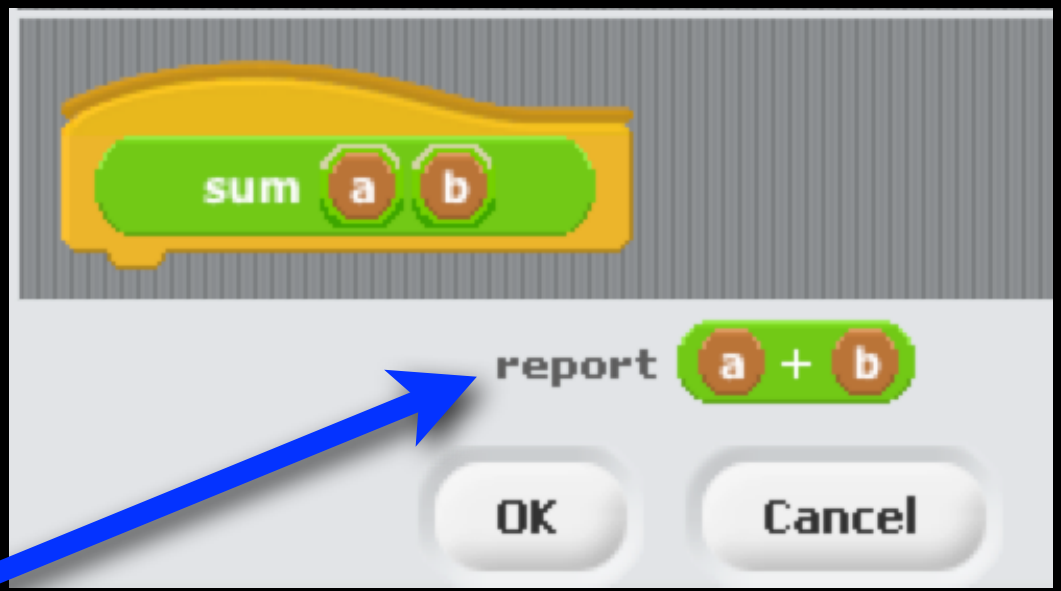
```
>>> def sum(a,b):
...     return (a+b)
...
>>> c=sum(5,7)
>>> print c
12
```



BYOB ⟷ Python

Functions : Returning Values

```
>>> def sum(a,b):
...     return (a+b)
...
>>> c=sum(5,7)
>>> print c
12
```

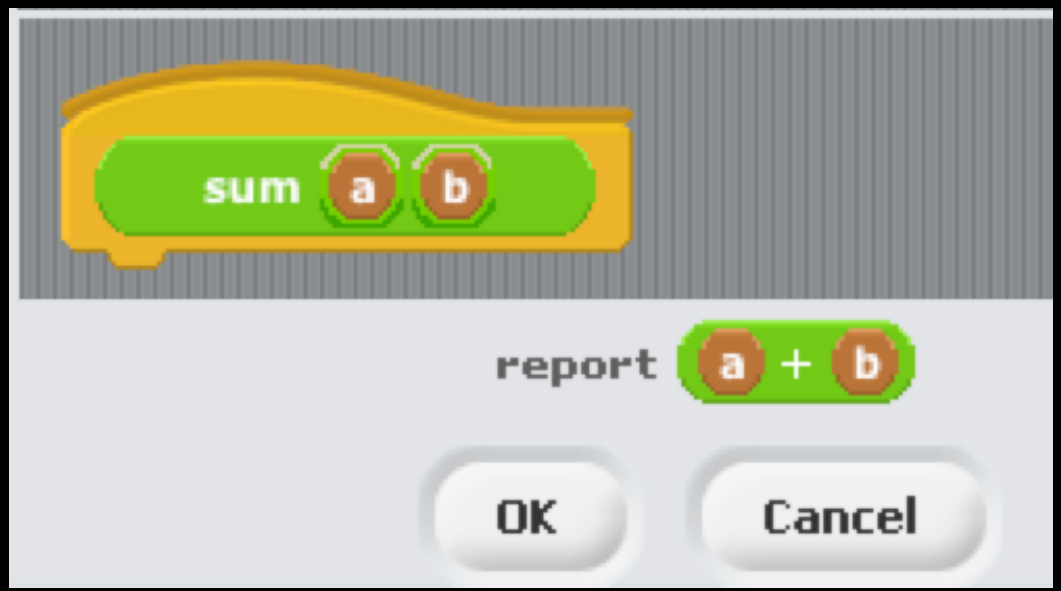


"return" and "report" are equivalent!

BYOB ⟷ Python

Functions : Returning Values

```
>>> def sum(a,b):
...     return (a+b)
...
>>> c=sum(5,7)
>>> print c
12
```



What is the type of the variable ‘c’?

BYOB ⟷ Python

Functions : Type? It depends!

```
>>> def sum(a,b):
...     return a+b
...
>>> c=sum(1,2)
>>> print c
3
>>> type(c)
<type 'int'>
```

BYOB ⟷ Python

Functions : Type? It depends!

```
>>> def sum(a,b):
...     return a+b
...
>>> c=sum(1,2)
>>> print c
3
>>> type(c)
<type 'int'>
```

```
>>> c=sum(1.0,2.0)
>>> print c
3.0
>>> type(c)
<type 'float'>
>>> c=sum("hello"," world")
>>> print c
hello world
>>> type(c)
<type 'str'>
```

BYOB ⟷ Python

Functions : C's type? It depends!

```
>>> def sum(a,b):
...     return a+b
...
>>> c=sum(1,2)
>>> print c
3
>>> type(c)
<type 'int'>
```

!

```
>>> c=sum(1.0,2.0)
>>> print c
3.0
>>> type(c)
<type 'float'>
>>> c=sum("hello"," world")
>>> print c
hello world
>>> type(c)
<type 'str'>
```

BYOB ⟷ Python

Functions : Practice

```
>>> def fun1( arg1, arg2 ):
...     return arg1 + arg2
...
>>> def fun2( arg3, arg4 ):
...     x = fun1( arg3, 1)
...     y = fun1( arg4, 1)
...     return x + y
...
>>> print fun2(5,6)
```

What will this print?

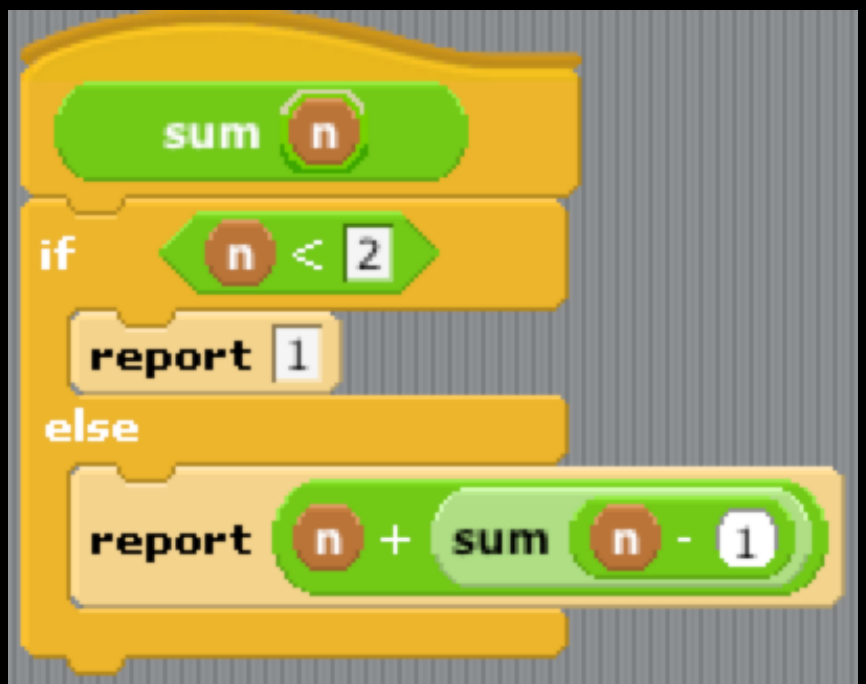
BYOB ⟷ Python

Functions : Practice

```
>>> def fun1( arg1, arg2 ):
...     return arg1 + arg2
...
>>> def fun2( arg3, arg4 ):
...     x = fun1( arg3, 1)
...     y = fun1( arg4, 1)
...     return x + y
...
>>> print fun2(5,6)
13
```

BYOB ⟷ Python

Functions : Recursion!



BYOB ⟷ Python

Functions : Recursion!

```
>>> def sum( n ):
...     if ( n == 0 ):
...         return 0
...     else:
...         return n + sum( n - 1 )
...
>>> sum(5)
15
```

BYOB ⟷ Python

Functions : Recursion! Within Reason!

```
>>> sum(1234)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 5, in sum
  File "<stdin>", line 5, in sum
                    ⋮
  File "<stdin>", line 5, in sum
  File "<stdin>", line 5, in sum
  File "<stdin>", line 5, in sum
  File "<stdin>", line 5, in sum
RuntimeError: maximum recursion depth
>>>
```

BYOB ⟷ Python

Functions : Scoping

Script in Sprite

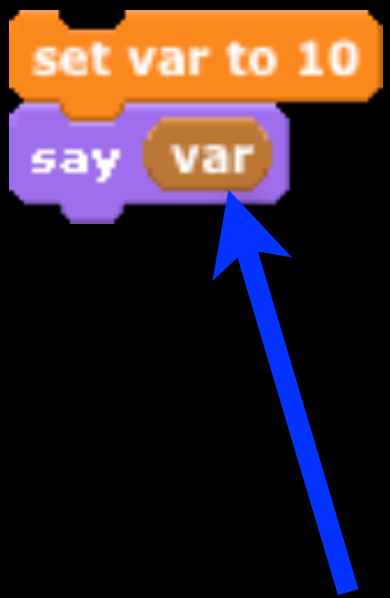
BYOB ⟷ Python

Functions : Scoping

What value of var will be "said?"

BYOB ⟷ Python

Functions : Scoping

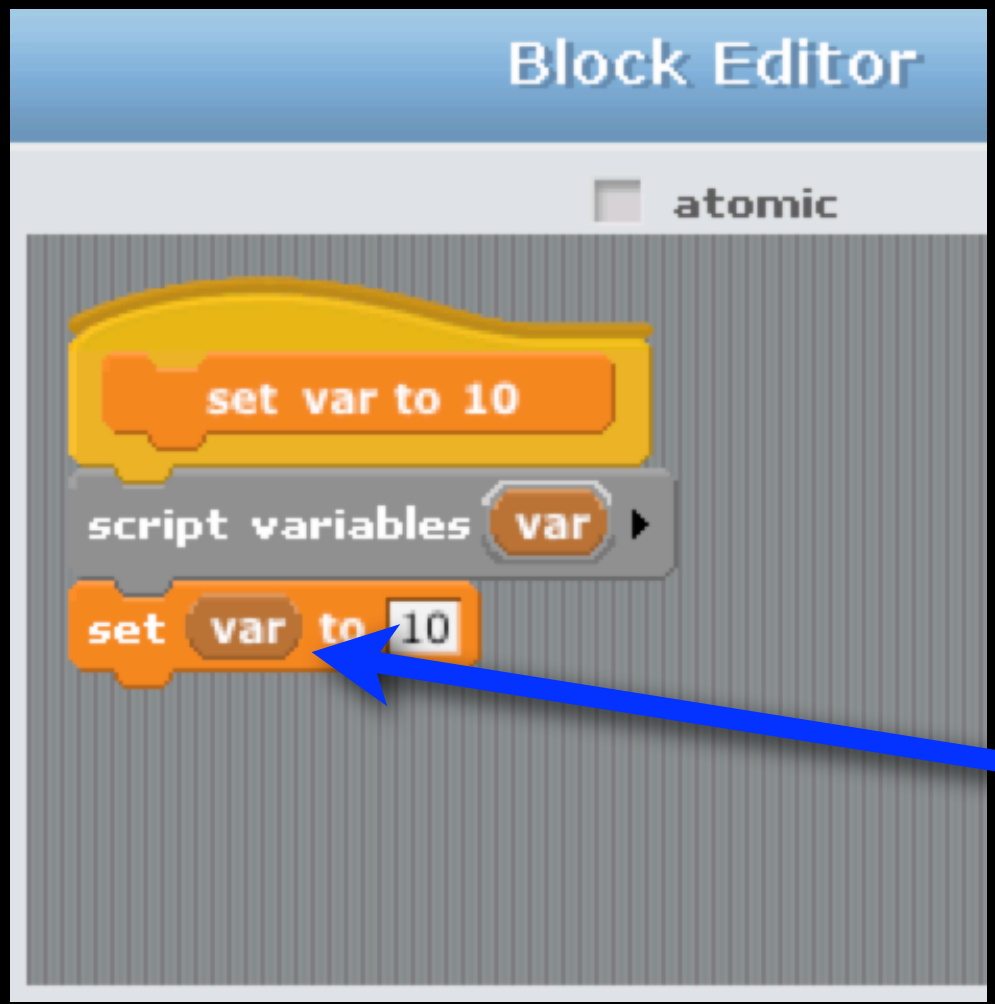


ERROR!

var doesn't exist (in this scope)!

BYOB ⟷ Python

Functions : Scoping



ERROR!

var doesn't exist (in this scope)!

BYOB ⟷ Python

Functions : Scoping

```
[/Users/headcrash]> python
Python 2.7.1 (r271:86882M, Nov 30 2010, 10:35:34)
[GCC 4.2.1 (Apple Inc. build 5664)] on darwin
Type "help", "copyright", "credits" or "license"
>>> print var
```

What value of var will be printed?

BYOB ⟷ Python

Functions : Scoping

```
>>> print var
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'var' is not defined
```

ERROR!

Don't worry about these lines for now...

BYOB ⟷ Python

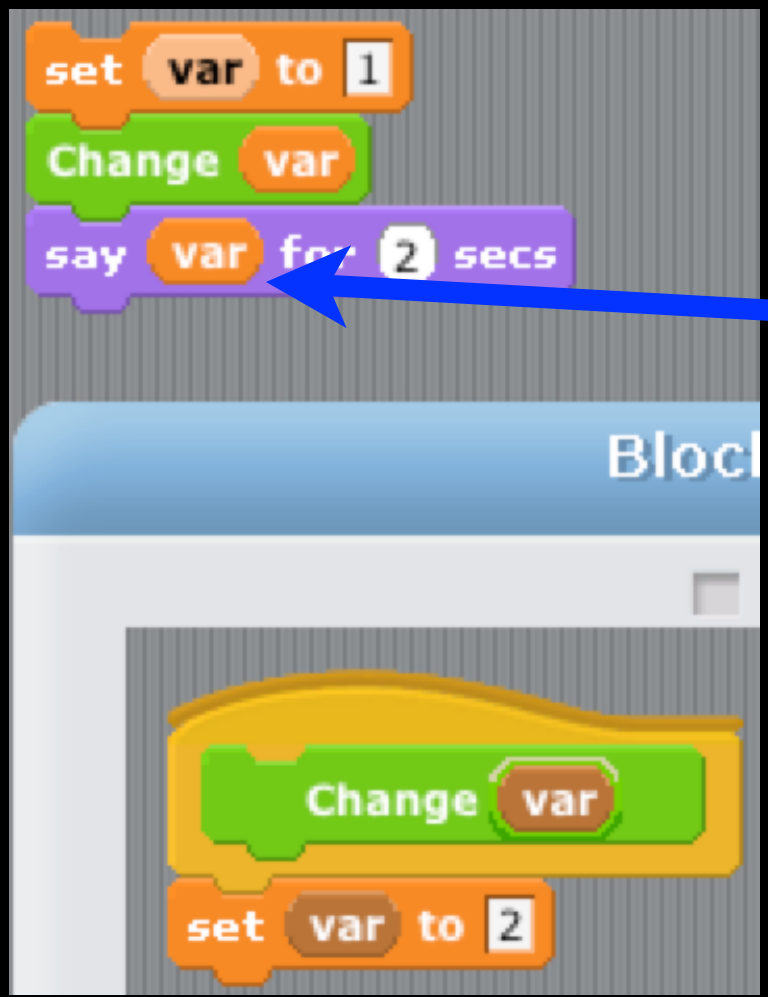
Functions : Scoping

Now `var` is declared, defined, AND initialized!

```
>>> var = 1
>>> print var
1
```

BYOB ⟷ Python

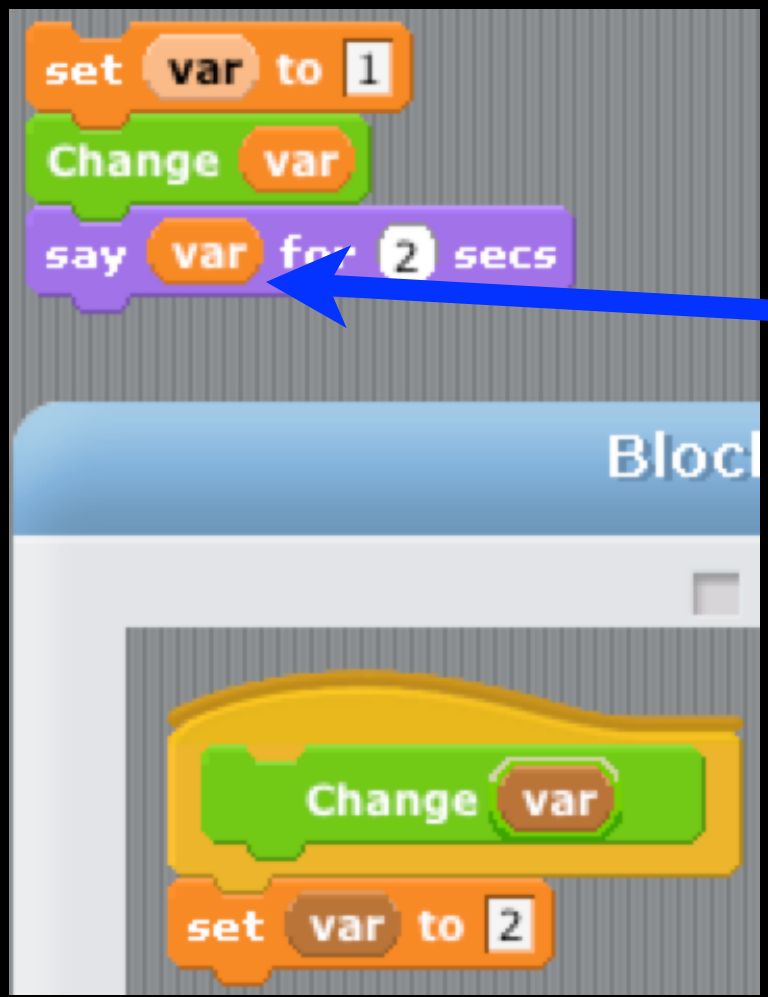
Functions : Scoping



What value of var will be "said"?

BYOB ⟷ Python

Functions : Scoping



BYOB ⟷ Python

Functions : Scoping

```
>>> def change(var):
...     var=2
...
>>> var=1
>>> change(var)
>>> print var
```

What value of var will be printed?

BYOB ⟷ Python

Functions : Scoping

```
>>> def change(var):
...     var=2
...
>>> var=1
>>> change(var)
>>> print var
1
```

BYOB ⟷ Python

Functions : Scoping

```
>>> def change(var):
...     var=2
...
>>> var=1
>>> change(var)
>>> print var
1
```

This is another very common source of bugs!

BYOB ⟷ Python

Functions : Scoping

```
>>> def change(var):
...     var=2
...     return var
...
>>> var=change(var)
>>> print var
2
```

This is more likely what you wanted...

BYOB ⟷ Python

Importing

```
>>> cos(1)
```

cosine(radians)

BYOB ⟷ Python

Importing

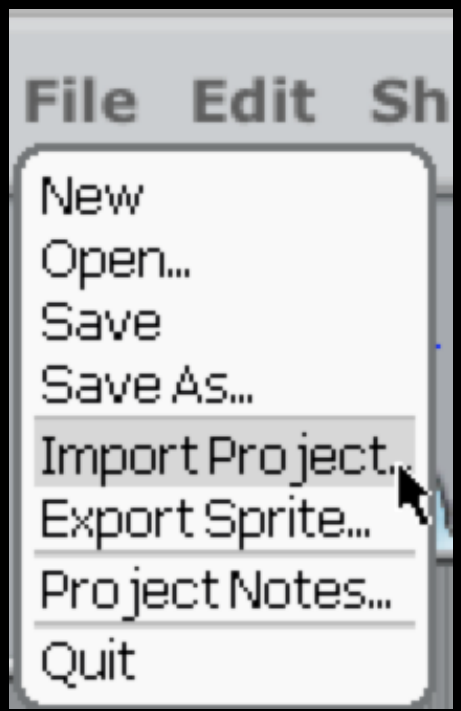
```
>>> cos(1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'cos' is not defined
```

ERROR!

Hmmmm....

BYOB ⟷ Python

Importing




```
>>> import math
```

"math" module

Beyond Blocks: Python #1

Importing

```
>>> import math
>>> math.cos(1)
0.5403023058681398
```

Beyond Blocks: Python #1

Importing

```
>>> import math
>>> math.cos(1)
0.540302305868139398
```

module.function(args)

Beyond Blocks: Python #1

Importing, help!

```
>>> help(math.cos)
```

Beyond Blocks: Python #1

Importing, help!

```
>>> help(math.cos)
```

module.function

Beyond Blocks: Python #1

Importing, help!

```
Help on built-in function cos in module math:

cos(...)
    cos(x)

    Return the cosine of x (measured in radians).
(END)
```

Beyond Blocks: Python #1

Help!

```
>>> help(math)
```

Beyond Blocks: Python #1

Help!

```
Help on module math:

NAME
    math

FILE
    /Library/Frameworks/Python.framework/Versions/2.7/lib/python2.7/lib-dynload/math.so

MODULE DOCS
    http://docs.python.org/library/math

DESCRIPTION
    This module is always available.  It provides access to the
    mathematical functions defined by the C standard.

FUNCTIONS
    acos(...)
        acos(x)

        Return the arc cosine (measured in radians) of x.

    acosh(...)
        acosh(x)
```

Beyond Blocks: Python #1

Help!

Python keyword

```
>>> help("import")

Related help topics: MODULES
```

Beyond Blocks: Python #1

Help!

```
>>> help("import")

Related help topics: MODULES
```

Note the quotes!

Beyond Blocks: Python #1

Help!

```
The ``import`` statement
************************

   import_stmt     ::= "import" module ["as" name] ( "," module ["as" name] )*
                   | "from" relative_module "import" identifier ["as" name]
                   ( "," identifier ["as" name] )*
                   | "from" relative_module "import" "(" identifier ["as" name]
                   ( "," identifier ["as" name] )* [","] ")"
                   | "from" module "import" "*"
   module          ::= (identifier ".")* identifier
   relative_module ::= "."* module | "."+
   name            ::= identifier

Import statements are executed in two steps: (1) find a module, and
initialize it if necessary; (2) define a name or names in the local
namespace (of the scope where the ``import`` statement occurs). The
statement comes in two forms differing on whether it uses the ``from``
keyword. The first form (without ``from``) repeats these steps for
each identifier in the list. The form with ``from`` performs step (1)
once, and then performs step (2) repeatedly.

To understand how step (1) occurs, one must first understand how
Python handles hierarchical naming of modules. To help organize
```

Beyond Blocks: Python #1

Sidebar: “sys” module

```
>>> import sys
>>> sys.getrecursionlimit()
1000
>>> sys.setrecursionlimit(2000)
>>> sum(1234)
761995
>>>
```

Beyond Blocks: Python #1

More Information

- **Python.org**: www.python.org
- **Python Docs**: www.python.org/doc/
- **Python Modules**: docs.python.org/modindex.html

Beyond Blocks: Python #1

More Information

- **Computer Science Circles: Python**

  cemclinux1.math.uwaterloo.ca/~cscircles/wordpress/using-this-website/

- **Dive Into Python**: diveintopython.org/toc/
- **Cal's Self-Paced Center**:

  inst.eecs.berkeley.edu/~selfpace/class/cs9h/

  How to Think Like a Computer Scientist (Python Version)

  www.greenteapress.com/thinkpython/thinkCSpy/html/