Besides Blocks: Python Session #1

Instructor: Dan Garcia

(thanks to Glenn Sugden for the first version of these slides)

Outline

- Computational Thinking (5)
- Codification (10)
- Python? (5)
    - Getting Started (10)
    - Why? (5)
- Syntax (20)
- Turtle (5)

The Goals of BJC

- BJC's goal is not to teach you Snap*!*
- It's to teach you critical thinking about societal implications of computing
- It's also to teach you how to program (Snap*!* is the best intro language we know) and help you succeed in CS61A
- More importantly it's to teach you how to think like a computer scientist in life, called "computational thinking"

Computational Thinking

- It’s using abstraction (removing detail and generalization with parameters)
- It’s understanding the value of a “spec” that specifies a contract
- It’s the iterative design cycle: design, prototype, implement, evaluate (loop)
- It’s thinking about how solutions scale, parallelize, generalize, and trying to foresee the unintended consequences!

Why Python? (1/2)

- Up until now, it's just been Snap*!*
  - There's an advantage in just one language, there's only one cognitive tax paid for "learning a new language"
- However, we want CS61A success too!
  - The feeling is this will help, it's in Python
  - We also see the benefit of another tool, and learning when you'd use Snap*!* vs Python…
- It's "Besides" blocks, not "Beyond" blocks!

Props to edX and BB teams!

(see website)

Codification

- Jens Moenig wrote Snap*!* with design support from Brian Harvey.
- Here is the coolest program ever.

tinyurl.com/BJCcodification

  - Lesson: Snap*!* is Turing Complete, so anything they can do, we can do.

Why Python (2/2)

- Easy to learn and use, looks like pseudocode
    - Minimal text-based syntax
        - Easy to cut & paste, people type faster than drag
    - Has a Turtle mode!
- Popular
    - lots of online support
    - Incredible libraries
    - Makes you marketable

Learning Python

- 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
- Plan: Lec, Lab, Dis, Lec, Lab

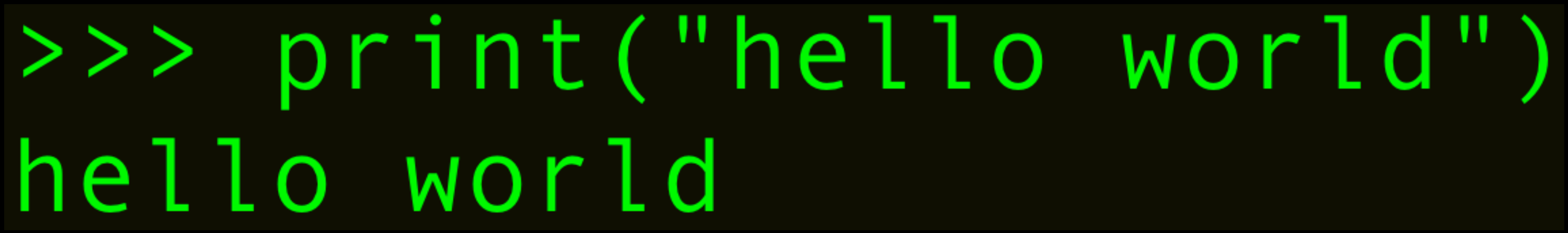
BJC Future: edX SPOC

- SPOC: "Small Private Online Course"
  - Hybrid MOOC
  - Think of MOOC = ebook
  - Teacher signs up class, picks parts they want
  - The forum discussions are self-contained
  - Teacher gets analytics of only their students
  - Teacher is in control
  - We're going to trial edX with the "Besides Blocks" labs

Beyond Blocks: Python #1

Installation: Mac Check

- Open Terminal
- Type “`python3`” 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:

  docs.python.org/3.4/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

```
unix% python -V
Python 3.4.0
```

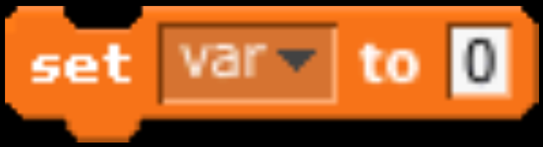
We'll be talking about version 3.4.0 in here.

If curious, there's more version info at:
https://docs.python.org/3.4/whatsnew/

Snap! ⟷ Python

Snap! ⟷ Python

Variables

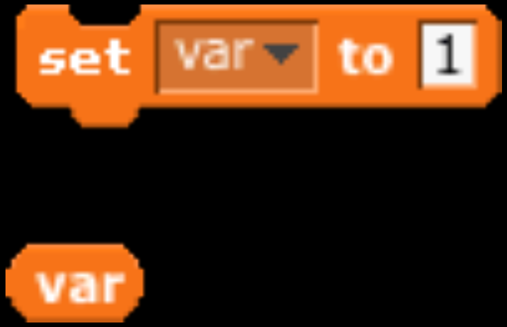


set var to 0

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

Snap! ↔ Python

Variables

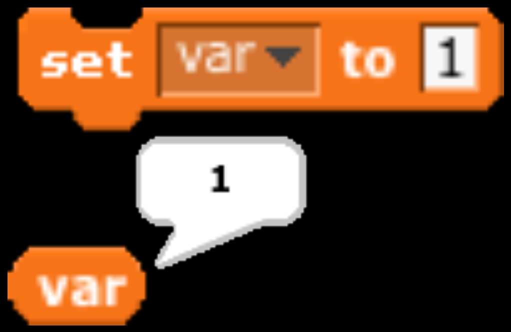


```
set var to 1

var
```

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

Snap! ⟷ Python

Variables



set var to 1

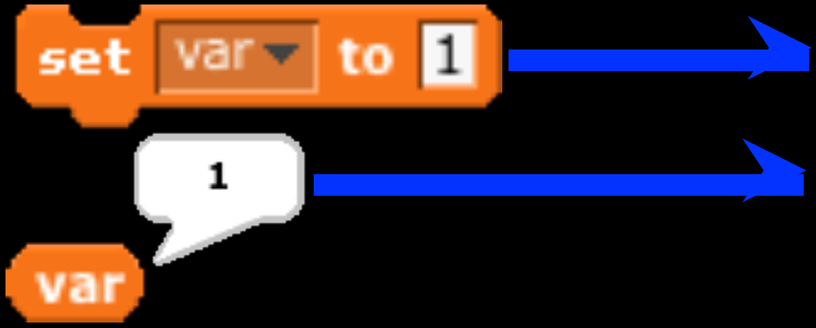
1

var

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

Snap! ↔ Python

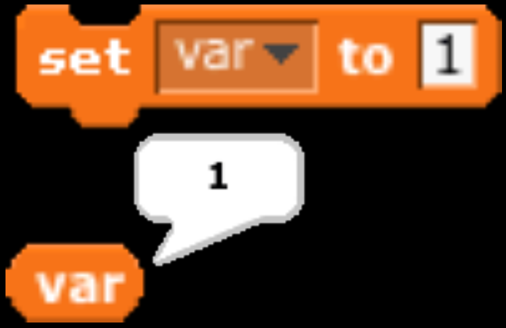
Variables



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

Snap! ⟷ Python

Variables

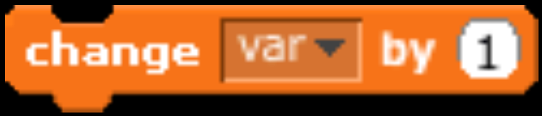


```
>>> var=1
>>> var
1
```

NOTE:
Assignment doesn’t “evaluate” to anything, so nothing is printed!

Snap! ⟷ Python

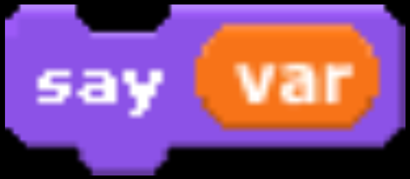
Variables


change var by 1

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

Snap! ↔ Python

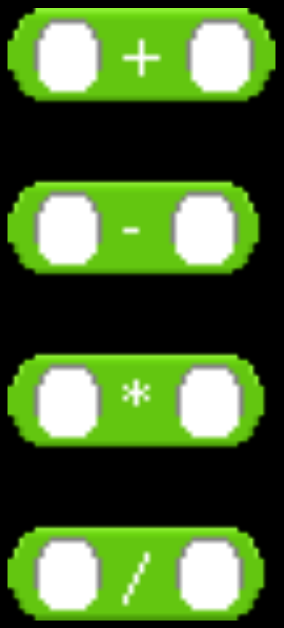
Variables



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

Snap! ↔ Python

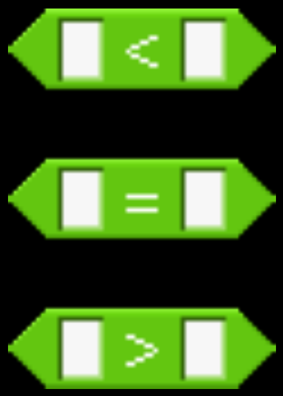
Operators

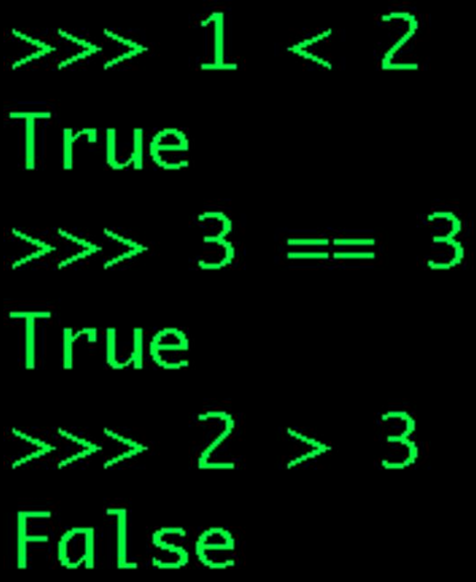


```
>>> 1+1
2
>>> 2-1
1
>>> 2*2
4
>>> 6/2
3
```

Snap! ↔ Python

Operators

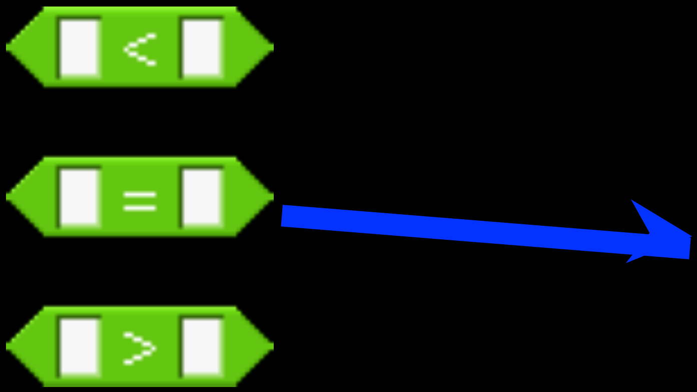




```
>>> 1 < 2
True
>>> 3 == 3
True
>>> 2 > 3
False
```

Snap! ⟷ Python

Operators

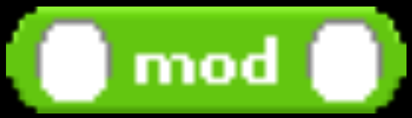


```
>>> 1 < 2
True
>>> 3 == 3
True
>>> 2 > 3
False
```

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

Snap! ↔ Python

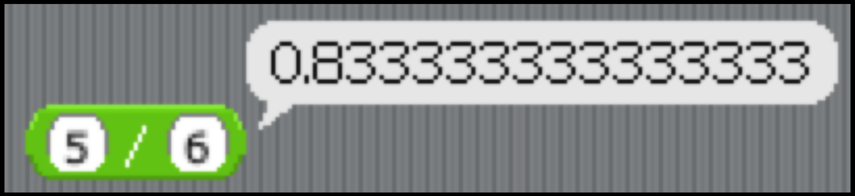
Operators



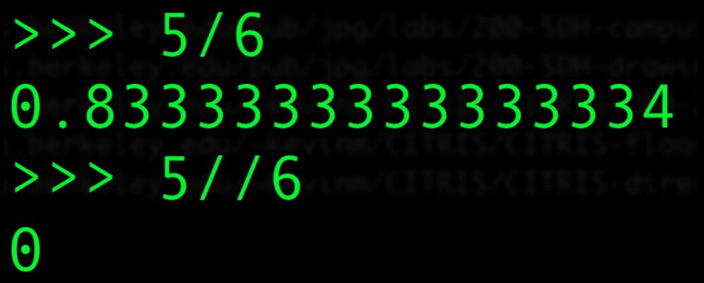
```
>>> 3 % 2
1
>>> 12345 % 678
141
```

Snap! ↔ Python

Sidebar: Division (integer vs. real/float)



```
0.8333333333333333
```

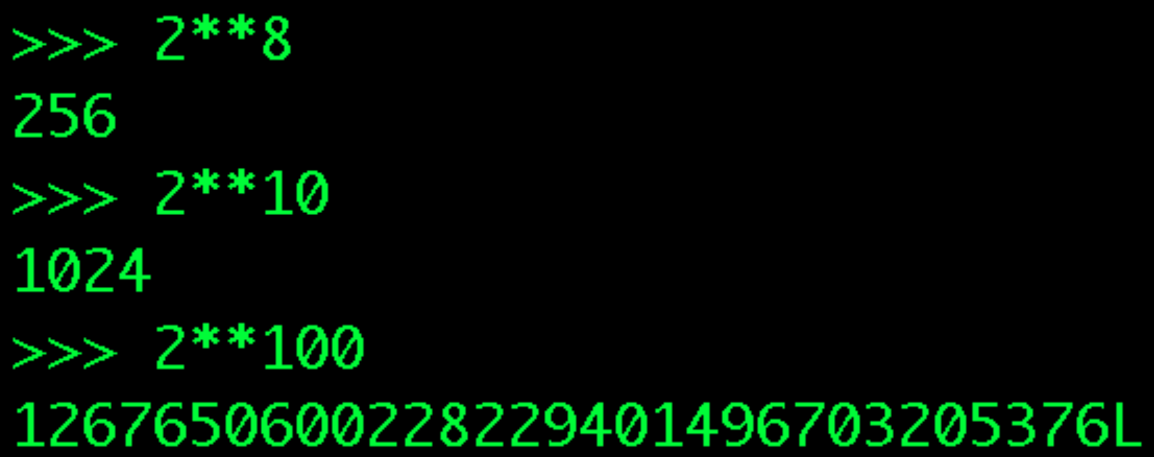


```
>>> 5/6
0.8333333333333334
>>> 5//6
0
```

Snap! ↔ Python

Sidebar: Exponent

Snap! has e^x and 10^x,
but Python can do any base & exponent!

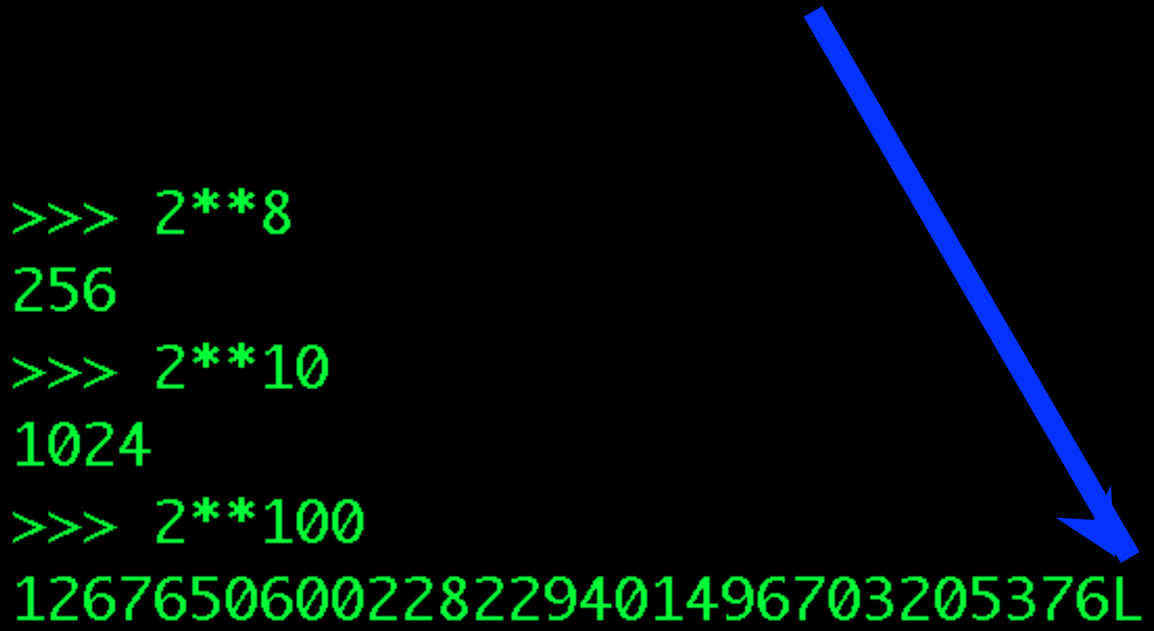


```
>>> 2**8
256
>>> 2**10
1024
>>> 2**100
1267650600228229401496703205376L
```

Snap! ⟷ Python

Sidebar: Exponent

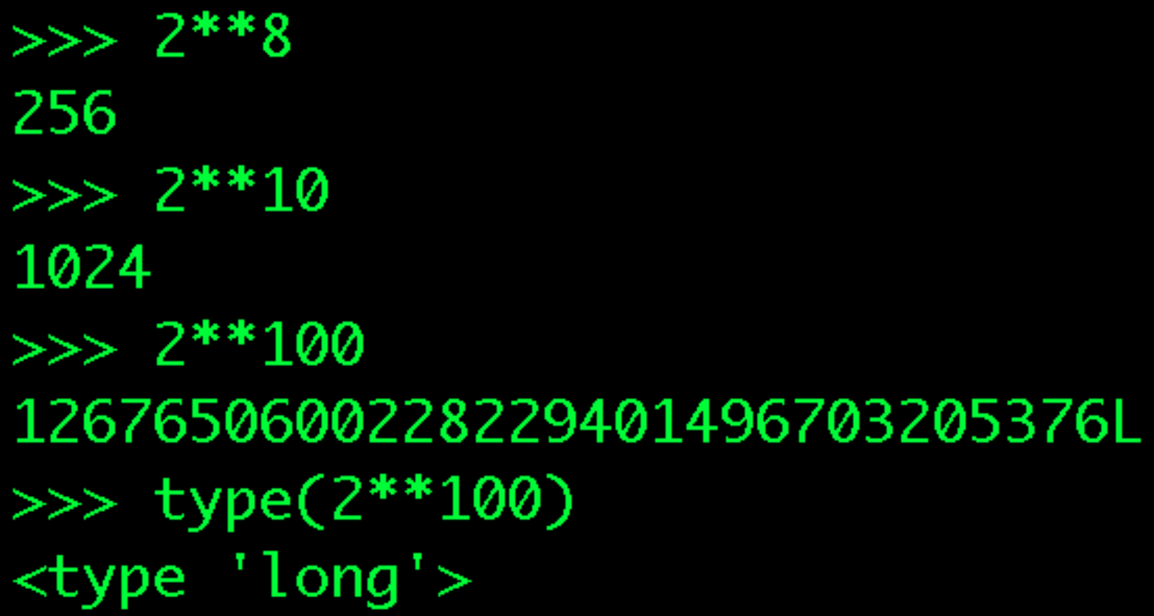
What's that "L?"



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

Snap! ⟷ Python

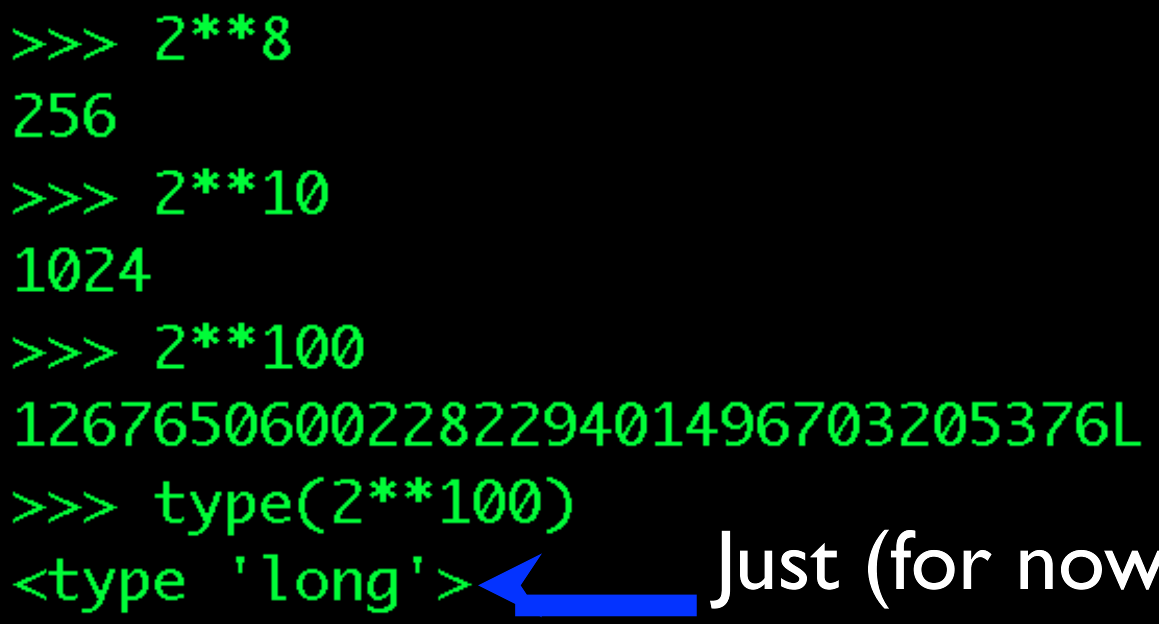
Sidebar: Exponent



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

Snap! ⟷ Python

Sidebar: Exponent

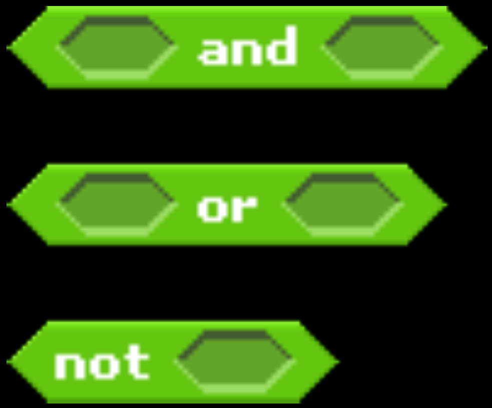


```
>>> 2**8
256
>>> 2**10
1024
>>> 2**100
1267650600228229401496703205376L
>>> type(2**100)
<type 'long'>
```

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

Snap! ⟷ Python

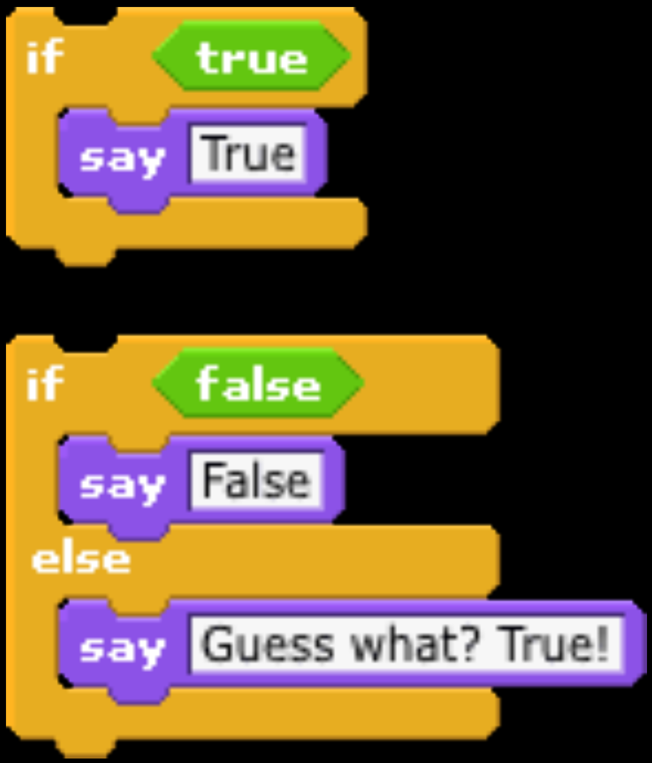
Operators



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

Snap! ⟷ Python

Conditionals

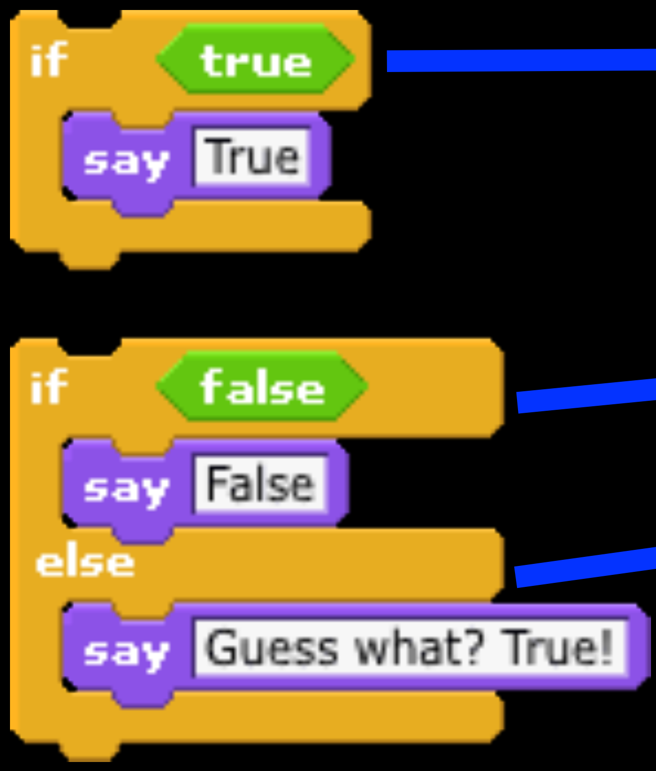


```
if true
  say True

if false
  say False
else
  say Guess what? True!
```

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

Snap! ⟷ Python

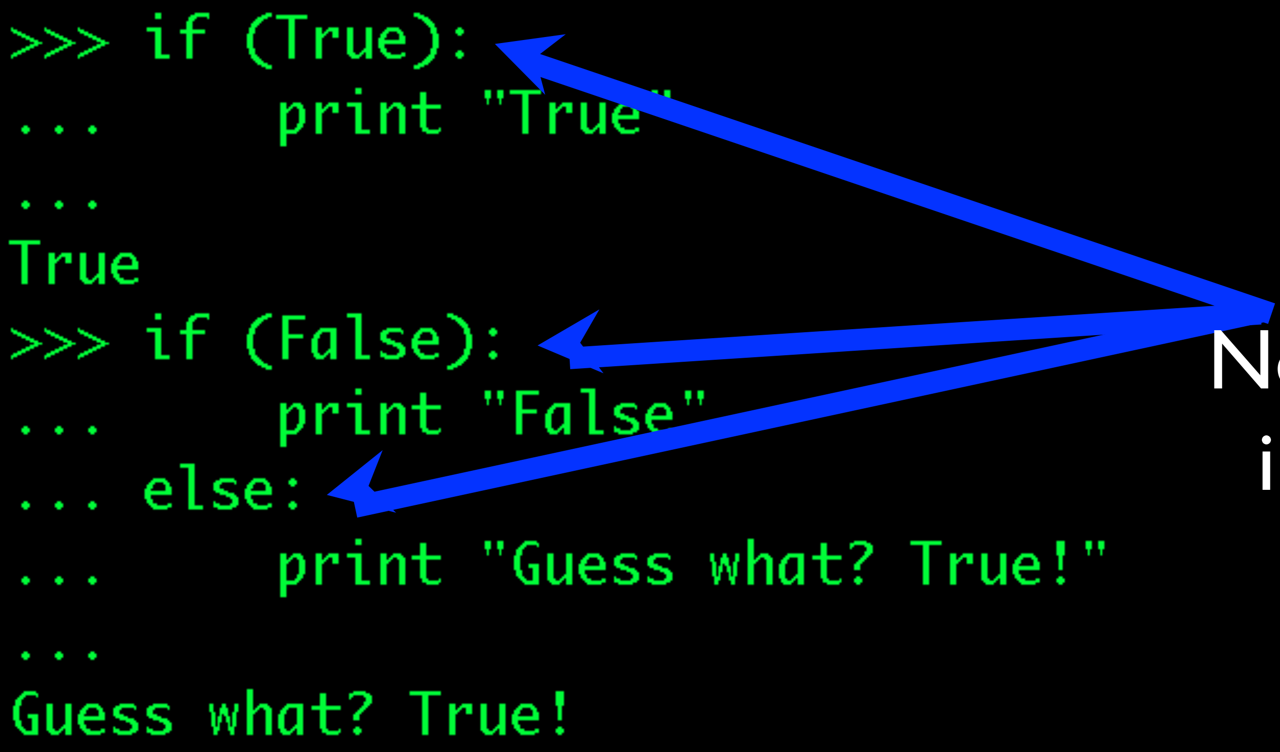
Conditionals



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

Snap! ↔ Python

Conditionals

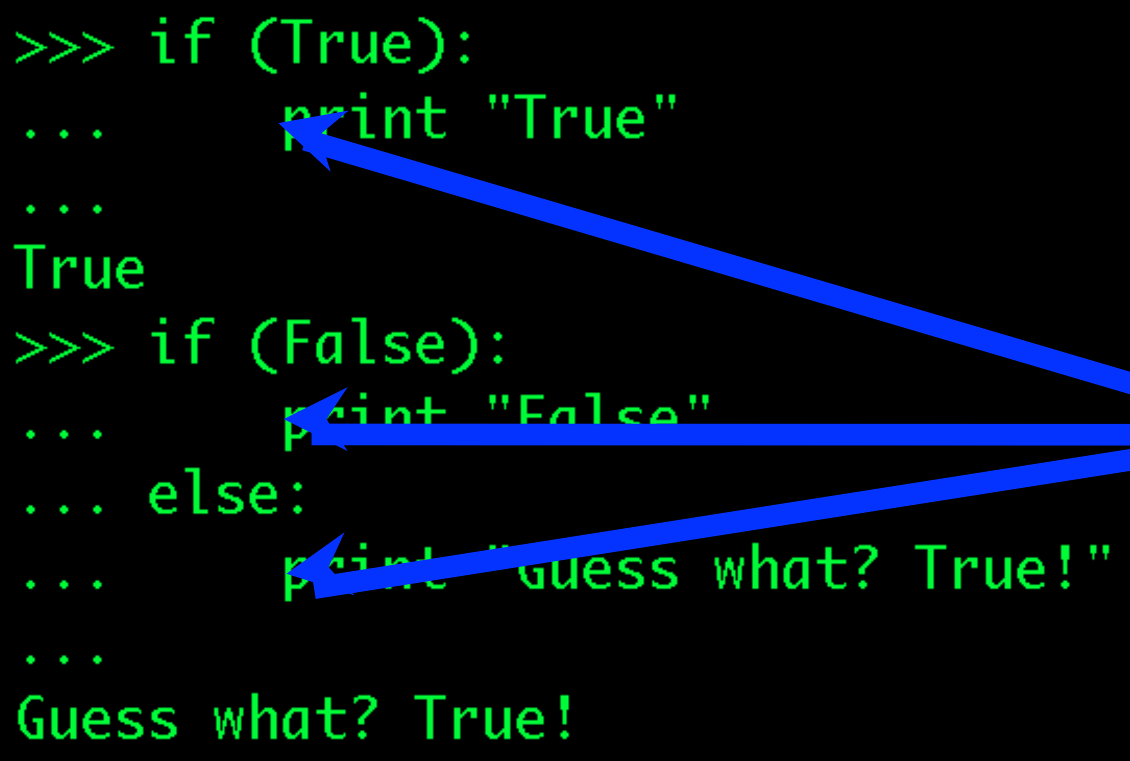


```
>>> if (True):
...     print "True"
...
True
>>> if (False):
...     print "False"
... else:
...     print "Guess what? True!"
...
Guess what? True!
```

Notice the colon and indentation *syntax*!

Snap! ⟷ Python

Conditionals

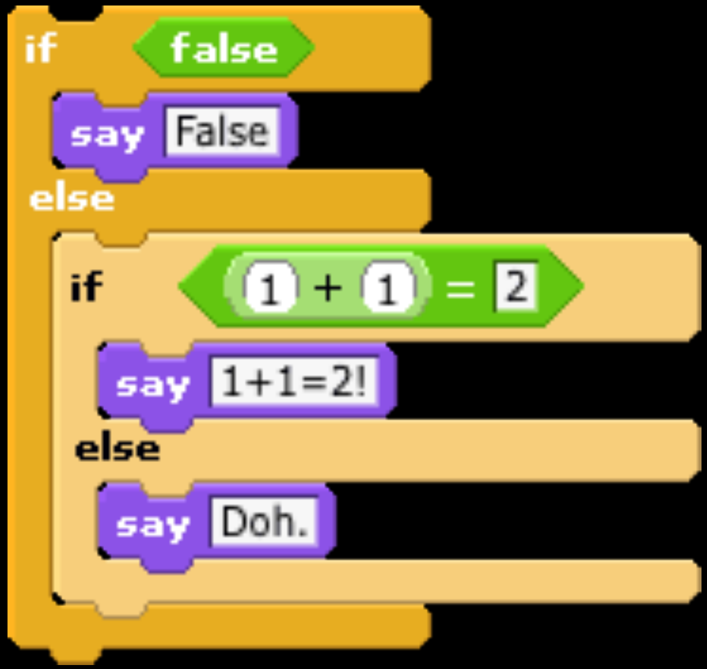


```
>>> if (True):
...      print "True"
...
True
>>> if (False):
...      print "False"
... else:
...      print "Guess what? True!"
...
Guess what? True!
```

Notice the colon and indentation *syntax*!

Snap! ⟷ Python

Conditionals

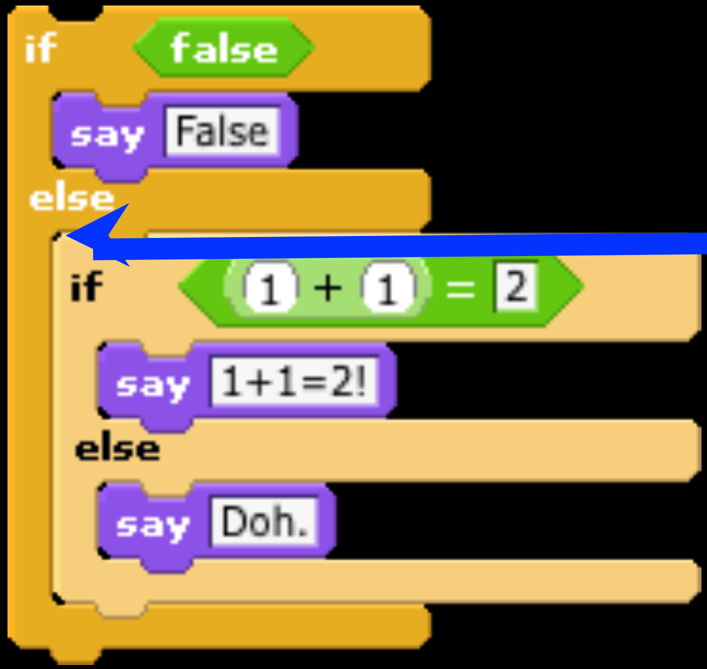


```
if false
  say False
else
  if (1 + 1) = 2
    say 1+1=2!
  else
    say Doh.
```

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

Snap! ↔ Python

Conditionals

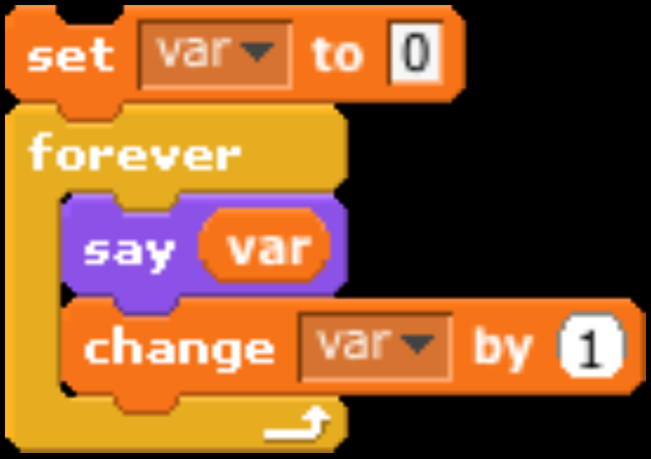


```
if false
    say False
else
    if (1 + 1) = 2
        say 1+1=2!
    else
        say Doh.
```

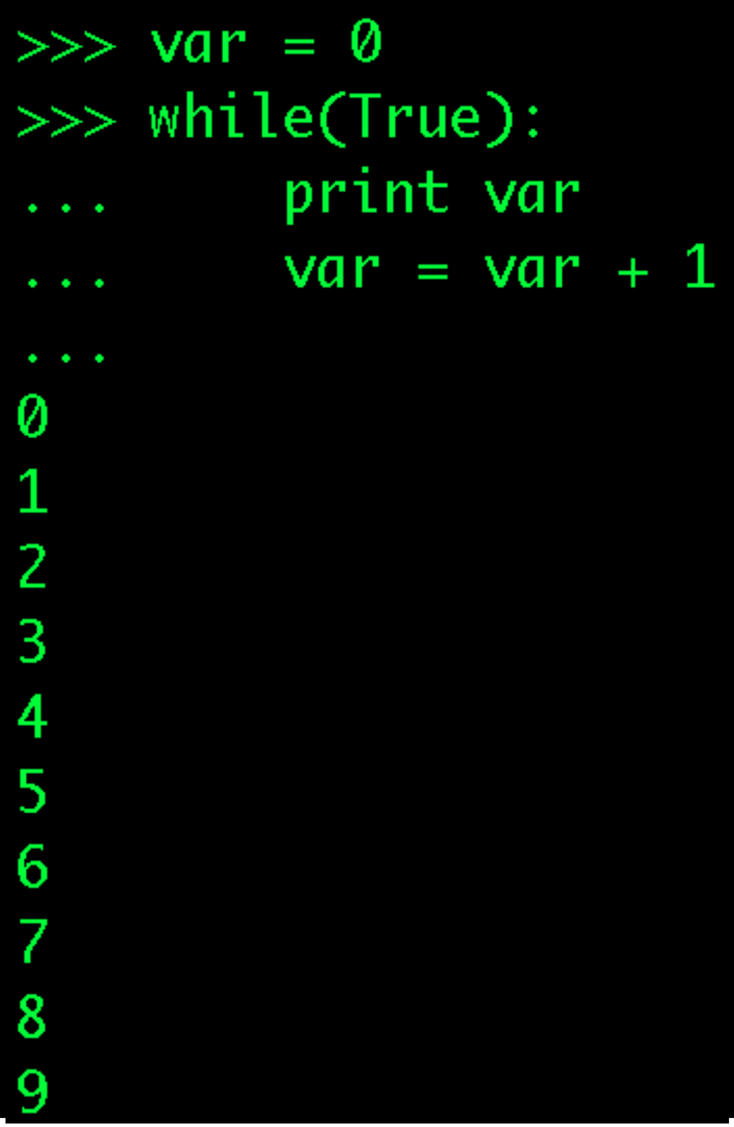
```
>>> if (False):
...     print "False"
... elif (1+1==2):
...     print "1+1==2!"
... else:
...     print "Doh."
...
1+1==2!
```

Snap! ↔ Python

Loops



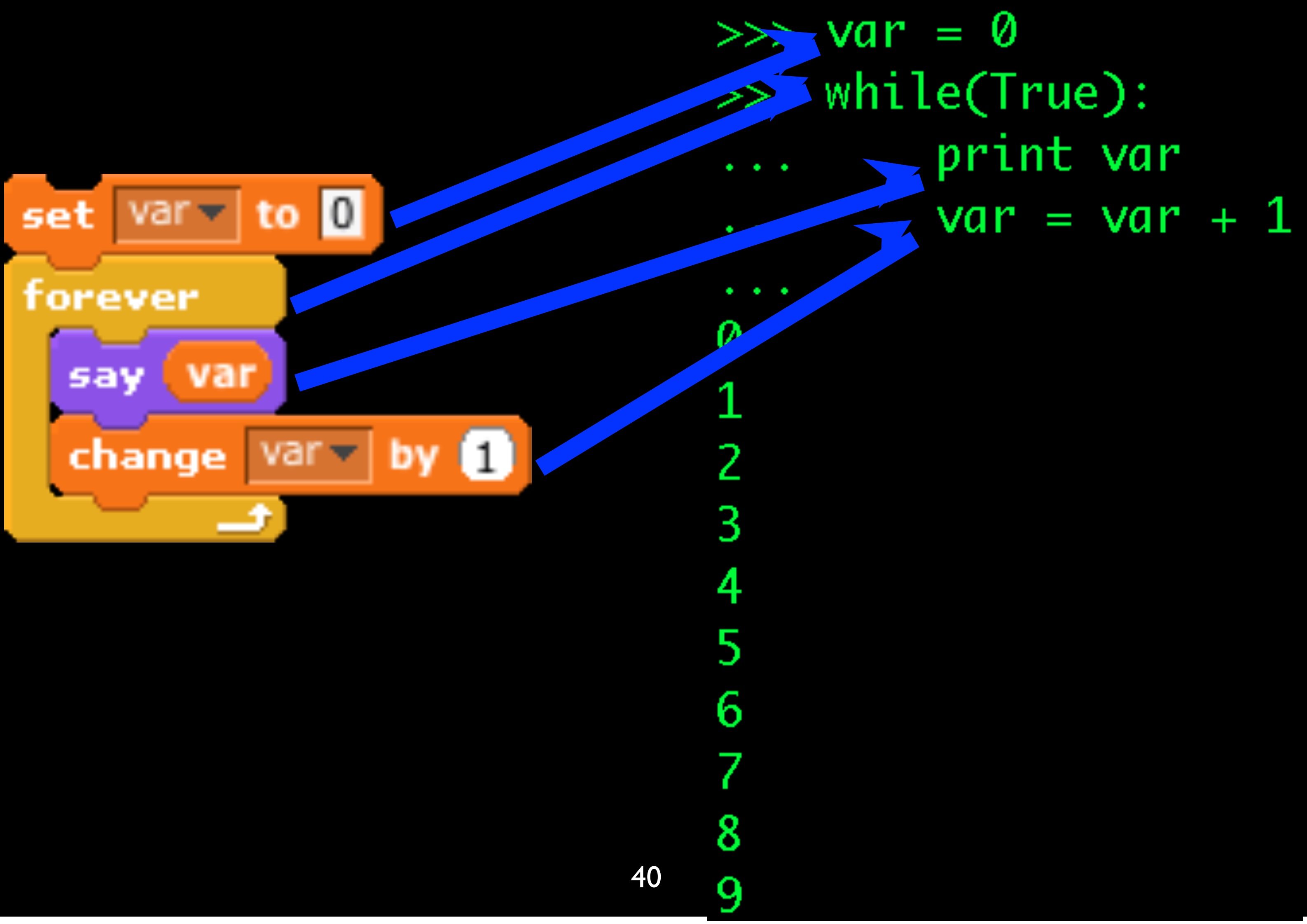
```
set var to 0
forever
    say var
    change var by 1
```



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

Snap! ↔ Python

Loops

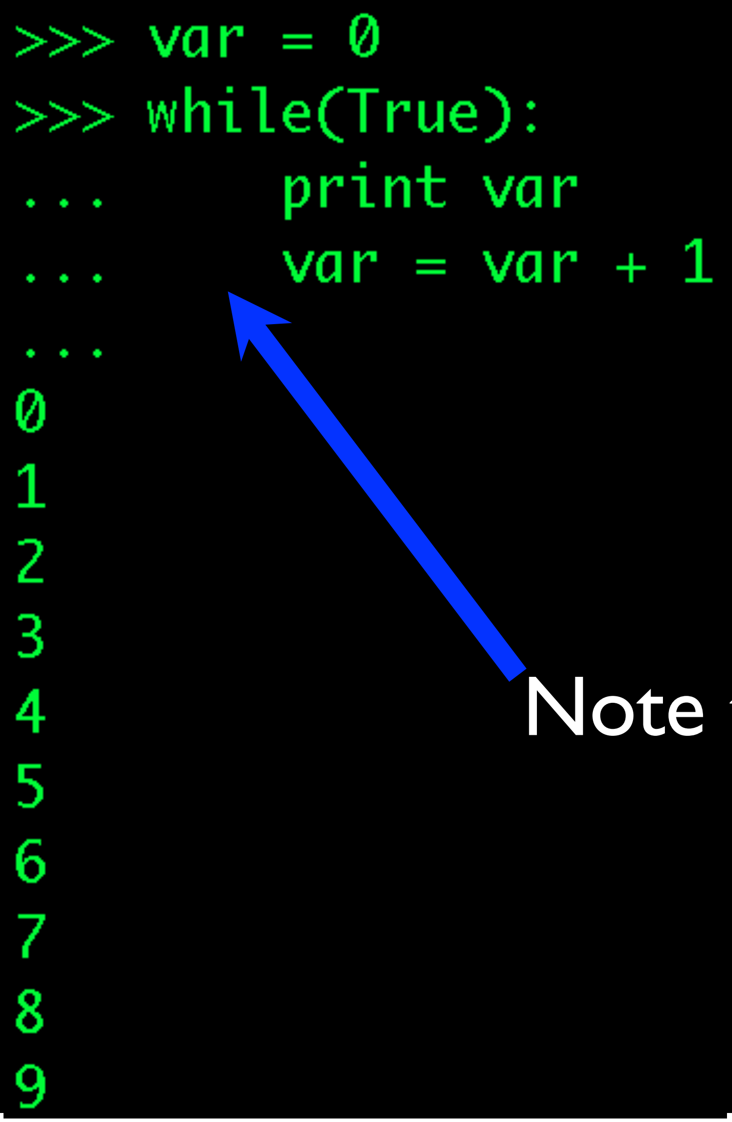


```
set var to 0
forever
    say var
    change var by 1
```

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

Snap! ↔ Python

Loops

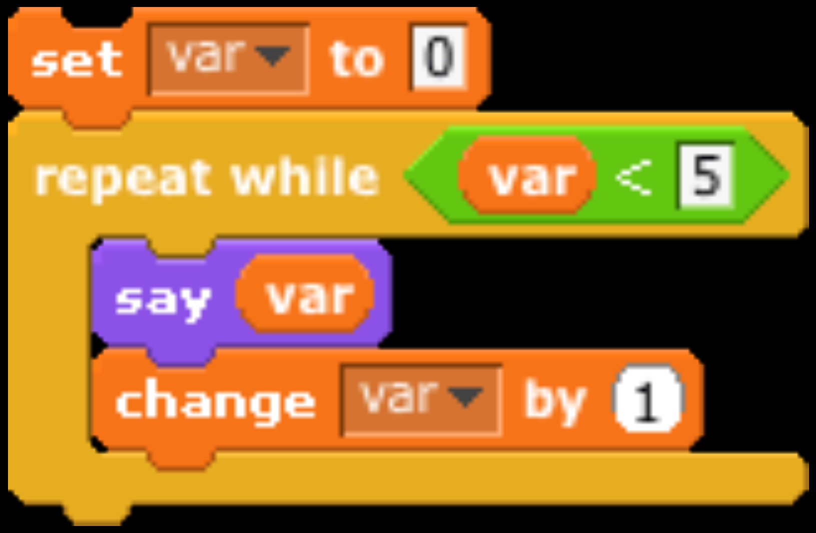


```
>>> var = 0
>>> while(True):
...     print var
...     var = var + 1
...
0
1
2
3
4
5
6
7
8
9
```

Note the indentation (again)!

Snap! ↔ Python

Loops

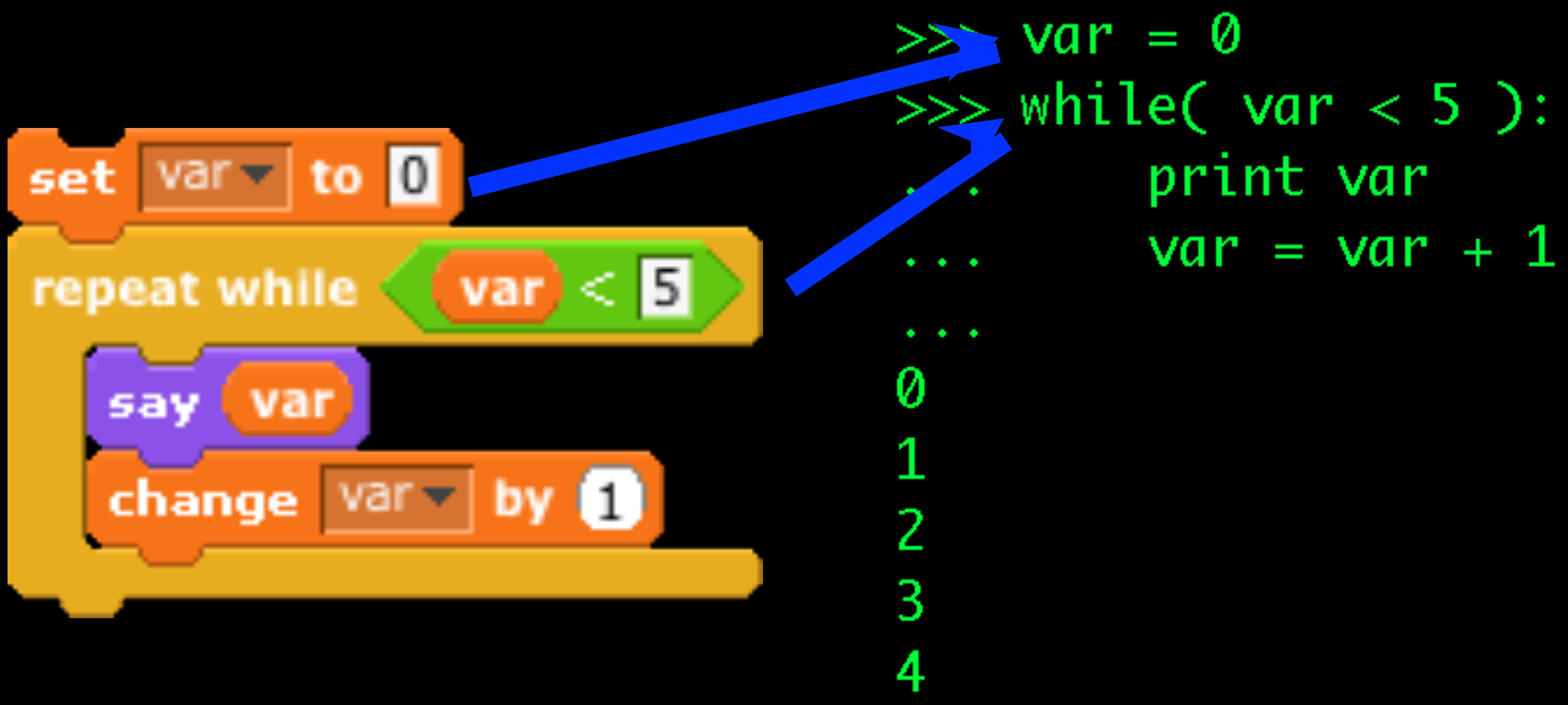


```
set var to 0
repeat while var < 5
    say var
    change var by 1
```

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

Snap! ⟷ Python

Loops

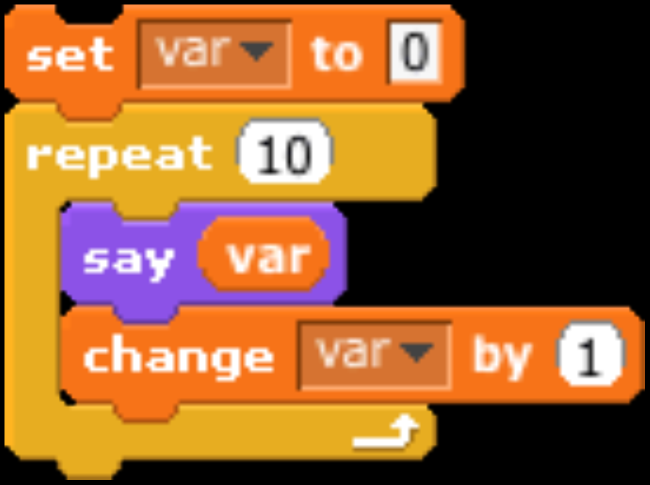


```
set var to 0
repeat while var < 5
    say var
    change var by 1
```

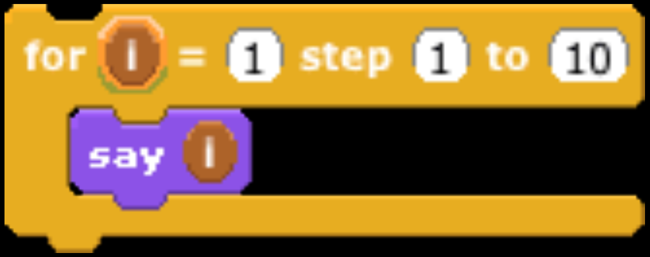
```
>>> var = 0
>>> while( var < 5 ):
...     print var
...     var = var + 1
...
0
1
2
3
4
```

Snap! ↔ Python

More Loops



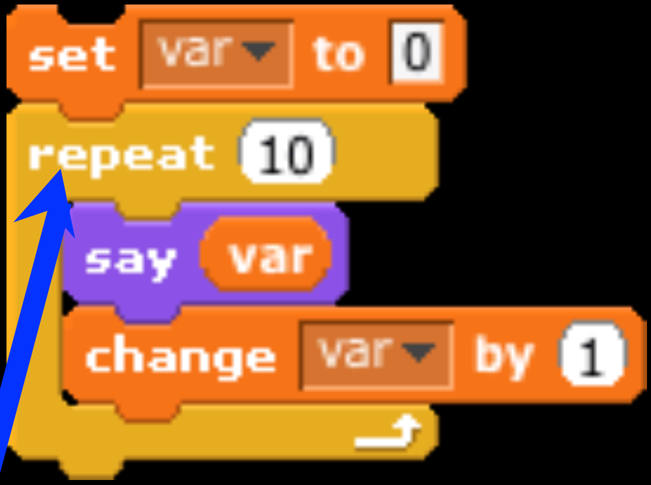
```
set var to 0
repeat 10
    say var
    change var by 1
```



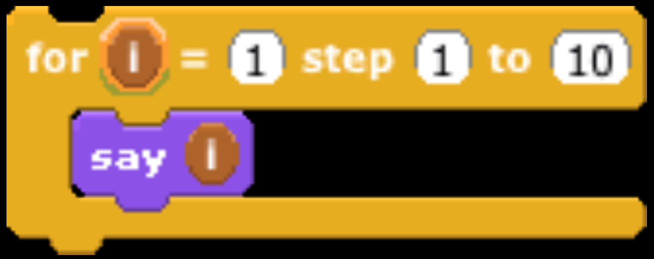
```
for i = 1 step 1 to 10
    say i
```

Snap! ⟷ Python

More Loops



```
set var to 0
repeat 10
    say var
    change var by 1
```



```
for i = 1 step 1 to 10
    say i
```

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

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

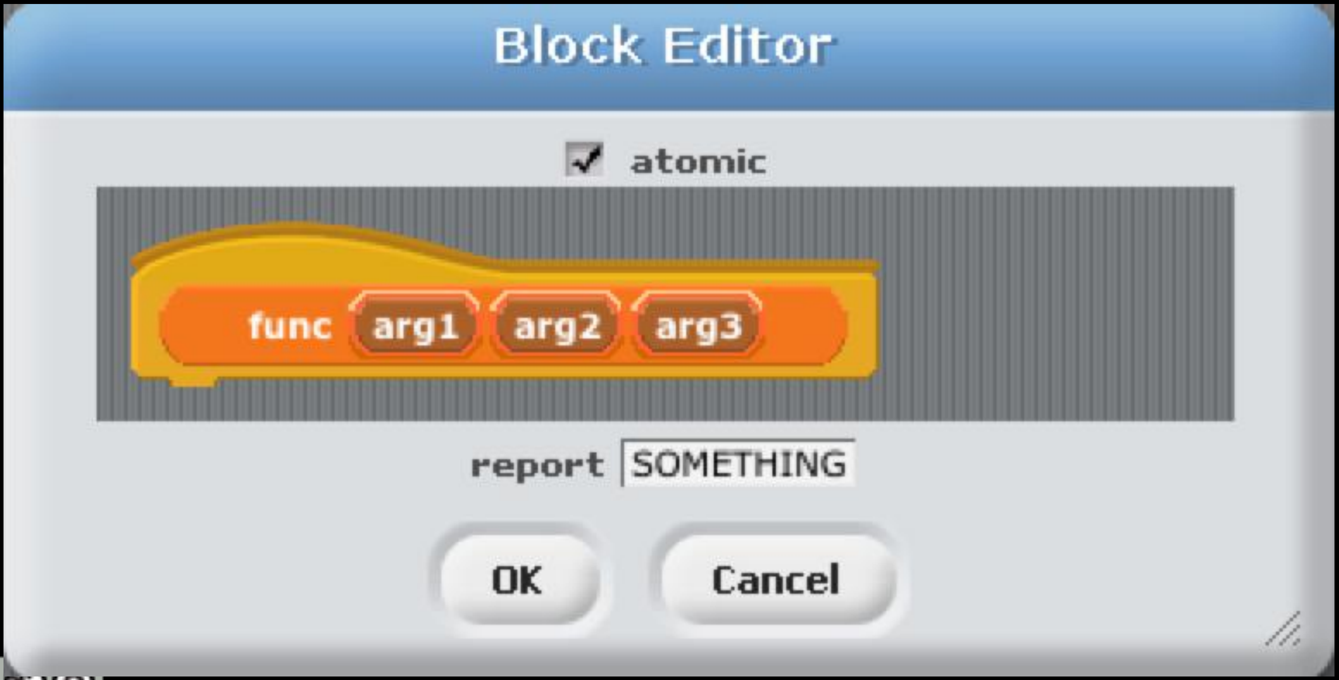
Snap! ↔ Python

Functions: Calling

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

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

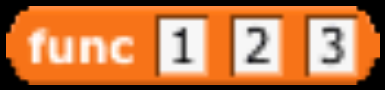
- Equivalent to creating & running a Snap! block:



Block Editor

☑ atomic

report SOMETHING



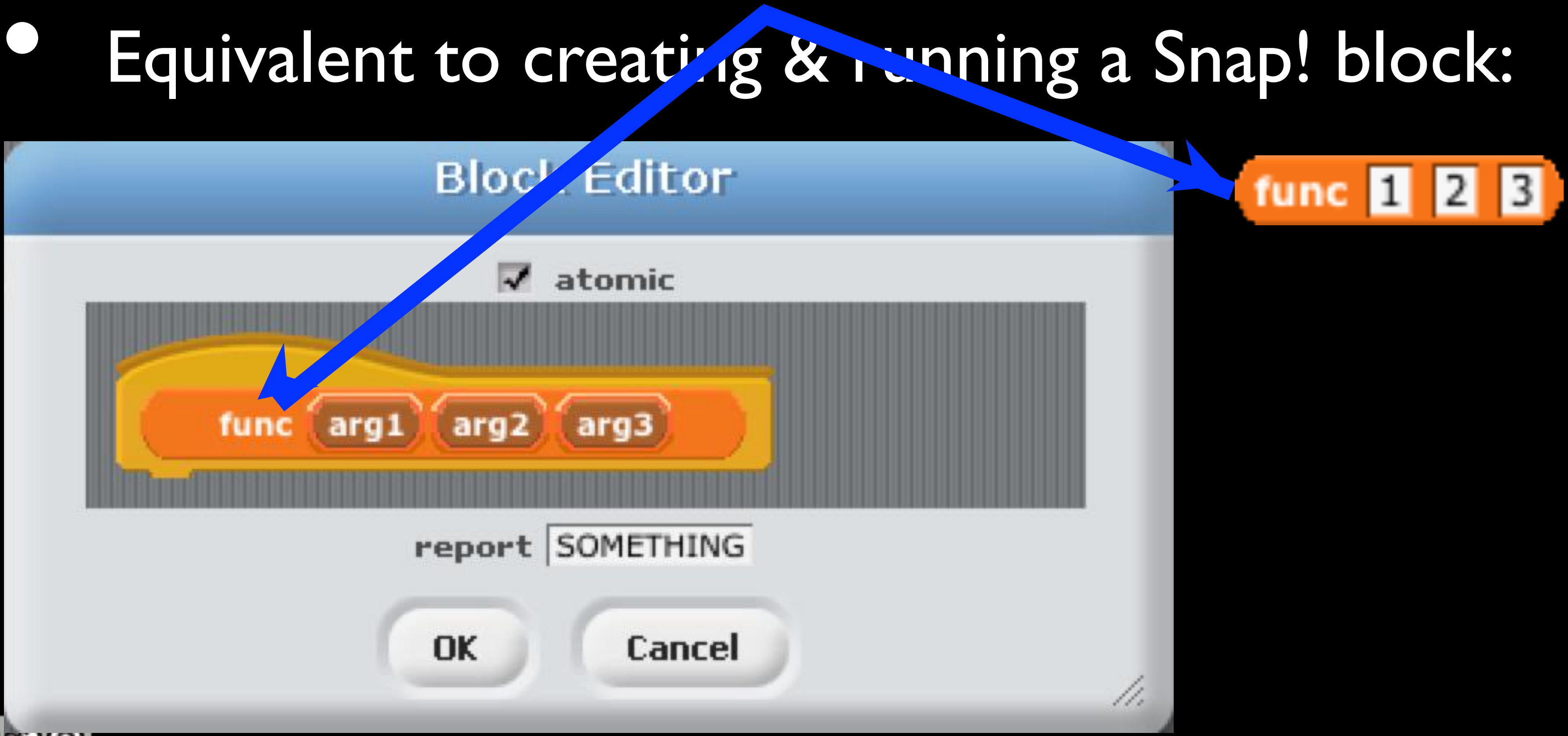
Snap! ↔ Python

Functions: Calling

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

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

- Equivalent to creating & running a Snap! block:



☑ atomic

report SOMETHING

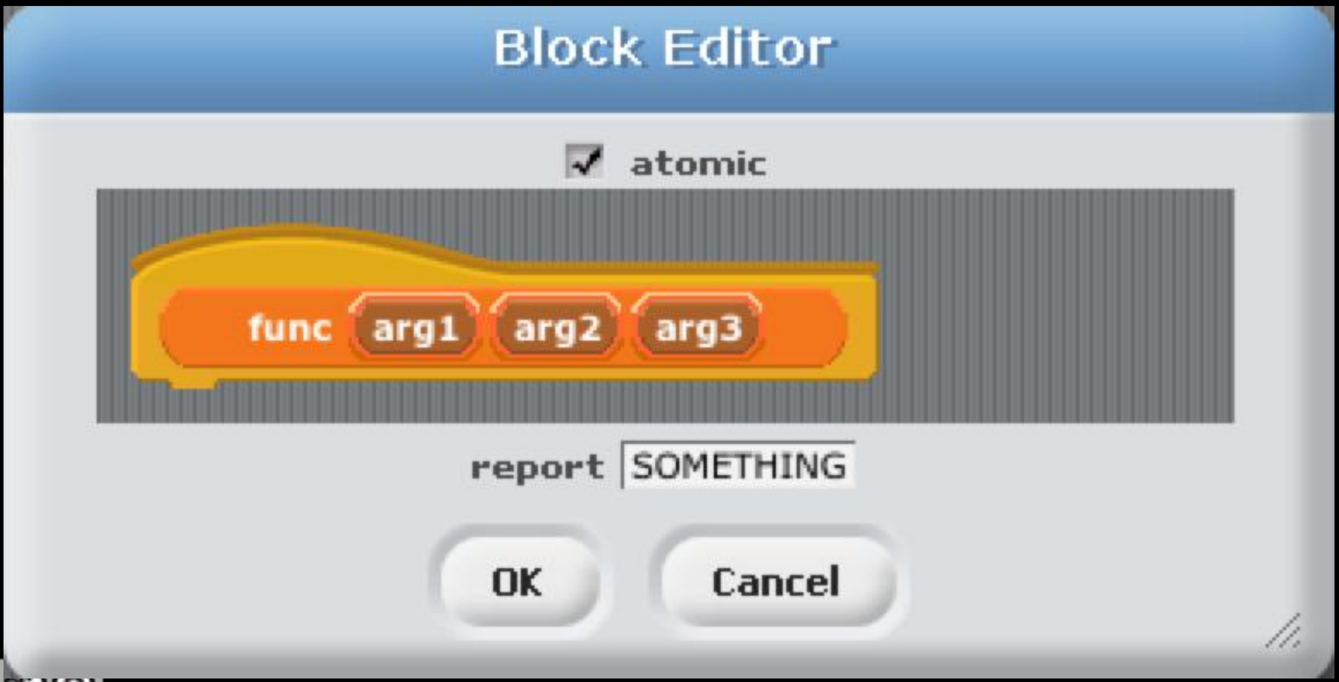
Snap! ↔ Python

Functions: Calling

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

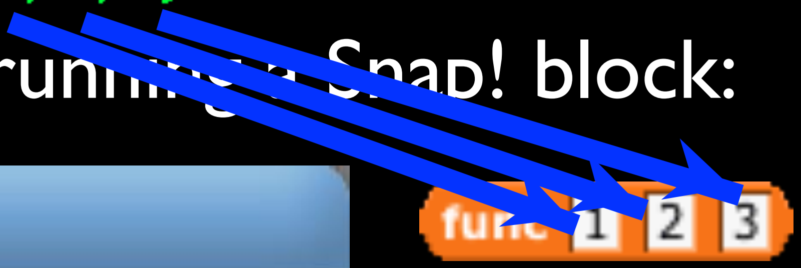
```
>>> func(1,2,3)
```

- Equivalent to creating & running a Snap! block:



☑ atomic

report SOMETHING



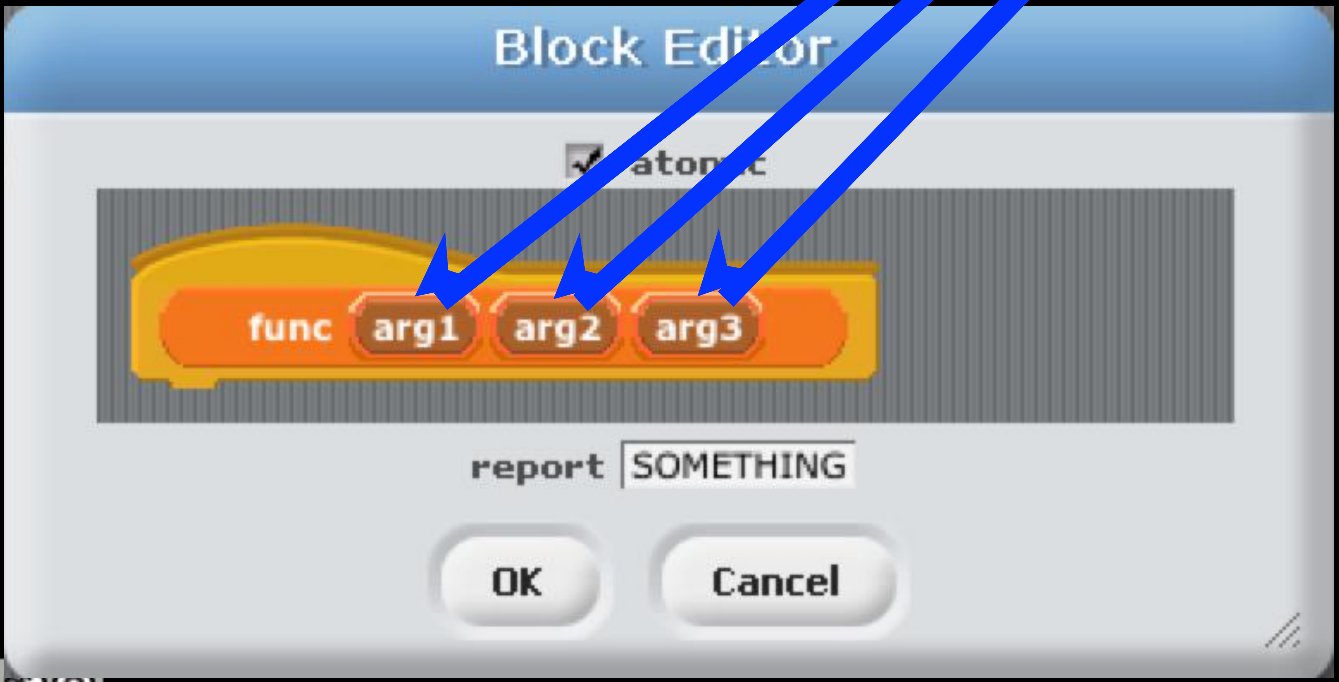
Snap! ↔ Python

Functions: Calling

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

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

- Equivalent to creating & running a Snap! block:

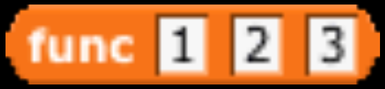


Block Editor

☑ atomic

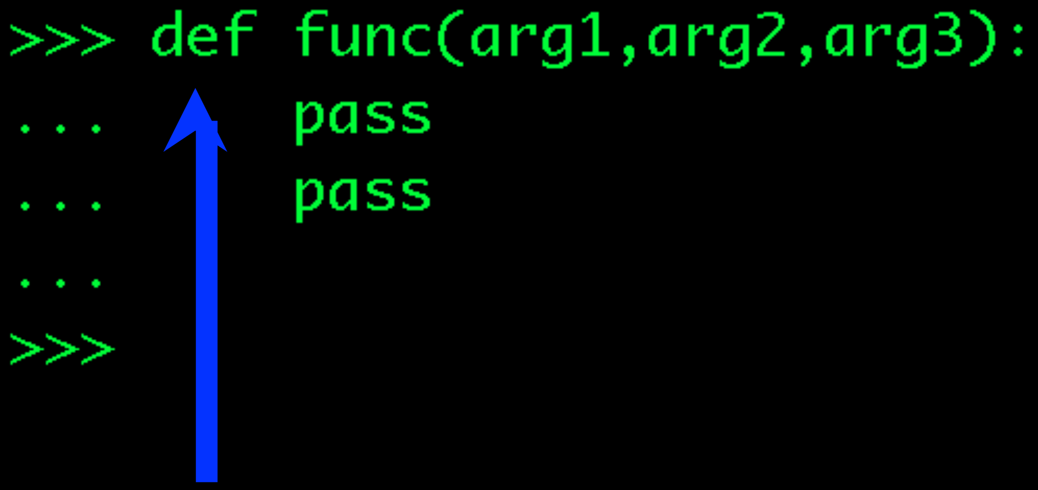
func arg1 arg2 arg3

report SOMETHING



Snap! ↔ Python

Functions : Defining

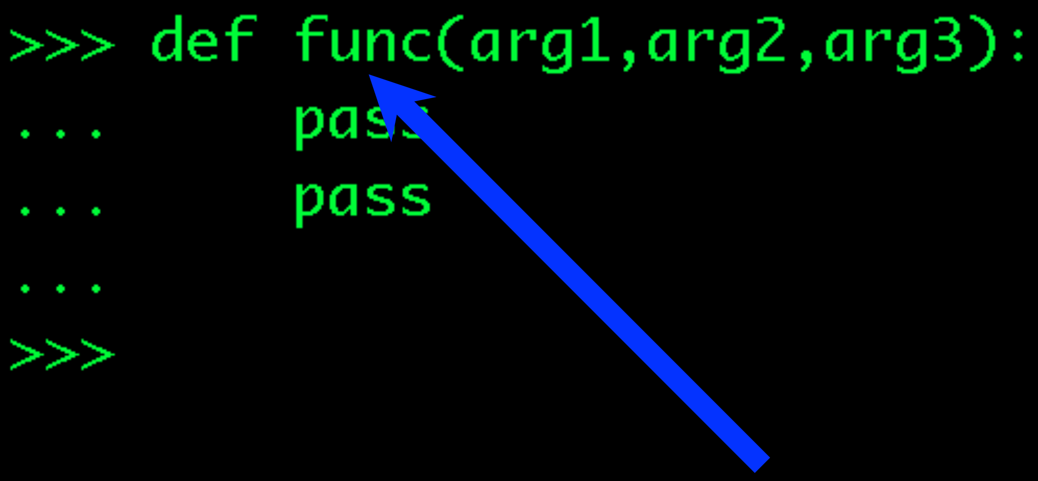


```
>>> def func(arg1,arg2,arg3):
...     pass
...     pass
...
>>>
```

Keyword: DEF

Snap! ↔ Python

Functions : Defining

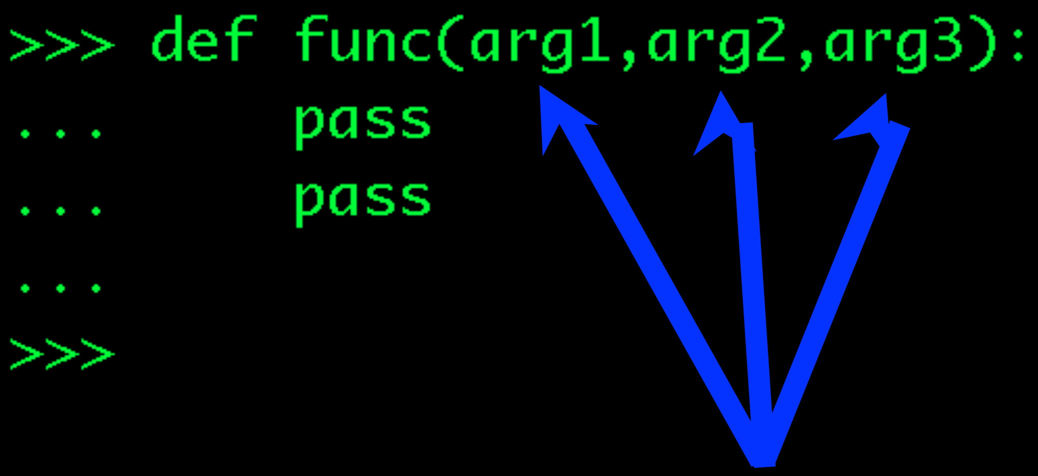


```
>>> def func(arg1,arg2,arg3):
...         pass
...         pass
...
>>>
```

Name of the function

Snap! ↔ Python

Functions : Defining

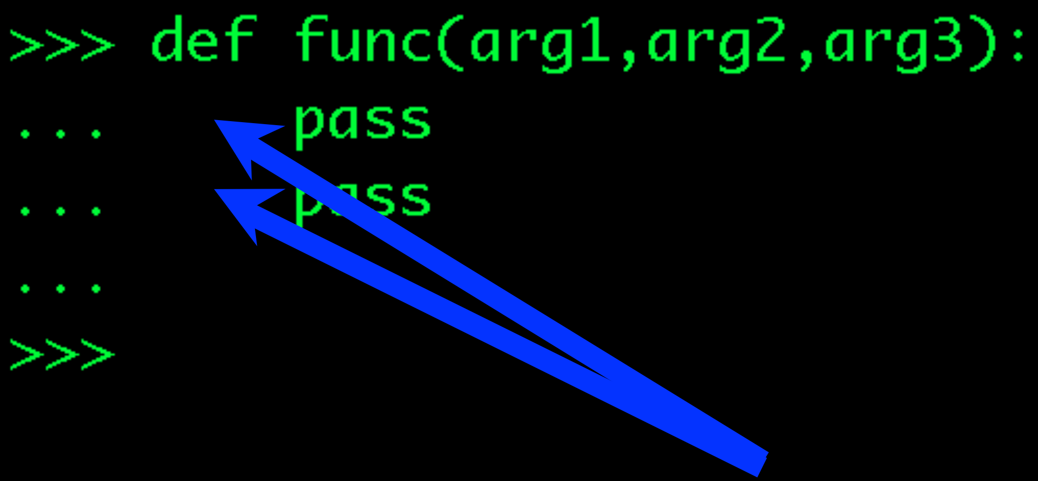


```
>>> def func(arg1,arg2,arg3):
...     pass
...     pass
...
>>>
```

“Arguments,” or inputs to the function

Snap! ↔ Python

Functions : Defining



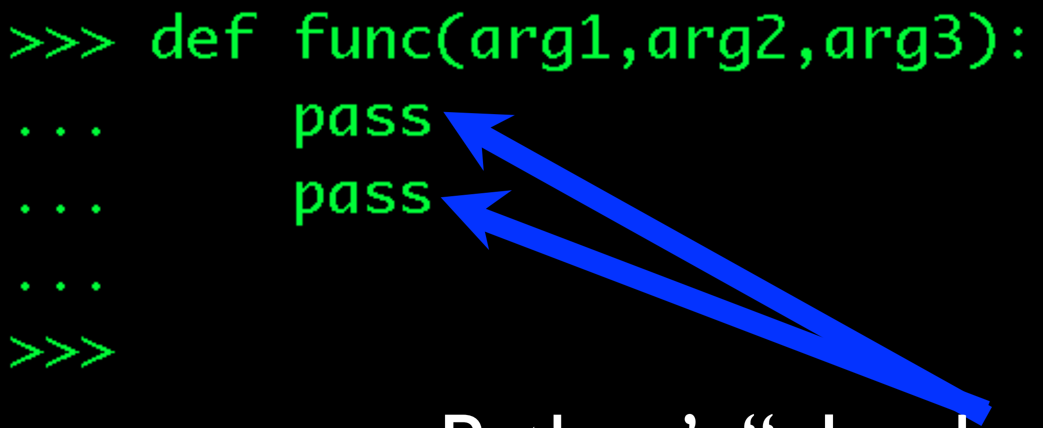
```
>>> def func(arg1,arg2,arg3):
...     pass
...     pass
...
>>>
```

Indentation: the key to “scope.”

We’ll talk about “scope” later...

Snap! ↔ Python

Functions : Defining



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

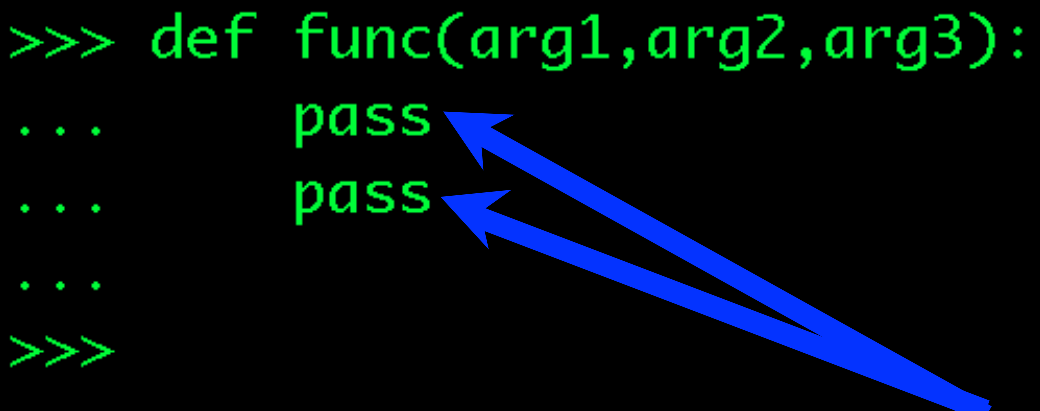
pass: Python’s “placeholder” or NOP

NOP: short for “NO OPeration”

(or do nothing...)

Snap! ↔ Python

Functions : Defining



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

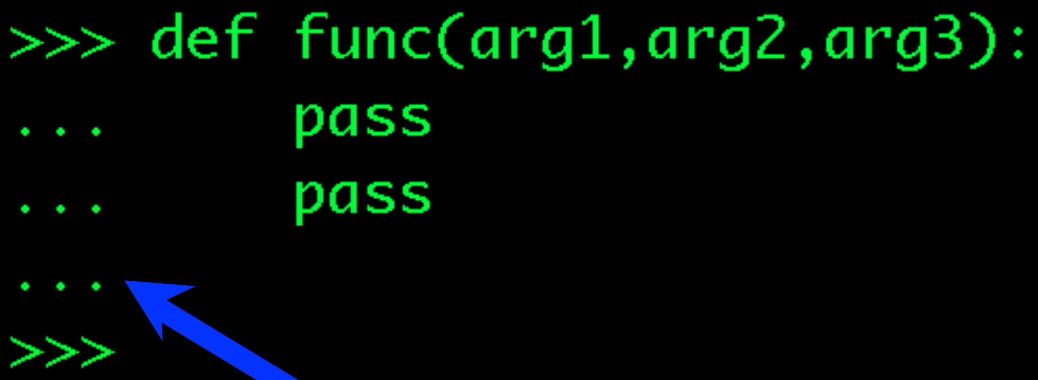
pass: Python’s “placeholder” or NOP

NOP: short for “NO OPeration”

Functions *must* have a body!

Snap! ⟷ Python

Functions : Defining



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

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

Snap! ↔ Python

Sidebar: Keywords

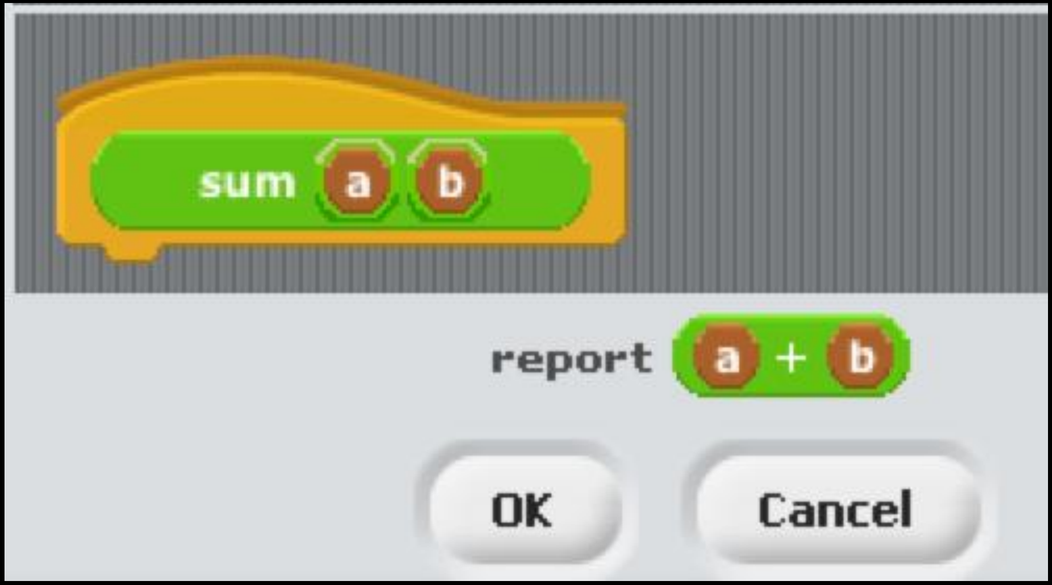
```
and        del        from       not        while
as         elif       global     or         with
assert     else       if         pass       yield
break      except     import     print
class      exec       in         raise
continue   finally    is         return
def        for        lambda 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

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

Snap! ⟷ Python

Functions : Returning Values

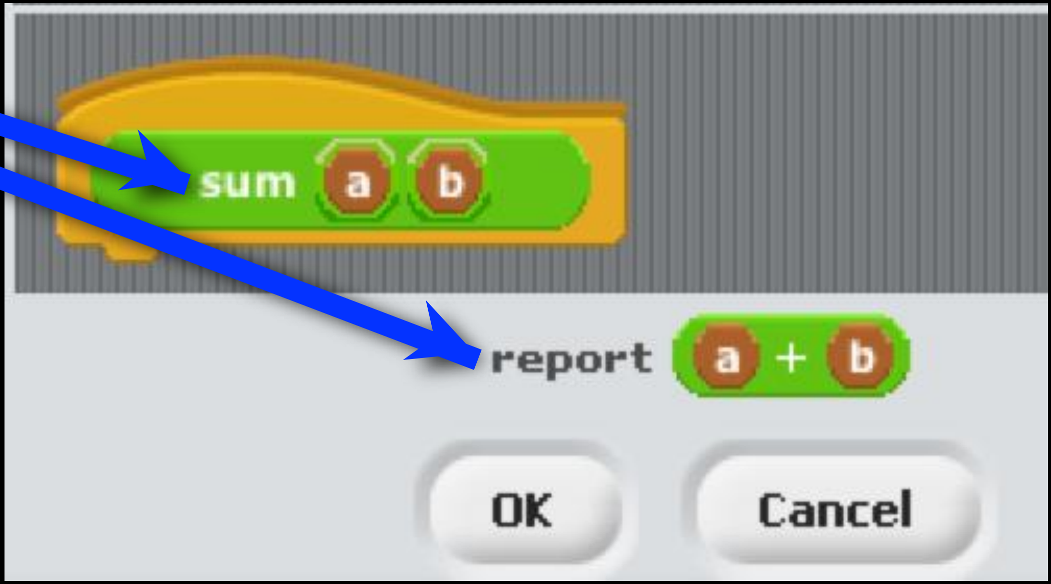
```
>>> def sum(a,b):
...     return (a+b)
...
>>> c=sum(5,7)
>>> print c
12
```



Snap! ↔ Python

Functions : Returning Values

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



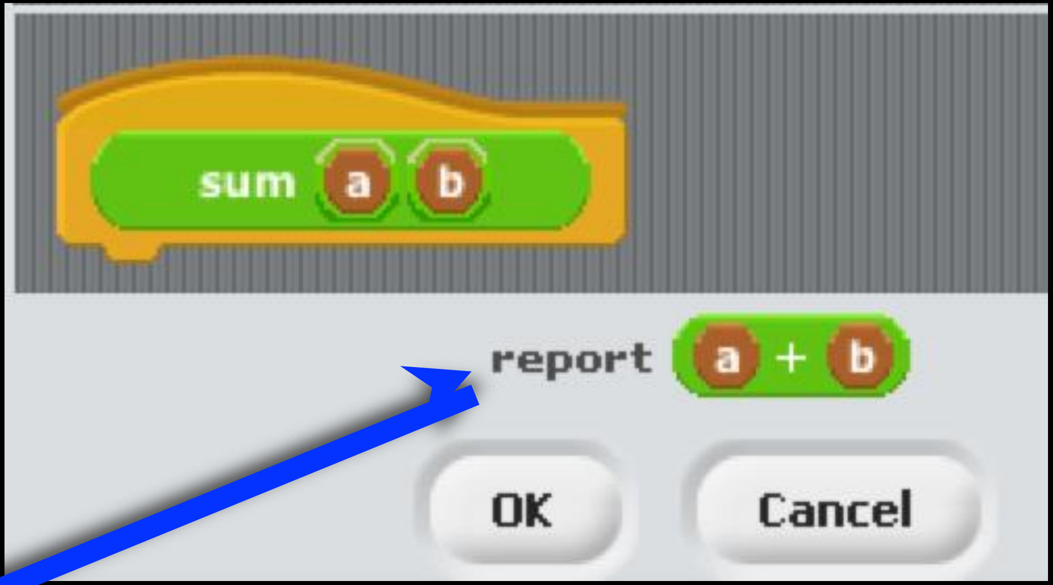
sum a b

report a + b

Snap! ↔ Python

Functions : Returning Values

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

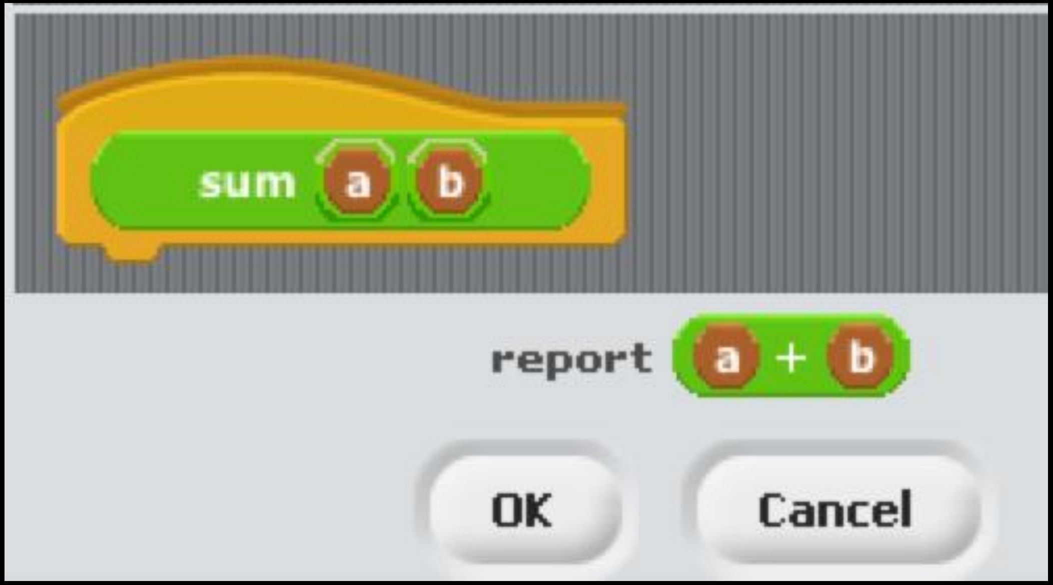


“return” and “report” are equivalent!

Snap! ⟷ Python

Functions : Returning Values

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



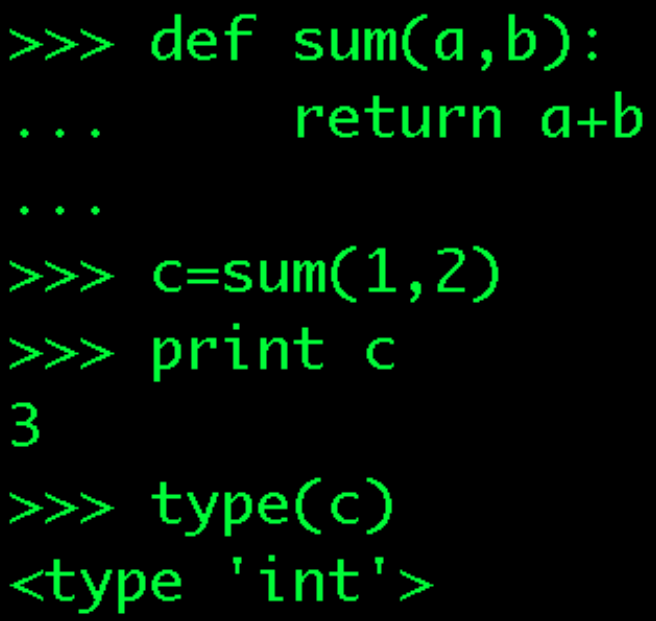
sum (a) (b)

report (a) + (b)

What is the type of the variable ‘c’?

Snap! ⟷ Python

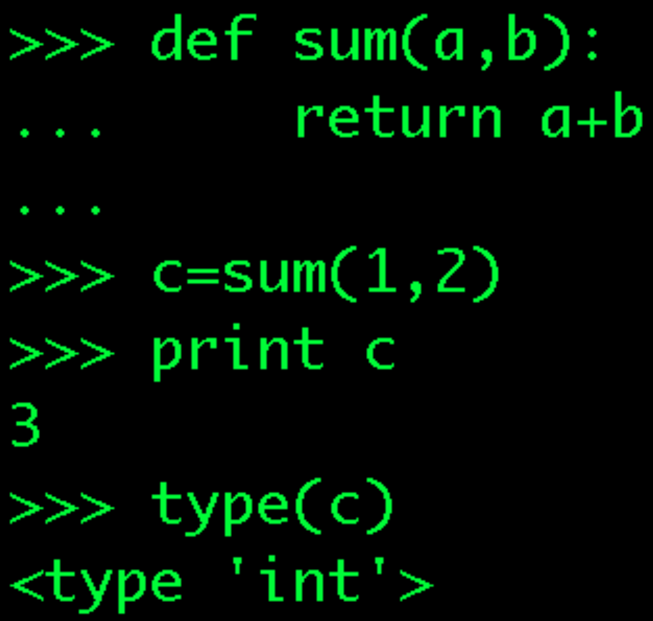
Functions : Type? It depends!



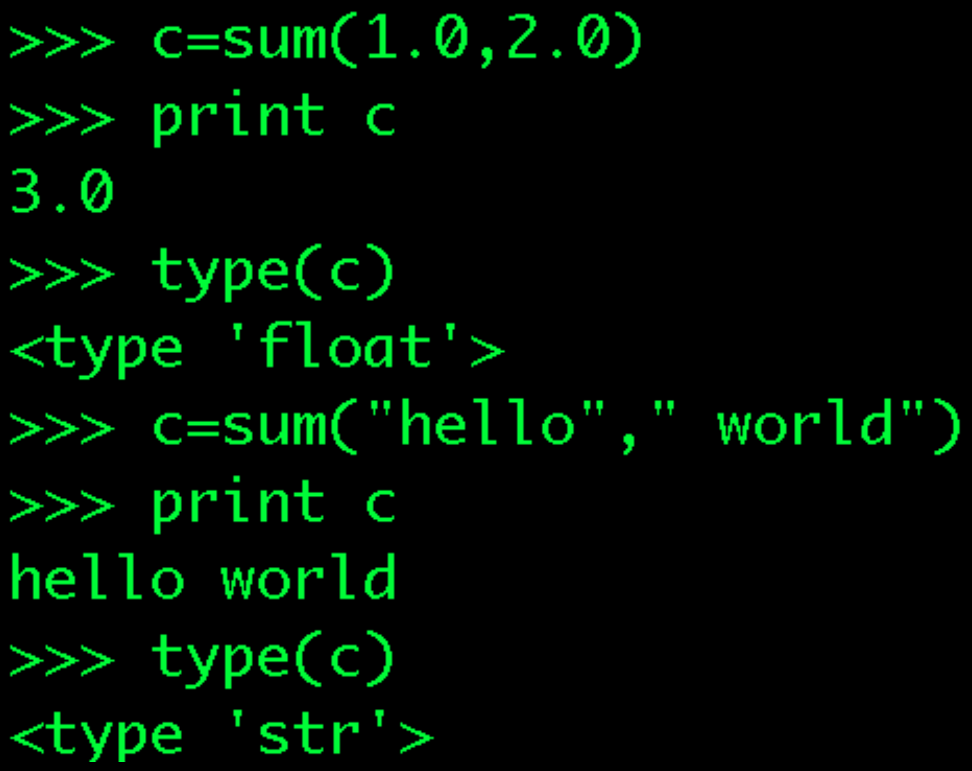
```
>>> def sum(a,b):
...     return a+b
...
>>> c=sum(1,2)
>>> print c
3
>>> type(c)
<type 'int'>
```

Snap! ↔ Python

Functions : Type? It depends!



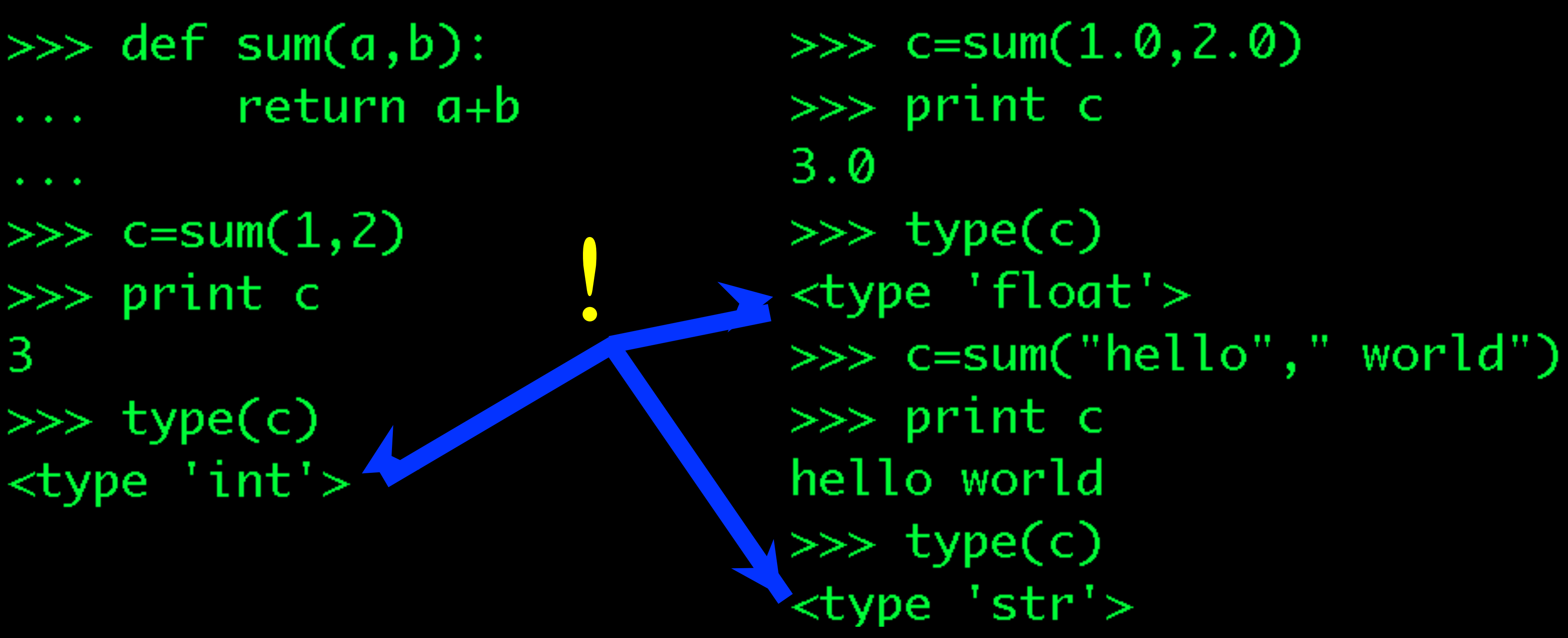
```
>>> def sum(a,b):
...     return a+b
...
>>> c=sum(1,2)
>>> print c
3
>>> type(c)
<type 'int'>
```



```
>>> 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'>
```

Snap! ↔ Python

Functions : C’s type? It depends!

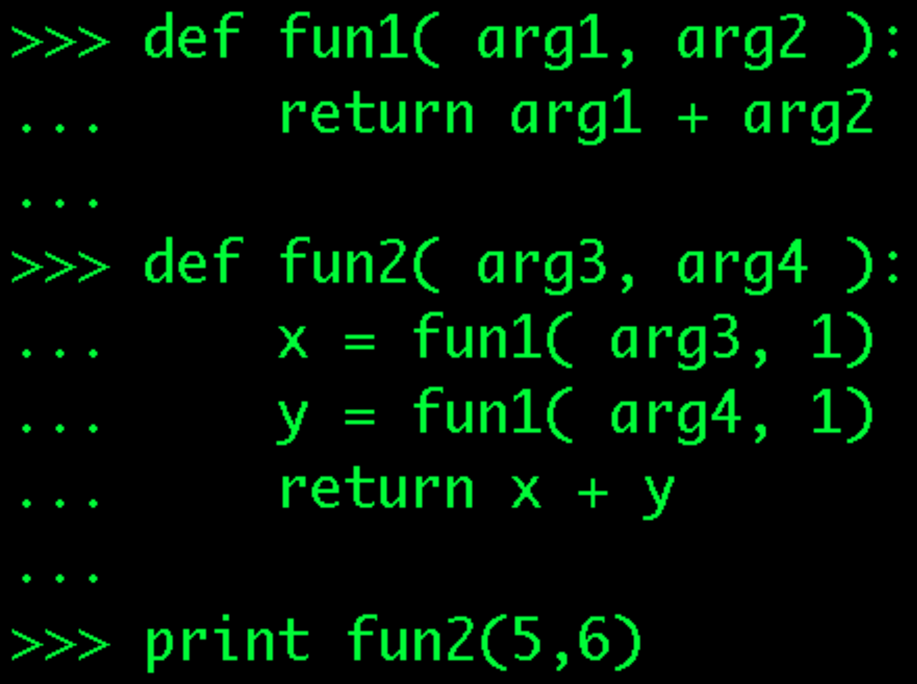


```
>>> def sum(a,b):
...     return a+b
...
>>> c=sum(1,2)
>>> print c
3
>>> type(c)
<type 'int'>
```

```
>>> 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'>
```

Snap! ↔ Python

Functions : Practice

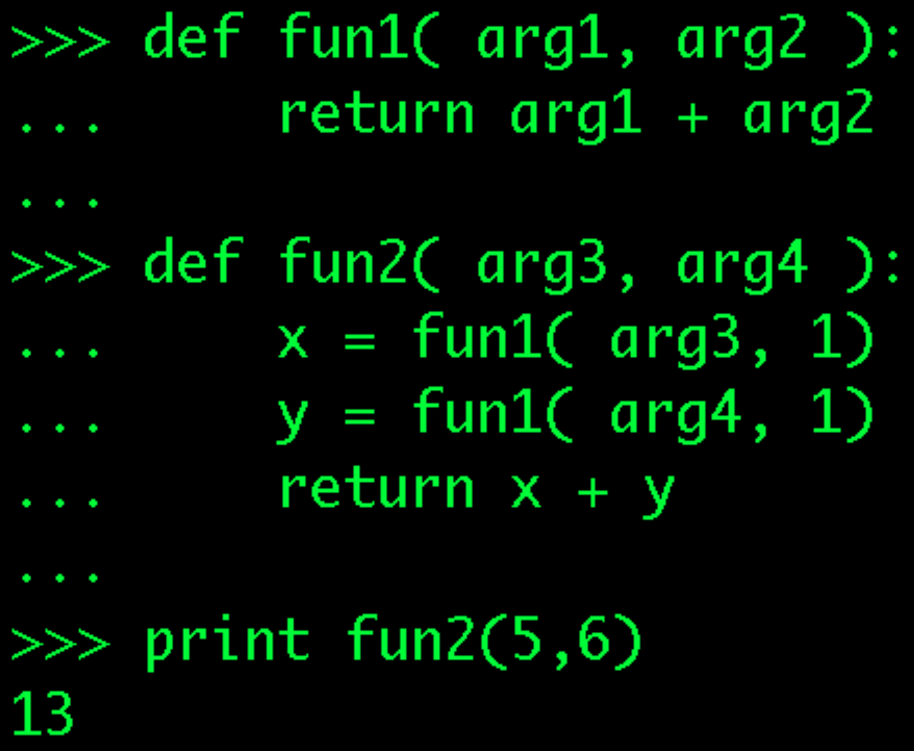


```
>>> def fun1( arg1, arg2 ):
...     return arg1 + arg2
...
>>> def fun2( arg3, arg4 ):
...     x = fun1( arg3, 1)
...     y = fun1( arg4, 1)
...     return x + y
...
>>> print fun2(5,6)
```

What will this print?

Snap! ↔ Python

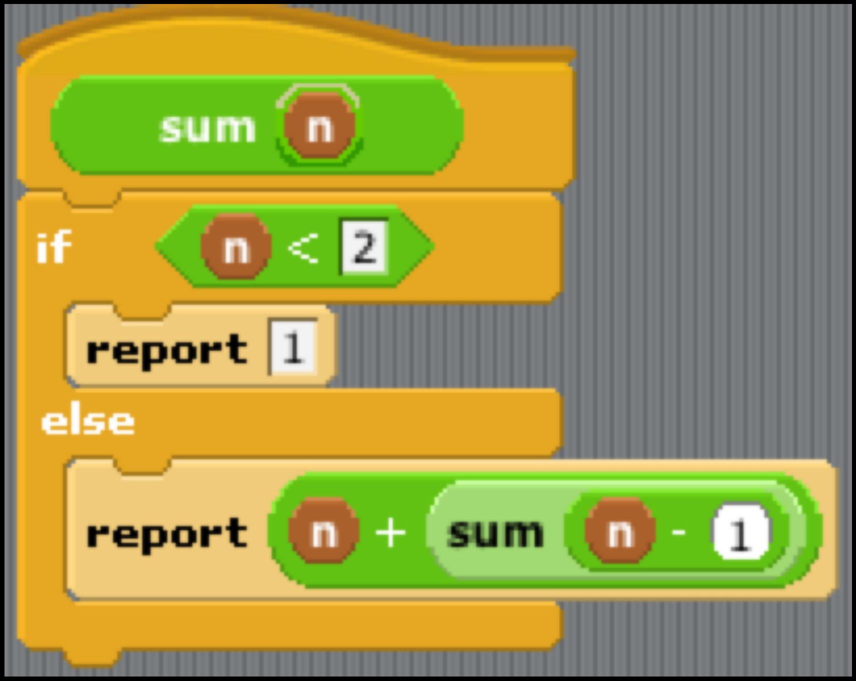
Functions : Practice



```
>>> 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
```

Snap! ↔ Python

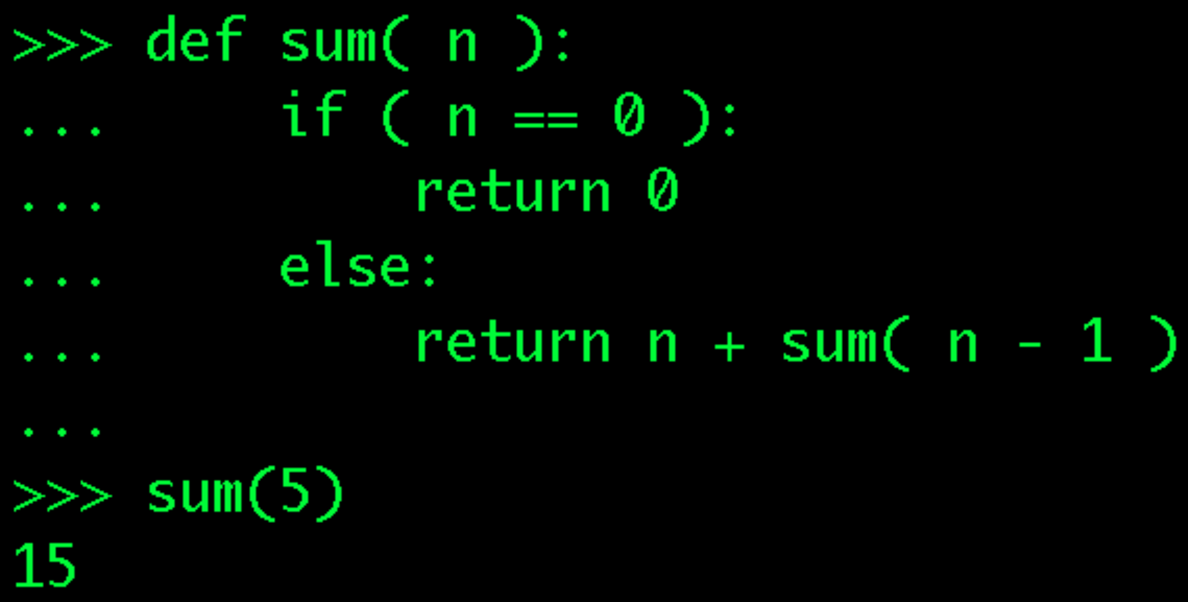
Functions : Recursion!



```
sum (n)
if <(n) < 2>
    report 1
else
    report (n) + (sum ((n) - 1))
```

Snap! ↔ Python

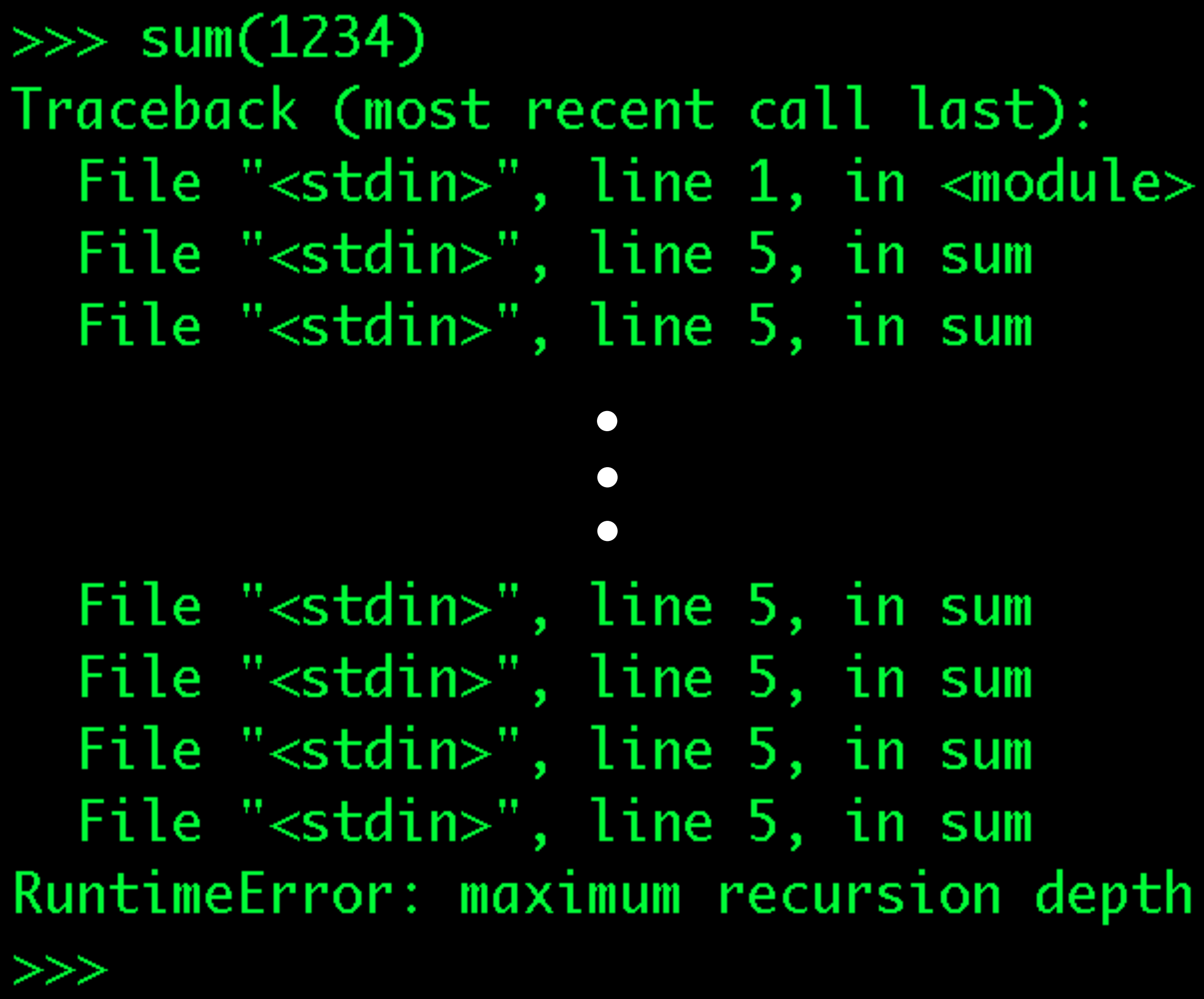
Functions : Recursion!



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

Snap! ↔ Python

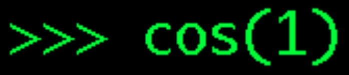
Functions : Recursion! Within Reason!



```
>>> 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
>>>
```

Snap! ⟷ Python

Importing

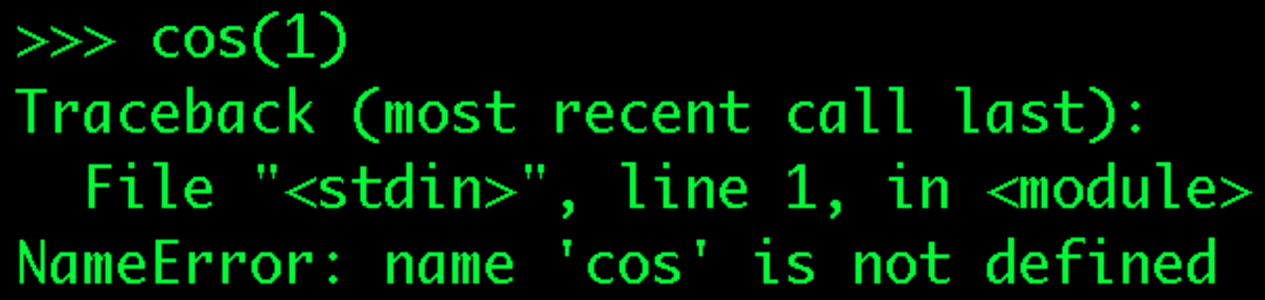


```
>>> cos(1)
```

cosine(radians)

Snap! ⟷ Python

Importing



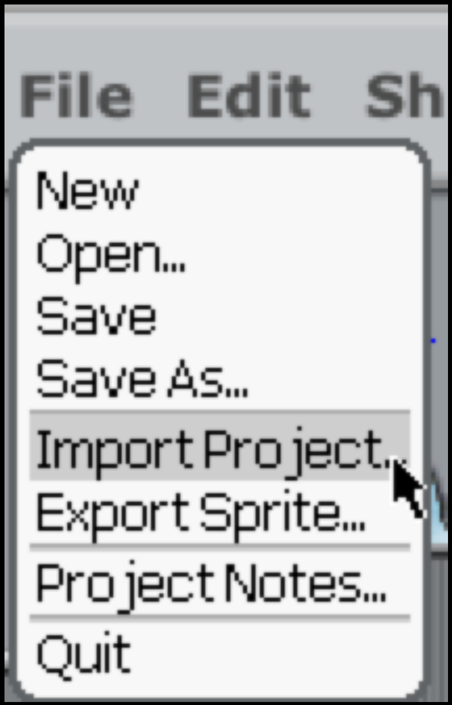
```
>>> cos(1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'cos' is not defined
```

ERROR!

Hmmmm....

Snap! ↔ Python

Importing

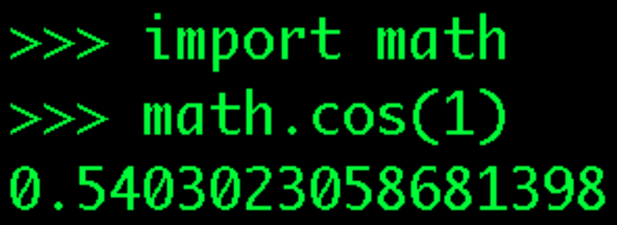


```
>>> import math
```

“math” module

Beyond Blocks: Python #1

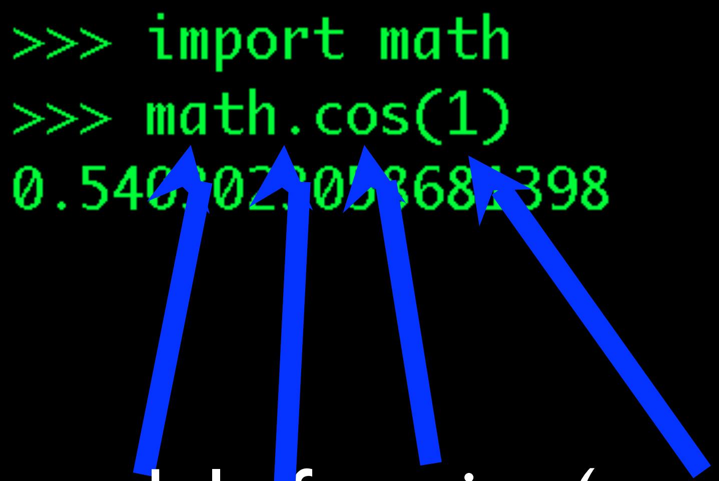
Importing



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

Beyond Blocks: Python #1

Importing

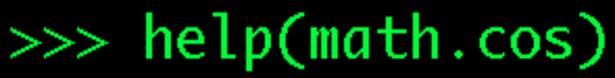


```
>>> import math
>>> math.cos(1)
0.5403023058681398
```

module.function(args)

Beyond Blocks: Python #1

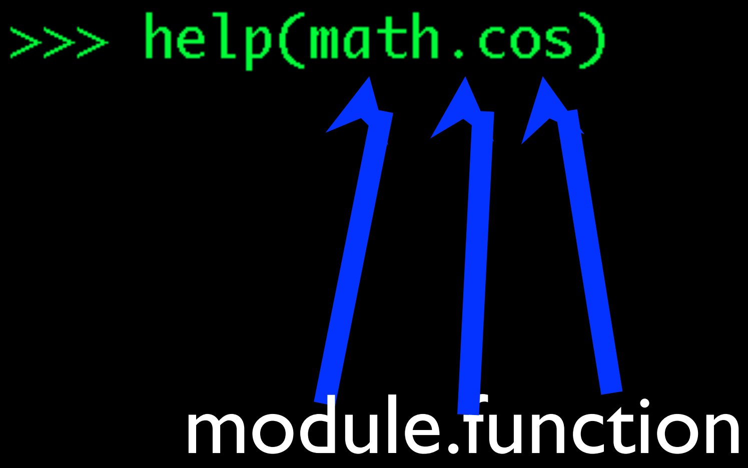
Importing, help!



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

Beyond Blocks: Python #1

Importing, help!

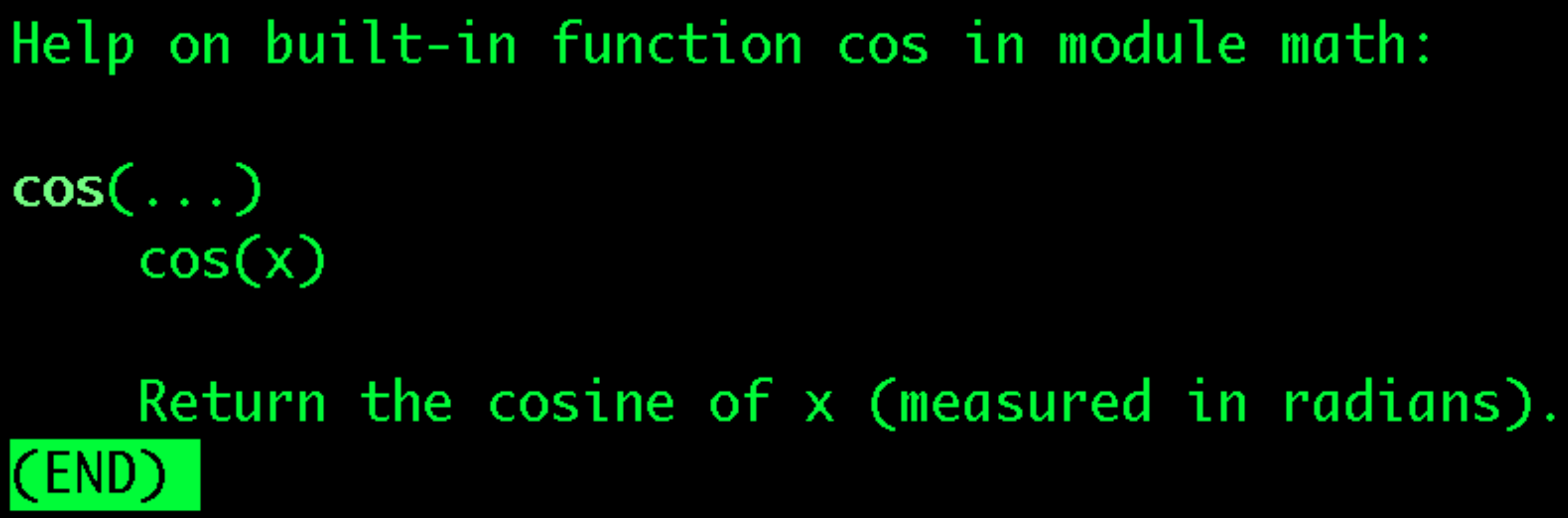


```
>>> help(math.cos)
```

module.function

Beyond Blocks: Python #1

Importing, help!

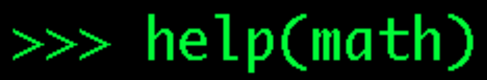


```
Help on built-in function cos in module math:

cos(...)
    cos(x)

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

import antigravity

Beyond Blocks: Python #1

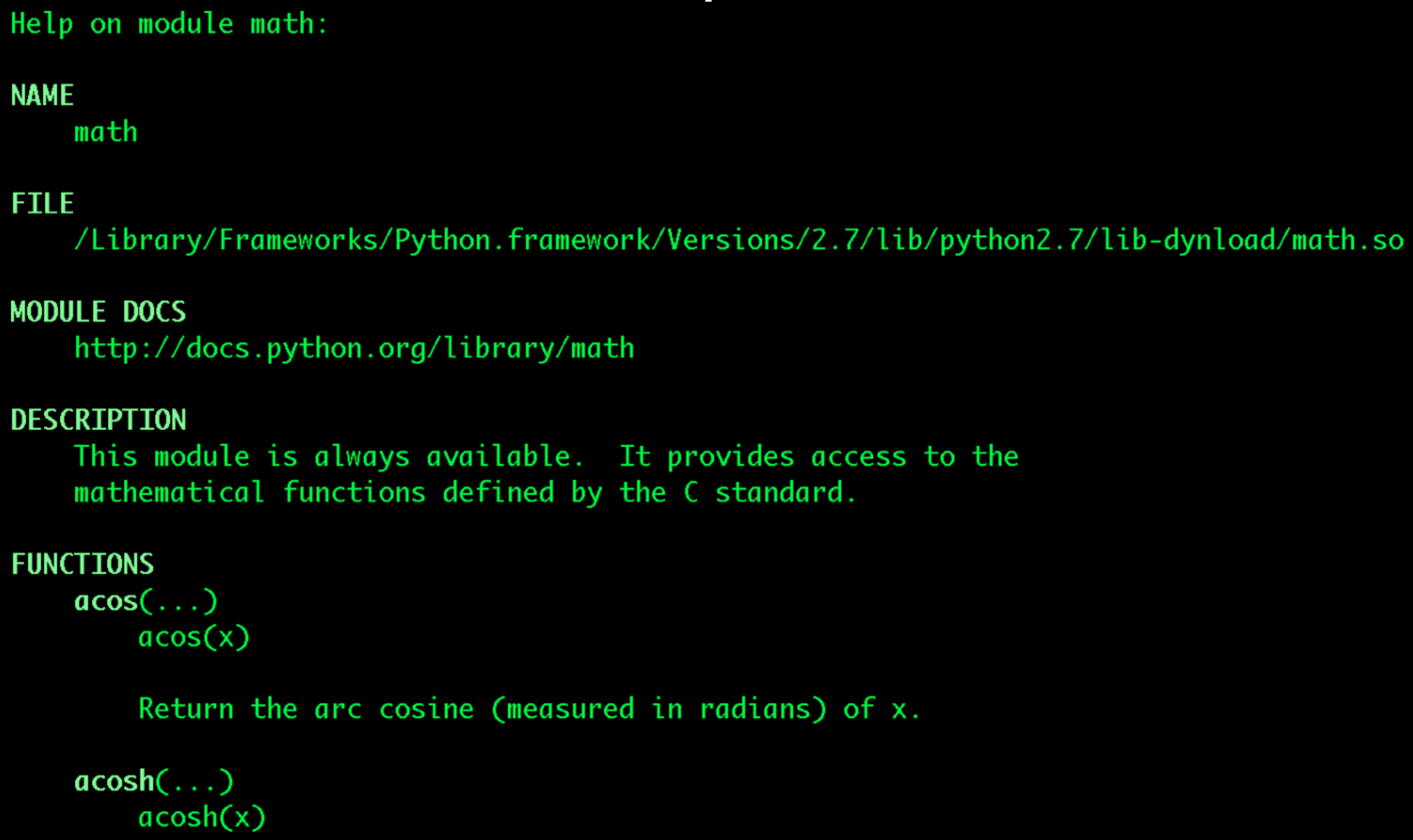
Help!



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

Beyond Blocks: Python #1

Help!

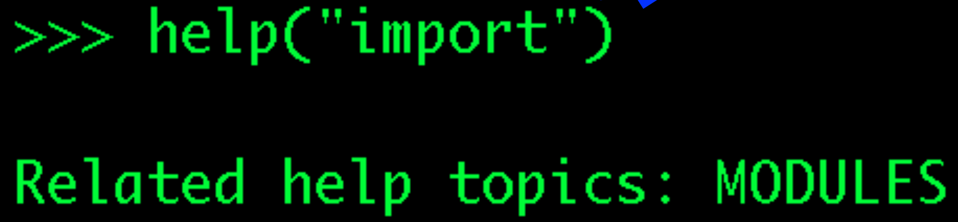


```
Help on module math:

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!

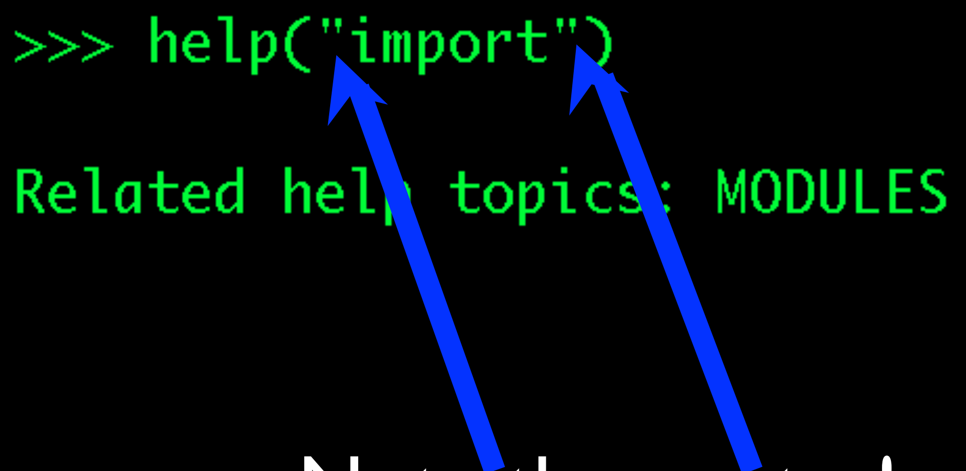
Python keyword



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

Related help topics: MODULES
```

Beyond Blocks: Python #1

Help!

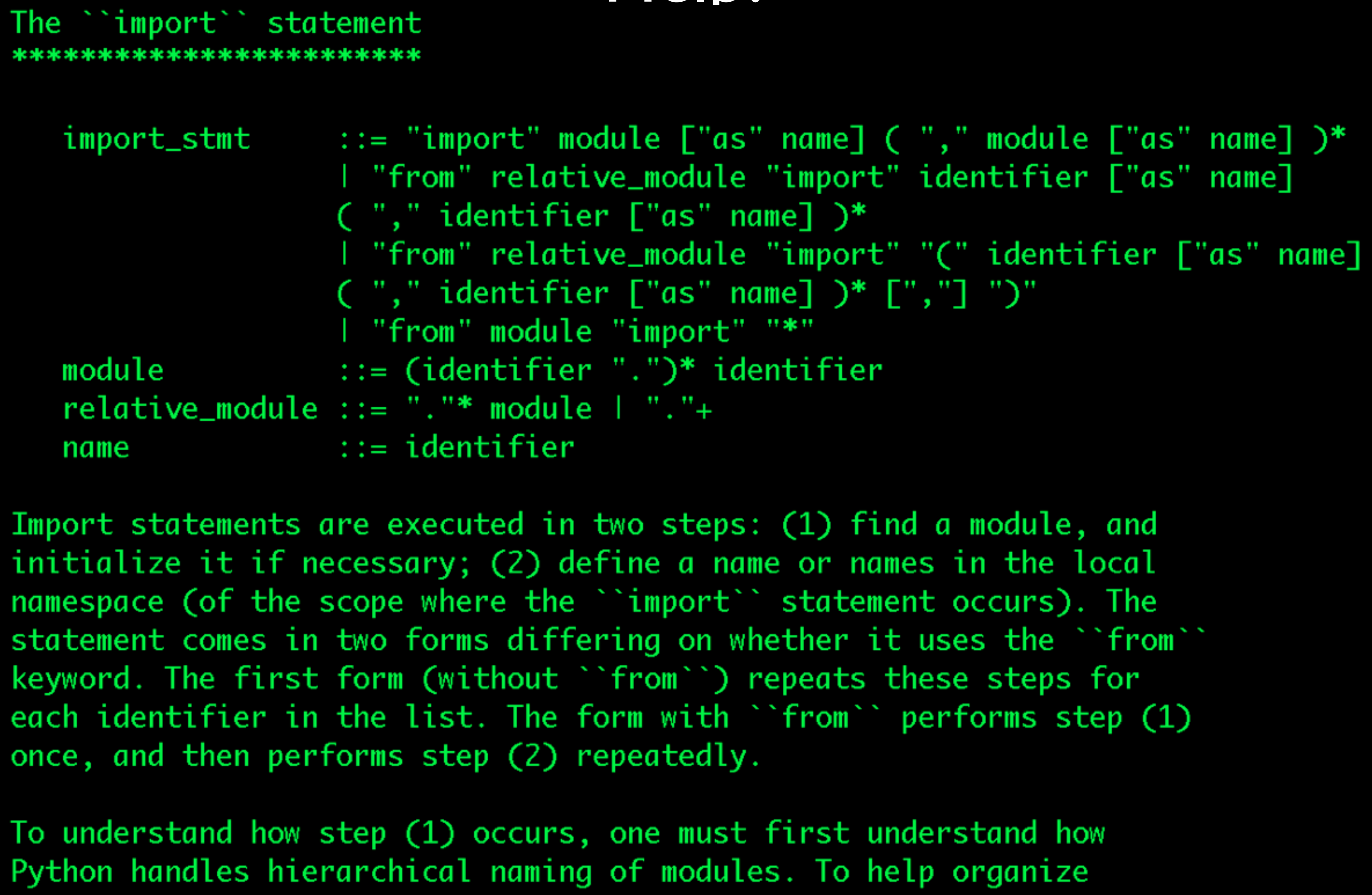


```
>>> help("import")

Related help topics: MODULES
```

Note the quotes!

Beyond Blocks: Python #1

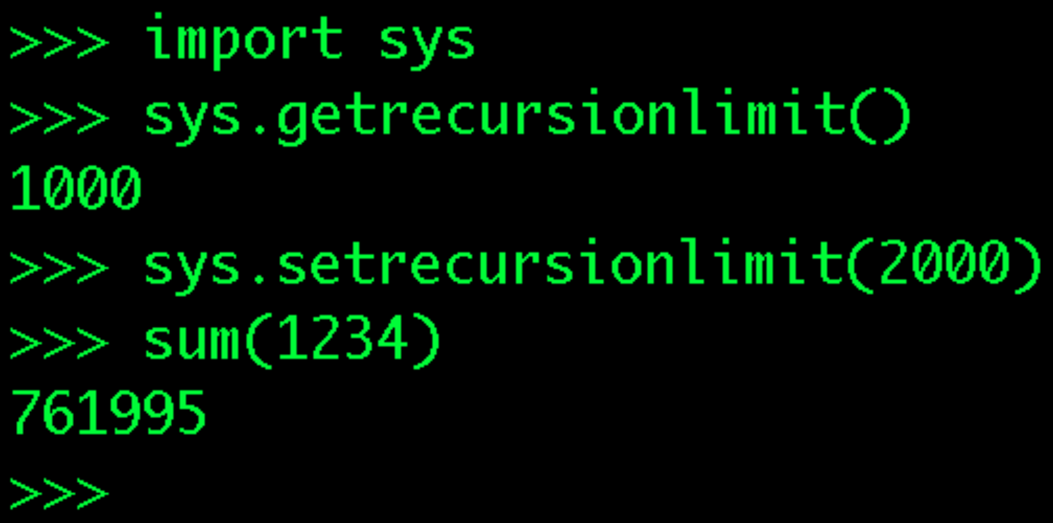
Help!



```
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
>>>
```

Turtle Module

```
from turtle import *
color('red', 'yellow')
begin_fill()
while True:
    forward(200)
    left(170)
    if abs(pos()) < 1:
        break
end_fill()
done()
```

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/