Practical Computing for Scientists

Armin Sobhani

CSCI 2000U

UOIT – Fall 2015

UNIVERSITY OF ONTARIO
INSTITUTE OF TECHNOLOGY

Python

First-Class Functions

by Greg Wilson

An integer is 32 bits of data…

An integer is 32 bits of data…

…that variables can refer to

An integer is 32 bits of data…

…that variables can refer to

A string is a sequence of bytes representing characters…

An integer is 32 bits of data…

…that variables can refer to

A string is a sequence of bytes representing characters…

…that variables can refer to

An integer is 32 bits of data…

…that variables can refer to

A string is a sequence of bytes representing characters…

…that variables can refer to

A function is a sequence of bytes representing instructions…

An integer is 32 bits of data…

…that variables can refer to

A string is a sequence of bytes representing characters…

…that variables can refer to

A function is a sequence of bytes representing instructions…

…and yes, variables can refer to them to

An integer is 32 bits of data…

…that variables can refer to

A string is a sequence of bytes representing characters…

…that variables can refer to

A function is a sequence of bytes representing instructions…

…and yes, variables can refer to them to

This turns out to be very useful, and very powerful

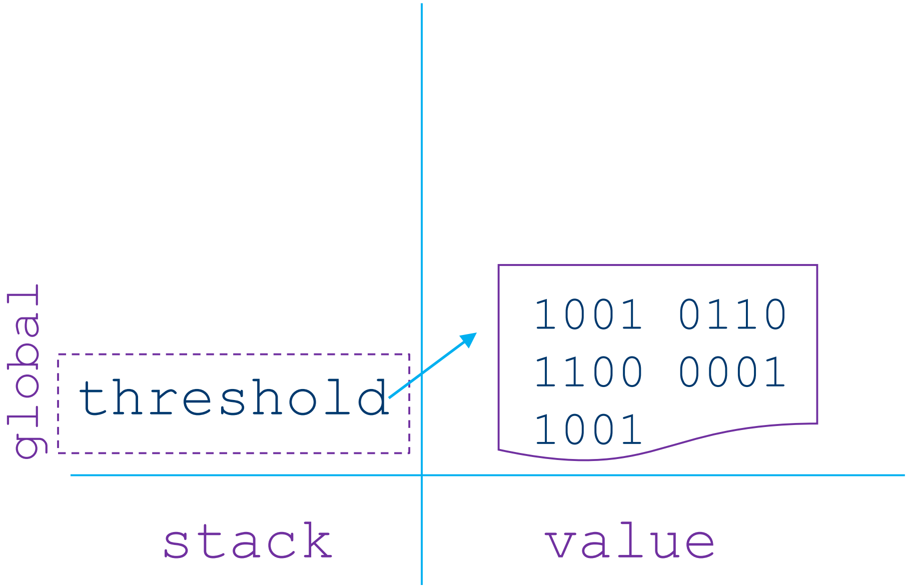
What happens when a function is defined

What happens when a function is defined

```
def threshold(signal):
  return 1.0 / sum(signal)
```

What happens when a function is defined

```
def threshold(signal):
  return 1.0 / sum(signal)
```

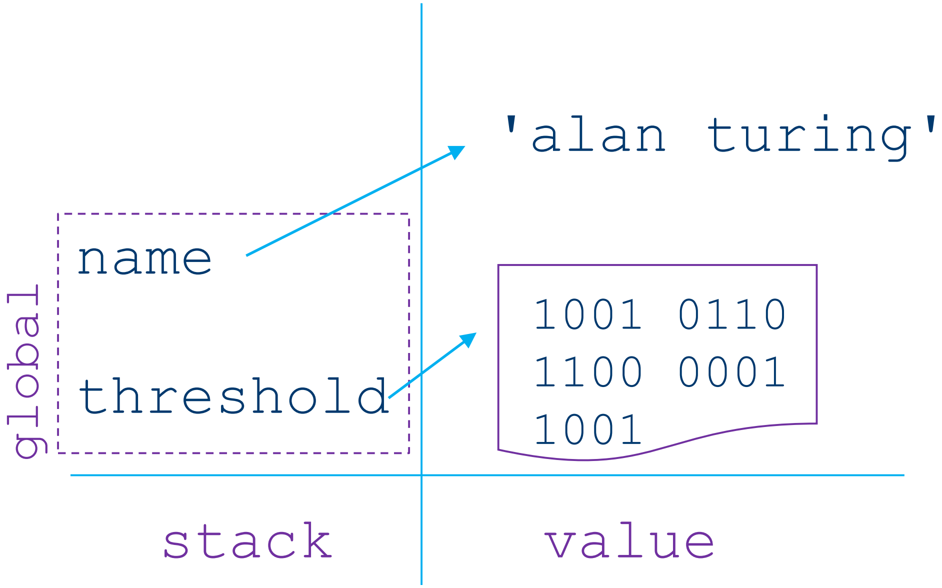


What happens when a function is defined

```
def threshold(signal):
    return 1.0 / sum(signal)
```

Not really very different from:

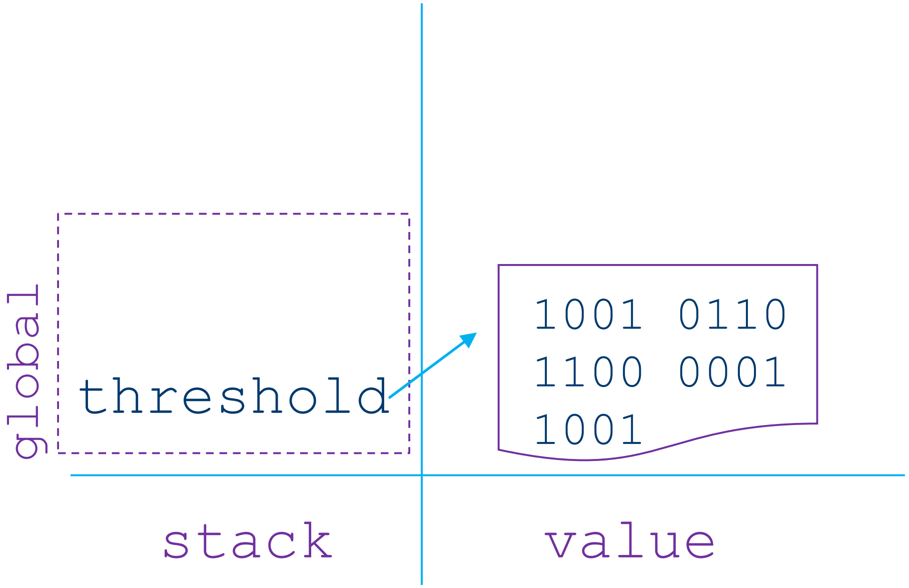
```
name = 'Alan Turing'
```



Can assign that value to another variable

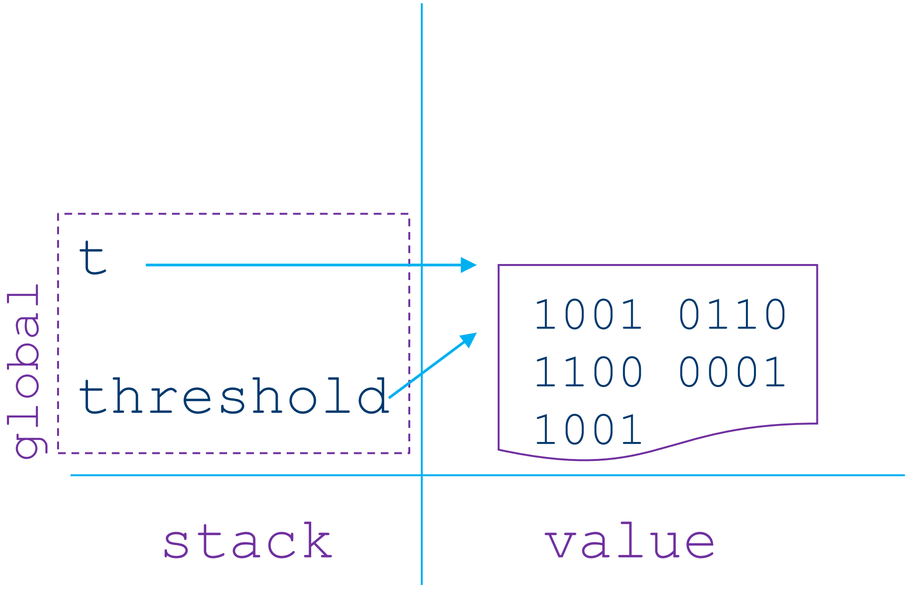
Can assign that value to another variable

```
def threshold(signal):
  return 1.0 / sum(signal)
```



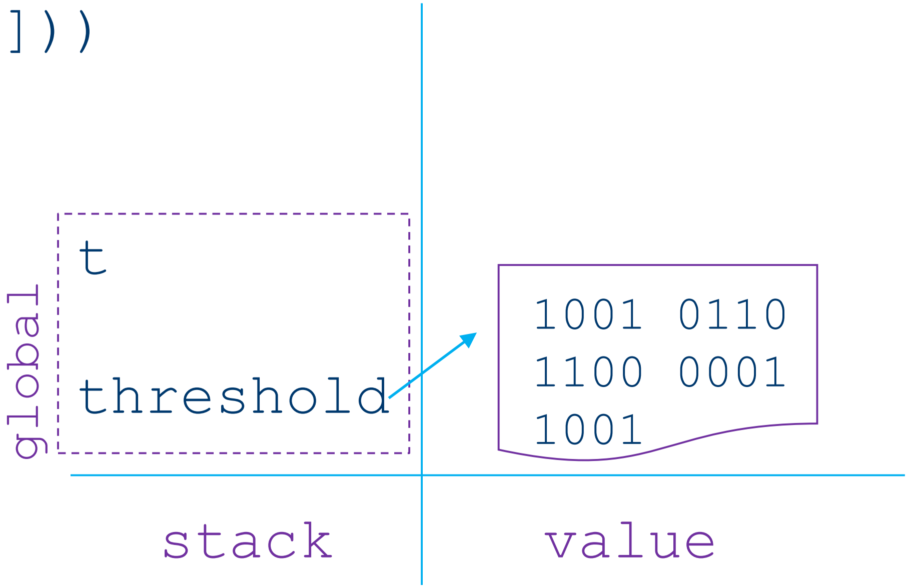
Can assign that value to another variable

```
def threshold(signal):
  return 1.0 / sum(signal)

t = threshold
```



Can assign that value to another variable

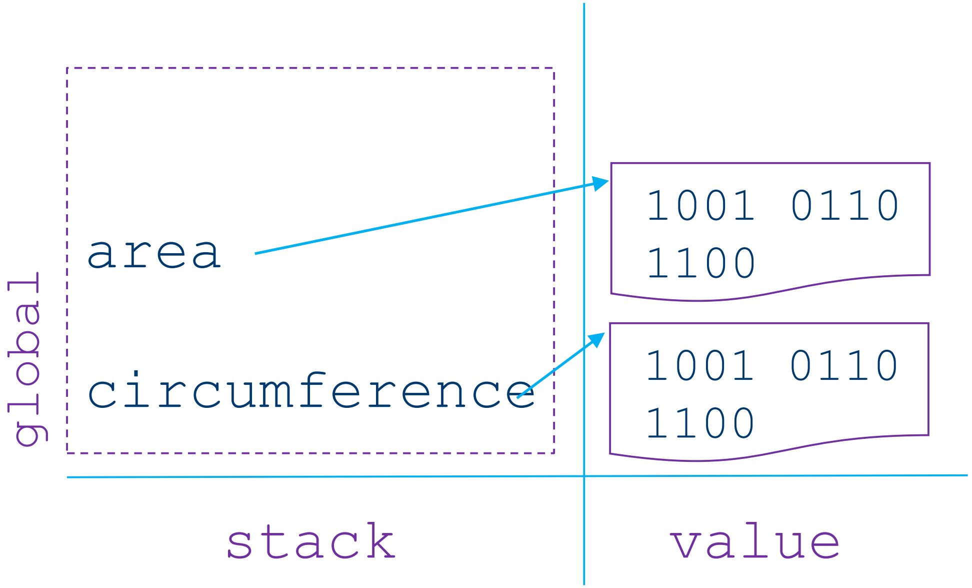
```
def threshold(signal):
  return 1.0 / sum(signal)

t = threshold
print(t([0.1, 0.4, 0.2]))
1.42857
```



Can put (a reference to) the function in a list

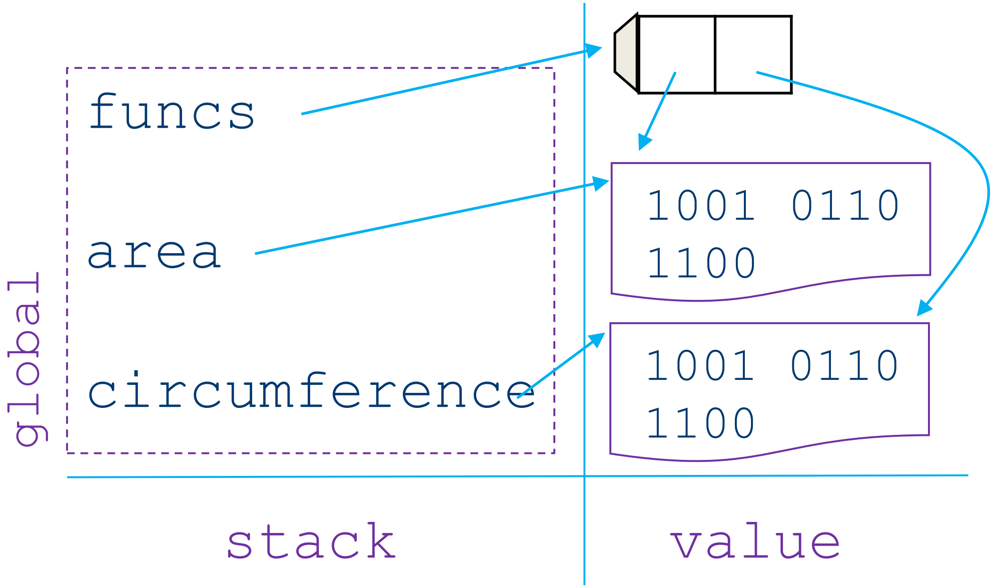
Can put (a reference to) the function in a list

```
def area(r):
  return PI * r * r

def circumference(r):
  return 2 * PI * r
```



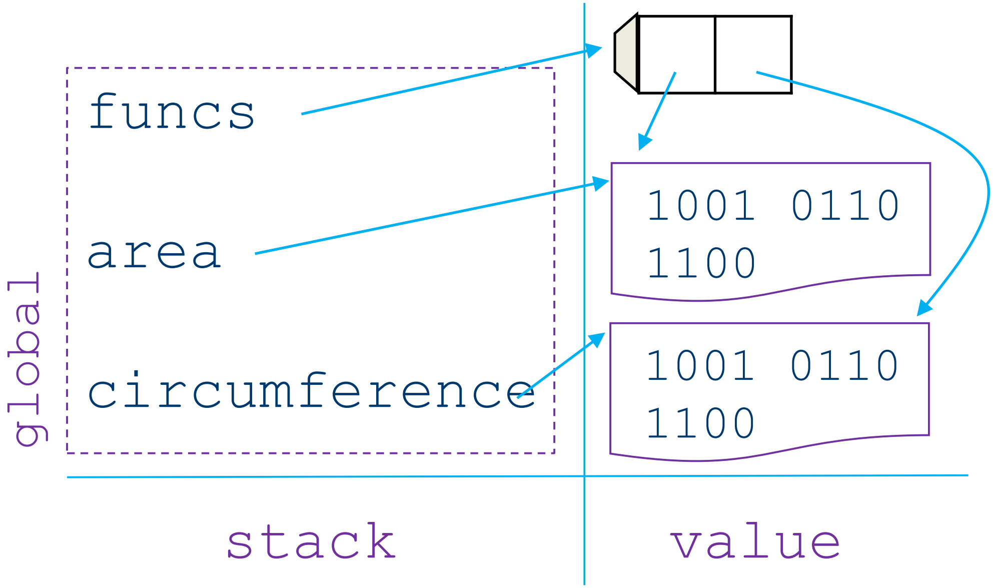
Can put (a reference to) the function in a list

```
def area(r):
  return PI * r * r

def circumference(r):
  return 2 * PI * r

funcs = [area, circumference]
```



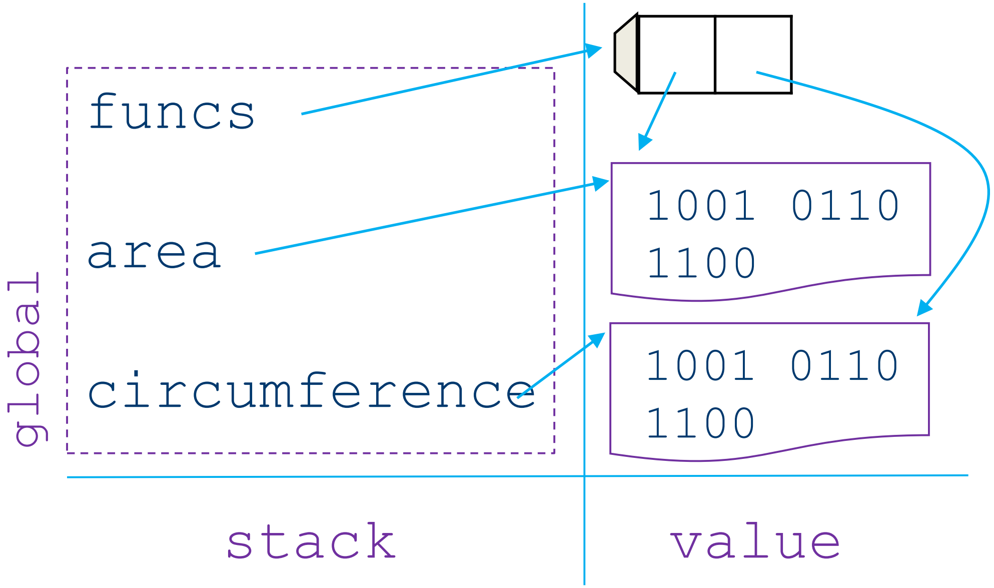
Can put (a reference to) the function in a list

```
def area(r):
  return PI * r * r

def circumference(r):
  return 2 * PI * r

funcs = [area, circumference]

for f in funcs:
  print(f(1.0))
```



Can put (a reference to) the function in a list

```
def area(r):
  return PI * r * r

def circumference(r):
  return 2 * PI * r

funcs = [area, circumference]

for f in funcs:
  print(f(1.0))
3.14159
6.28318
```



Can pass (a reference to) the function into a function

Can pass (a reference to) the function into a function

```
def call_it(func, value):
  return func(value)
```

Can pass (a reference to) the function into a function

```
def call_it(func, value):
  return func(value)

print(call_it(area, 1.0))
3.14159
```

Can pass (a reference to) the function into a function

```
def call_it(func, value):
  return func(value)

print(call_it(area, 1.0))
3.14159

print(call_it(circumference, 1.0))
6.28318
```

Can now write functions of functions

Can now write functions of functions

```
def do_all(func, values):
  result = []
  for v in values:
    temp = func(v)
    result.append(temp)
  return result
```

Can now write functions of functions

```
def do_all(func, values):
  result = []
  for v in values:
    temp = func(v)
    result.append(temp)
  return result
```

```
print(do_all(area, [1.0, 2.0, 3.0]))
```

Can now write functions of functions

```
def do_all(func, values):
  result = []
  for v in values:
    temp = func(v)
    result.append(temp)
  return result
              print(do_all(area, [1.0, 2.0, 3.0]))
              [3.14159, 12.56636, 28.27431]
```

Can now write functions of functions

```
def do_all(func, values):
  result = []
  for v in values:
    temp = func(v)
    result.append(temp)
  return result
```

```
print(do_all(area, [1.0, 2.0, 3.0]))
[3.14159, 12.56636, 28.27431]
```

```
def slim(text):
  return text[1:-1]
```

Can now write functions of functions

```
def do_all(func, values):
  result = []
  for v in values:
    temp = func(v)
    result.append(temp)
  return result
```

```
print(do_all(area, [1.0, 2.0, 3.0]))
[3.14159, 12.56636, 28.27431]

def slim(text):
  return text[1:-1]

print(do_all(slim, ['abc', 'defgh']))
b efg
```

Higher-order functions allow re-use of control flow

Higher-order functions allow re-use of control flow

```
def combine_values(func, values):
  current = values[0]
  for v in range(1, len(values)):
    current = func(current, v)
  return current
```

Higher-order functions allow re-use of control flow

```
def combine_values(func, values):
  current = values[0]
  for v in range(1, len(values)):
    current = func(current, v)
  return current

def add(x, y): return x + y
def mul(x, y): return x * y
```

Higher-order functions allow re-use of control flow

```
def combine_values(func, values):
  current = values[0]
  for v in range(1, len(values)):
    current = func(current, v)
  return current

def add(x, y): return x + y
def mul(x, y): return x * y

print(combine_values(add, [1, 3, 5]))
9
```

Higher-order functions allow re-use of control flow

```
def combine_values(func, values):
  current = values[0]
  for v in range(1, len(values)):
    current = func(current, v)
  return current

def add(x, y): return x + y
def mul(x, y): return x * y

print(combine_values(add, [1, 3, 5]))
9
print(combine_values(mul, [1, 3, 5]))
15
```

Without higher order functions

Without higher order functions

	op_1	op_2	op_3
data_structure_A	do_1A	do_2A	do_3A
data_structure_B	do_1B	do_2B	do_3B
data_structure_C	do_1C	do_2C	do_3C

Without higher order functions

	op_1	op_2	op_3
data_structure_A	do_1A	do_2A	do_3A
data_structure_B	do_1B	do_2B	do_3B
data_structure_C	do_1C	do_2C	do_3C

total: 9

Without higher order functions

	op_1	op_2	op_3
data_structure_A	do_1A	do_2A	do_3A
data_structure_B	do_1B	do_2B	do_3B
data_structure_C	do_1C	do_2C	do_3C

total: 9

With higher order functions

	op_1	op_2	op_3
operate_on_A			
operate_on_B			
operate_on_C			

Without higher order functions

	op_1	op_2	op_3
data_structure_A	do_1A	do_2A	do_3A
data_structure_B	do_1B	do_2B	do_3B
data_structure_C	do_1C	do_2C	do_3C

total: 9

With higher order functions

	op_1	op_2	op_3
operate_on_A			
operate_on_B			
operate_on_C			

total: 6

Must need to know *something* about the function in order to call it

Must need to know *something* about the function

in order to call it

Like number of arguments

Must need to know *something* about the function

in order to call it

Like ~~number of arguments~~

Must need to know *something* about the function

in order to call it

Like ~~number of arguments~~

```
def add_all(*args):
  total = 0
  for a in args:
    total += a
  return total
```

Must need to know *something* about the function

in order to call it

Like ~~number of arguments~~

```
def add_all(*args):
  total = 0
  for a in args:
    total += a
  return total
```

Must need to know *something* about the function

in order to call it

Like ~~number of arguments~~

```
def add_all(*args):
  total = 0
  for a in args:
    total += a
  return total

print(add_all())
0
```

Must need to know *something* about the function

in order to call it

Like ~~number of arguments~~

```
def add_all(*args):
  total = 0
  for a in args:
    total += a
  return total

print(add_all())
0
print(add_all(1, 2, 3))
6
```

Combine with "regular" parameters

Combine with "regular" parameters

```
def combine_values(func, *values):
  current = values[0]
  for i in range(1, len(values)):
    current = func(current, v)
  return current
```

Combine with "regular" parameters

```
def combine_values(func, *values):
  current = values[0]
  for i in range(1, len(values)):
    current = func(current, v)
  return current
```

Combine with "regular" parameters

```
def combine_values(func, *values):
  current = values[0]
  for i in range(1, len(values)):
    current = func(current, v)
  return current

print(combine_values(add, 1, 3, 5))
9
```

Combine with "regular" parameters

```
def combine_values(func, *values):
  current = values[0]
  for i in range(1, len(values)):
    current = func(current, v)
  return current

print(combine_values(add, 1, 3, 5))
9
```

What does `combine_values(add)` do?

Combine with "regular" parameters

```
def combine_values(func, *values):
  current = values[0]
  for i in range(1, len(values)):
    current = func(current, v)
  return current

print(combine_values(add, 1, 3, 5))
9
```

What does `combine_values(add)` do?

What should it do?

`filter(F, S)`	select elements of `S` for which `F` is `True`

`filter(F, S)`	select elements of `S` for which `F` is `True`
`map(F, S)`	apply `F` to each element of `S`

`filter(F, S)`	select elements of `S` for which `F` is `True`
`map(F, S)`	apply `F` to each element of `S`
`reduce(F, S)`	use `F` to combine all elements of `S`

`filter(F, S)`	select elements of `S` for which `F` is `True`
`map(F, S)`	apply `F` to each element of `S`
`reduce(F, S)`	use `F` to combine all elements of `S`

```
def positive(x): return x >= 0
print(filter(positive, [-3, -2, 0, 1, 2]))
[0, 1, 2]
```

`filter(F, S)`	select elements of `S` for which `F` is `True`
`map(F, S)`	apply `F` to each element of `S`
`reduce(F, S)`	use `F` to combine all elements of `S`

```
def positive(x): return x >= 0
print(filter(positive, [-3, -2, 0, 1, 2]))
[0, 1, 2]
```

```
def negate(x): return -x
print(map(negate, [-3, -2, 0, 1, 2]))
[3, 2, 0, -1, -2]
```

`filter(F, S)`	select elements of `S` for which `F` is `True`
`map(F, S)`	apply `F` to each element of `S`
`reduce(F, S)`	use `F` to combine all elements of `S`

```
def positive(x): return x >= 0
print(filter(positive, [-3, -2, 0, 1, 2]))
[0, 1, 2]

def negate(x): return -x
print(map(negate, [-3, -2, 0, 1, 2]))
[3, 2, 0, -1, -2]

def add(x, y): return x+y
print reduce(add, [-3, -2, 0, 1, 2])
-2
```

What is programming?

What is programming?

Novice: writing instructions for the computer

What is programming?

Novice: writing instructions for the computer

Expert: creating and combining abstractions

What is programming?

Novice: writing instructions for the computer

Expert: creating and combining abstractions

figure out what the pattern is

What is programming?

Novice: writing instructions for the computer

Expert: creating and combining abstractions

figure out what the pattern is

write it down as clearly as possible

What is programming?

Novice: writing instructions for the computer

Expert: creating and combining abstractions

figure out what the pattern is

write it down as clearly as possible

build more patterns on top of it

What is programming?

Novice: writing instructions for the computer

Expert: creating and combining abstractions

figure out what the pattern is

write it down as clearly as possible

build more patterns on top of it

But limits on short-term memory still apply

What is programming?

Novice: writing instructions for the computer

Expert: creating and combining abstractions

figure out what the pattern is

write it down as clearly as possible

build more patterns on top of it

But limits on short-term memory still apply

Hard to understand what meta-meta-functions actually do

Python

Libraries

by Greg Wilson

UNIVERSITY OF ONTARIO
INSTITUTE OF TECHNOLOGY

A function is a way to turn a bunch of related statements into a single "chunk"

A function is a way to turn a bunch of related statements into a single "chunk"

– Avoid duplication

A function is a way to turn a bunch of related statements into a single "chunk"

- Avoid duplication
- Make code easier to read

A function is a way to turn a bunch of related statements into a single "chunk"

- Avoid duplication
- Make code easier to read

A *library* does the same thing for related functions

A function is a way to turn a bunch of related statements into a single "chunk"

- Avoid duplication
- Make code easier to read

A *library* does the same thing for related functions

Hierarchical organization

A function is a way to turn a bunch of related statements into a single "chunk"

- Avoid duplication
- Make code easier to read

A *library* does the same thing for related functions

Hierarchical organization

family	library
genus	function
species	statement

Every Python file can be used as a library

Every Python file can be used as a library

Use import to load it

Every Python file can be used as a library

Use import to load it

```
# halman.py
def threshold(signal):
  return 1.0 / sum(signal)
```

Every Python file can be used as a library

Use import to load it

```
# halman.py
def threshold(signal):
  return 1.0 / sum(signal)
```

```
# program.py
import halman
readings = [0.1, 0.4, 0.2]
print('signal threshold is', halman.threshold(readings))
```

Every Python file can be used as a library

Use import to load it

```
# halman.py
def threshold(signal):
  return 1.0 / sum(signal)
```

```
# program.py
import halman
readings = [0.1, 0.4, 0.2]
print('signal threshold is', halman.threshold(readings))
```

```
$ python program.py
signal threshold is 1.42857
```

When a module is imported, Python:

When a module is imported, Python:

1. Executes the statements it contains

When a module is imported, Python:

1. Executes the statements it contains
2. Creates an object that stores references to the top-level items in that module

When a module is imported, Python:

1. Executes the statements it contains
2. Creates an object that stores references to the top-level items in that module

```
# noisy.py
print('is this module being loaded?')
NOISE_LEVEL = 1/3
```

When a module is imported, Python:

1. Executes the statements it contains
2. Creates an object that stores references to the top-level items in that module

```
# noisy.py
print('is this module being loaded?')
NOISE_LEVEL = 1/3
```

```
>>> import noisy
is this module being loaded?
```

When a module is imported, Python:

1. Executes the statements it contains
2. Creates an object that stores references to the top-level items in that module

```
# noisy.py
print('is this module being loaded?')
NOISE_LEVEL = 1/3
```

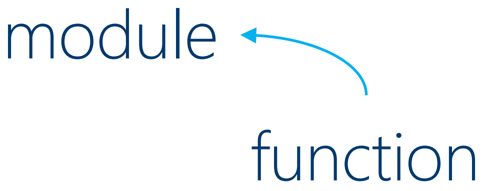
```
>>> import noisy
is this module being loaded?
>>> print(noisy.NOISE_LEVEL)
0.33333333
```

Each module is a *namespace*

Each module is a *namespace*

function

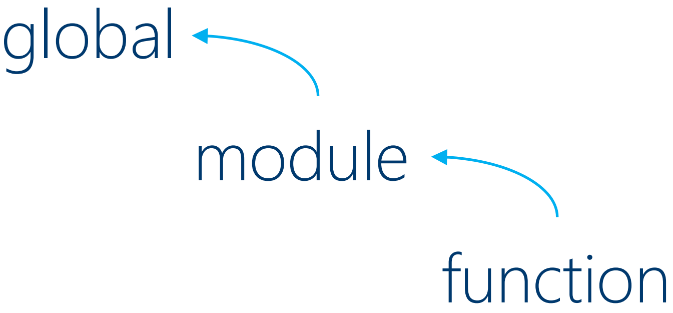
Each module is a *namespace*



module

function

Each module is a *namespace*

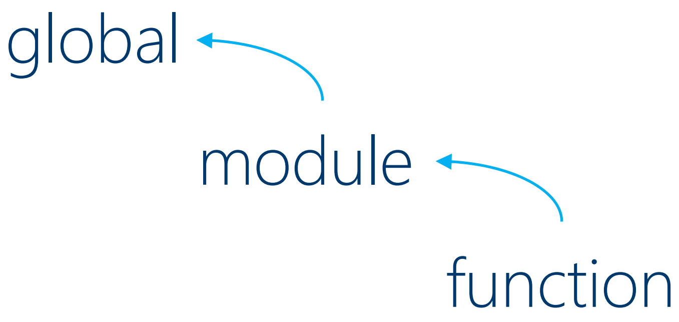


global

module

function

Each module is a *namespace*



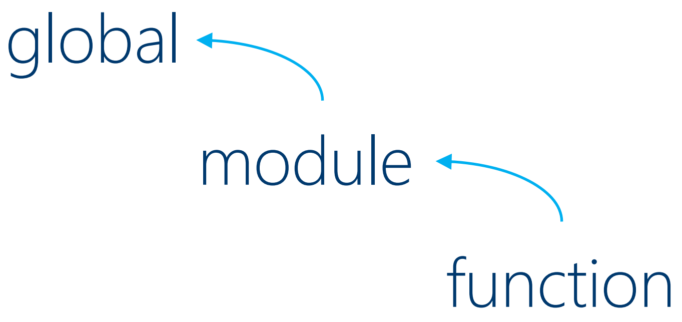
global

module

function

```
# module.py
NAME = 'Transylvania'

def func(arg):
  return NAME + ' ' + arg
```

Each module is a *namespace*


global

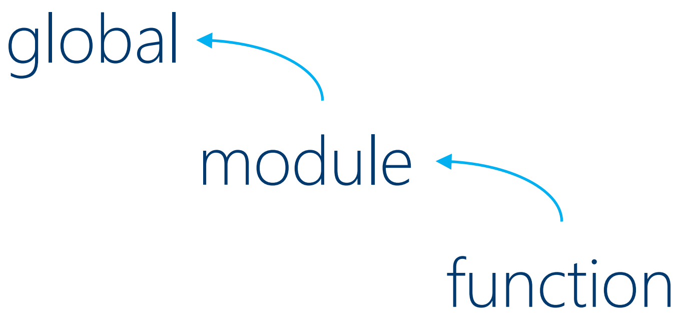
module

function

```
# module.py
NAME = 'Transylvania'

def func(arg):
  return NAME + ' ' + arg
```

```
>>> NAME = 'Hamunaptra'
```

Each module is a *namespace*



global

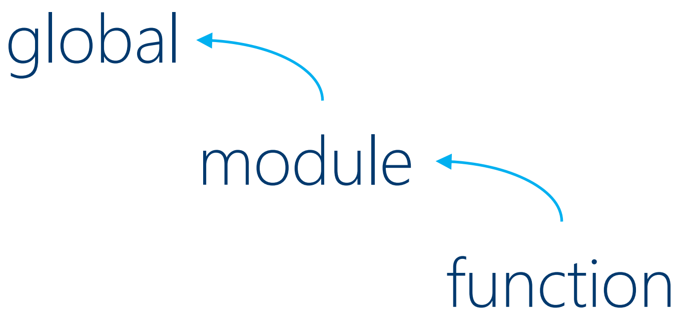
module

function

```
# module.py
NAME = 'Transylvania'

def func(arg):
  return NAME + ' ' + arg
```

```
>>> NAME = 'Hamunaptra`
>>> import module
```

Each module is a *namespace*



```
# module.py
NAME = 'Transylvania'

def func(arg):
  return NAME + ' ' + arg
```

```
>>> NAME = 'Hamunaptra`
>>> import module
>>> print(module.func('!!!'))
Transylvania !!!
```

Python comes with many standard libraries

Python comes with many standard libraries

```
>>> import math
```

Python comes with many standard libraries

```
>>> import math
>>> print(math.sqrt(2))
1.4142135623730951
```

Python comes with many standard libraries

```
>>> import math
>>> print(math.sqrt(2))
1.4142135623730951
>>> print(math.hypot(2, 3)) # sqrt(x**2 + y**2)
3.605551275463989l
```

Python comes with many standard libraries

```
>>> import math
>>> print(math.sqrt(2))
1.4142135623730951
>>> print(math.hypot(2, 3)) # sqrt(x**2 + y**2)
3.605551275463989l
>>> print(math.e, math.pi)  # as accurate as possible
2.718281828459045l 3.141592653589793l
```

Python also provides a help function

Python also provides a help function

```
>>> import math
>>> help(math)
Help on module math:
NAME
    math
FILE
    /usr/lib/python2.5/lib-dynload/math.so
MODULE DOCS
    http://www.python.org/doc/current/lib/module-math.html
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.
⋮
```

And some nicer ways to do imports

And some nicer ways to do imports

```
>>> from math import sqrt
>>> sqrt(3)
1.7320508075688772
```

And some nicer ways to do imports

```
>>> from math import sqrt
>>> sqrt(3)
1.7320508075688772
>>>from math import hypot as euclid
>>>euclid(3, 4)
5.0
```

And some nicer ways to do imports

```
>>>  from math import sqrt
>>>  sqrt(3)
1.7320508075688772
>>> from math import hypot as euclid
>>> euclid(3, 4)
5.0
>>> from math import *
>>> sin(pi)
1.2246063538223773e-16
>>>
```

And some nicer ways to do imports

```
>>> from math import sqrt
>>> sqrt(3)
1.7320508075688772
>>> from math import hypot as euclid
>>> euclid(3, 4)
5.0
>>> from math import *
>>> sin(pi)
1.2246063538223773e-16
>>>
```

Generally a bad idea

And some nicer ways to do imports

```
>>>  from math import sqrt
>>>  sqrt(3)
1.7320508075688772
>>> from math import hypot as euclid
>>> euclid(3, 4)
5.0
>>> from math import *
>>> sin(pi)
1.2246063538223773e-16
>>>
```

Generally a bad idea

Someone could add to the library after you start using it

Almost every program uses the sys library

Almost every program uses the sys library

```
>>> import sys
```

Almost every program uses the sys library

```
>>> import sys
>>> print(sys.version)
3.4.3 |Anaconda 2.3.0 (64-bit)| (default, Jun 4 2015, 15:29:08)
[GCC 4.4.7 20120313 (Red Hat 4.4.7-1)]
```

Almost every program uses the sys library

```
>>> import sys
>>> print(sys.version)
3.4.3 |Anaconda 2.3.0 (64-bit)| (default, Jun 4 2015, 15:29:08)
[GCC 4.4.7 20120313 (Red Hat 4.4.7-1)]
>>> print(sys.platform)
linux
```

Almost every program uses the sys library

```
>>> import sys
>>> print(sys.version)
3.4.3 |Anaconda 2.3.0 (64-bit)| (default, Jun 4 2015, 15:29:08)
[GCC 4.4.7 20120313 (Red Hat 4.4.7-1)]
>>> print(sys.platform)
linux
>>> print(sys.maxsize)
9223372036854775807
```

Almost every program uses the sys library

```
>>> import sys
>>> print(sys.version)
3.4.3 |Anaconda 2.3.0 (64-bit)| (default, Jun 4 2015, 15:29:08)
[GCC 4.4.7 20120313 (Red Hat 4.4.7-1)]
>>> print(sys.platform)
linux
>>> print(sys.maxsize)
9223372036854775807
>>> print(sys.path)
['', '/home/asobhani/anaconda3/lib/python34.zip', '/home
/asobhani/anaconda3/lib/python3.4', '/home/asobhani/anaconda3
/lib/python3.4/plat-linux', '/home/asobhani/anaconda3/lib
/python3.4/lib-dynload', '/home/asobhani/anaconda3/lib
/python3.4/site-packages', '/home/asobhani/anaconda3/lib
/python3.4/site-packages/Sphinx-1.3.1-py3.4.egg']
```

`sys.argv` holds command-line arguments

`sys.argv` holds command-line arguments

Script name is `sys.argv[0]`

`sys.argv` holds command-line arguments

Script name is `sys.argv[0]`

```
# echo.py
import sys
for i in range(len(sys.argv)):
  print(i, '"' + sys.argv[i] + '"')
```

`sys.argv` holds command-line arguments

Script name is `sys.argv[0]`

```
# echo.py
import sys
for i in range(len(sys.argv)):
  print(i, '"' + sys.argv[i] + '"')
```

```
$ python echo.py
0 "echo.py"
$
```

`sys.argv` holds command-line arguments

Script name is `sys.argv[0]`

```
# echo.py
import sys
for i in range(len(sys.argv)):
  print(i, '"' + sys.argv[i] + '"')
```

```
$ python echo.py
0 "echo.py"
$ python echo.py first second
0 "echo.py"
1 "first"
2 "second"
$
```

`sys.stdin` is *standard input* (e.g., the keyboard)

`sys.stdin` is *standard input* (e.g., the keyboard)

`sys.stdout` is *standard output* (e.g., the screen)

`sys.stdin` is *standard input* (e.g., the keyboard)

`sys.stdout` is *standard output* (e.g., the screen)

`sys.stderr` is *standard error* (usually also the screen)

```python
# count.py
import sys

def count_lines(reader):
  return len(reader.readlines())

if len(sys.argv) == 1:
  print(count_lines(sys.stdin))
else:
  rd = open(sys.argv[1], 'r')
  print(count_lines(rd))
  rd.close()
```

```
# count.py
import sys

def count_lines(reader):
  return len(reader.readlines())

if len(sys.argv) == 1:
  print(count_lines(sys.stdin))
else:
  rd = open(sys.argv[1], 'r')
  print(count_lines(rd))
  rd.close()
```

```
# count.py
import sys

def count_lines(reader):
  return len(reader.readlines())

if len(sys.argv) == 1:
  print(count_lines(sys.stdin))
else:
  rd = open(sys.argv[1], 'r')
  print(count_lines(rd))
  rd.close()
```

```
# count.py
import sys

def count_lines(reader):
  return len(reader.readlines())

if len(sys.argv) == 1:
  print(count_lines(sys.stdin))
else:
  rd = open(sys.argv[1], 'r')
  print(count_lines(rd))
  rd.close()
```

```
$ python count.py < a.txt
48
$
```

```python
# count.py
import sys

def count_lines(reader):
  return len(reader.readlines())

if len(sys.argv) == 1:
  print(count_lines(sys.stdin))
else:
  rd = open(sys.argv[1], 'r')
  print(count_lines(rd))
  rd.close()
```

```
$ python count.py < a.txt
48
$ python count.py b.txt
227
$
```

The more polite way

```
'''Count lines in files.  If no filename
arguments given, read from standard input.'''

import sys

def count_lines(reader):
  '''Return number of lines in text read from reader.'''
  return len(reader.readlines())

if __name__ == '__main__':
  ...as before...
```

The more polite way

```
'''Count lines in files.  If no filename
arguments given, read from standard input.'''

import sys

def count_lines(reader):
    '''Return number of lines in text read from reader.'''
    return len(reader.readlines())

if __name__ == '__main__':
    ...as before...
```

The more polite way

```
'''Count lines in files.  If no filename
arguments given, read from standard input.'''

import sys

def count_lines(reader):
  '''Return number of lines in text read from reader.'''
  return len(reader.readlines())

if __name__ == '__main__':
  ...as before...
```

If the first statement in a module or function is a string, it is saved as a *docstring*

If the first statement in a module or function is a string, it is saved as a *docstring*

Used for online (and offline) help

If the first statement in a module or function is a string, it is saved as a *docstring*

Used for online (and offline) help

```
# adder.py
'''Addition utilities.'''

def add(a, b):
    '''Add arguments.'''
    return a+b
```

If the first statement in a module or function is a string, it is saved as a *docstring*

Used for online (and offline) help

```python
# adder.py
'''Addition utilities.'''

def add(a, b):
  '''Add arguments.'''
  return a+b
```

```
>>> import adder
>>> help(adder)
NAME
  adder - Addition utilities.
FUNCTIONS
  add(a, b)
      Add arguments.
>>>
```

If the first statement in a module or function is a string, it is saved as a *docstring*

Used for online (and offline) help

```
# adder.py
'''Addition utilities.'''

def add(a, b):
  '''Add arguments.'''
  return a+b
```

```
>>> import adder
>>> help(adder)
NAME
   adder - Addition utilities.
FUNCTIONS
   add(a, b)
         Add arguments.
>>> help(adder.add)
add(a, b)
          Add arguments.
>>>
```

When Python loads a module, it assigns a value to the module-level variable __name__

When Python loads a module, it assigns a value to the module-level variable __name__

main program
'__main__'

When Python loads a module, it assigns a value to the module-level variable __name__

main program	loaded as library
'__main__'	module name

When Python loads a module, it assigns a value to the module-level variable __name__

main program	loaded as library
'__main__'	module name

```
...module definitions...

if __name__ == '__main__':
  ...run as main program...
```

When Python loads a module, it assigns a value to the module-level variable __name__

main program	loaded as library
'__main__'	module name

```
...module definitions...

if __name__ == '__main__':
   ...run as main program...
```

← Always executed

When Python loads a module, it assigns a value to the module-level variable __name__

main program	loaded as library
'__main__'	module name

```
...module definitions...

if __name__ == '__main__':
  ...run as main program...
```

Always executed

Only executed when file run directly

```
# stats.py
'''Useful statistical tools.'''

def average(values):
  '''Return average of values or None if no data.'''
  if values:
    return sum(values) / len(values)
  else:
    return None

if __name__ == '__main__':
  print('test 1 should be None:', average([]))
  print('test 2 should be 1:', average([1]))
  print('test 3 should be 2:', average([1, 2, 3]))
```

```
# test-stats.py
from stats import average
print('test 4 should be None:', average(set()))
print('test 5 should be -1:', average({0, -1, -2}))
```

```python
# test-stats.py
from stats import average
print('test 4 should be None:', average(set()))
print('test 5 should be -1:', average({0, -1, -2}))
```

```
$ python stats.py
test 1 should be None: None
test 2 should be 1: 1
test 3 should be 2: 2
$
```

```
# test-stats.py
from stats import average
print('test 4 should be None:', average(set()))
print('test 5 should be -1:', average({0, -1, -2}))
```

```
$ python stats.py
test 1 should be None: None
test 2 should be 1: 1
test 3 should be 2: 2
$ python test-stats.py
test 4 should be None: None
test 5 should be -1: -1
$
```

Python

Slicing

by Greg Wilson

UNIVERSITY OF ONTARIO INSTITUTE OF TECHNOLOGY

Lists, strings, and tuples are all *sequences*

Lists, strings, and tuples are all *sequences*

Can be indexed by integers in the range 0…len(X)-1

Lists, strings, and tuples are all *sequences*

Can be indexed by integers in the range 0...len(X)-1

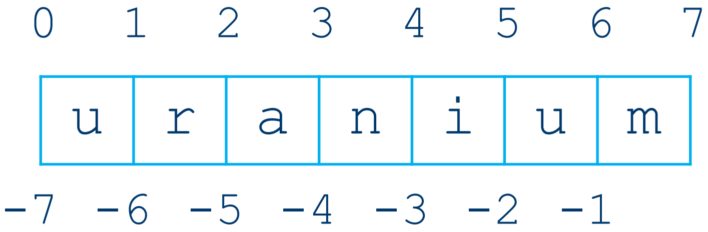
Can also be sliced using a range of indices

Lists, strings, and tuples are all *sequences*

Can be indexed by integers in the range 0...len(X)-1

Can also be sliced using a range of indices

```
>>> element = 'uranium'
>>>
```



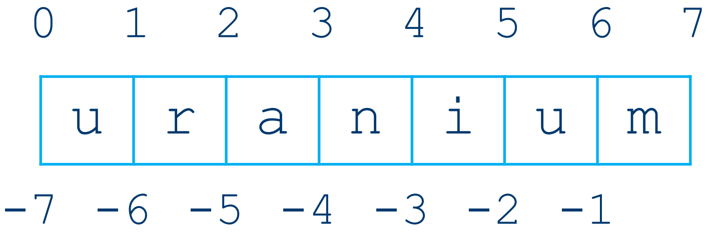
```
0   1   2   3   4   5   6   7
  u   r   a   n   i   u   m
-7  -6  -5  -4  -3  -2  -1
```

Lists, strings, and tuples are all *sequences*

Can be indexed by integers in the range 0…len(X)-1

Can also be sliced using a range of indices

```
>>> element = 'uranium'
>>> print(element[1:4])
ran
>>>
```



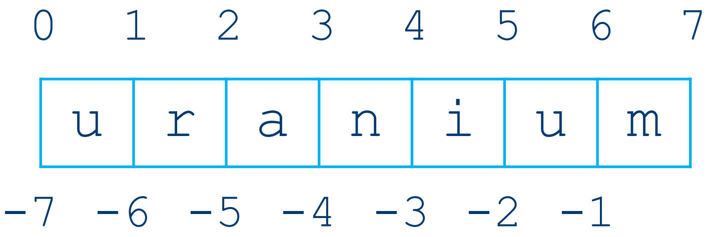
```
 0   1   2   3   4   5   6   7
 | u | r | a | n | i | u | m |
-7  -6  -5  -4  -3  -2  -1
```

Lists, strings, and tuples are all *sequences*

Can be indexed by integers in the range 0…len(X)-1

Can also be sliced using a range of indices

```
>>> element = 'uranium'
>>> print(element[1:4])
ran
>>> print(element[:4])
uran
>>>
```



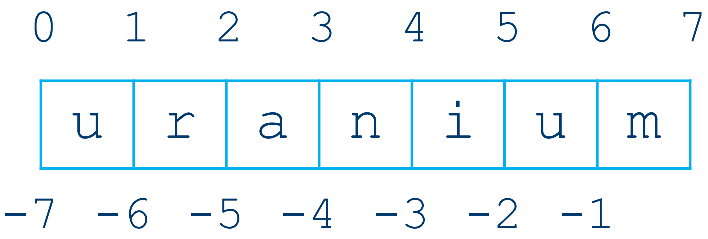
```
 0   1   2   3   4   5   6   7
 | u | r | a | n | i | u | m |
-7  -6  -5  -4  -3  -2  -1
```

Lists, strings, and tuples are all *sequences*

Can be indexed by integers in the range 0...len(X)-1

Can also be sliced using a range of indices

```
>>> element = 'uranium'
>>> print(element[1:4])
ran
>>> print(element[:4])
uran
>>> print(element[4:])
ium
>>>
```



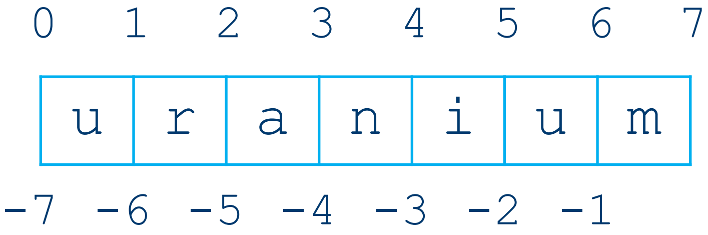
```
 0   1   2   3   4   5   6   7
 | u | r | a | n | i | u | m |
-7  -6  -5  -4  -3  -2  -1
```

Lists, strings, and tuples are all *sequences*

Can be indexed by integers in the range 0…len(X)-1

Can also be sliced using a range of indices

```
>>> element = 'uranium'
>>> print(element[1:4])
ran
>>> print(element[:4])
uran
>>> print(element[4:])
ium
>>> print(element[-4:])
nium
>>>
```



0 1 2 3 4 5 6 7

u r a n i u m

-7 -6 -5 -4 -3 -2 -1

Python checks bounds when indexing

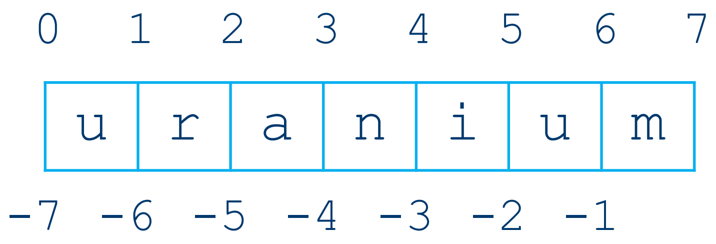
Python checks bounds when indexing

But truncates when slicing

Python checks bounds when indexing

But truncates when slicing

```
>>> element = 'uranium'
>>>
```

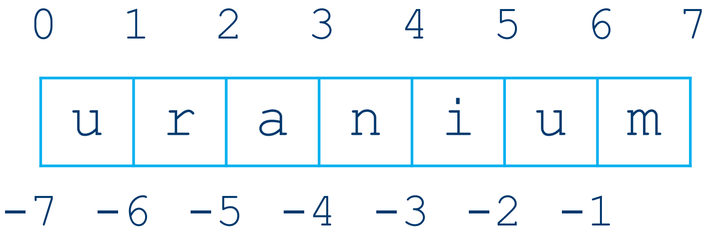


```
 0   1   2   3   4   5   6   7
 u   r   a   n   i   u   m
-7  -6  -5  -4  -3  -2  -1
```

Python checks bounds when indexing

But truncates when slicing

```
>>> element = 'uranium'
>>> print(element[400])
```



0 1 2 3 4 5 6 7

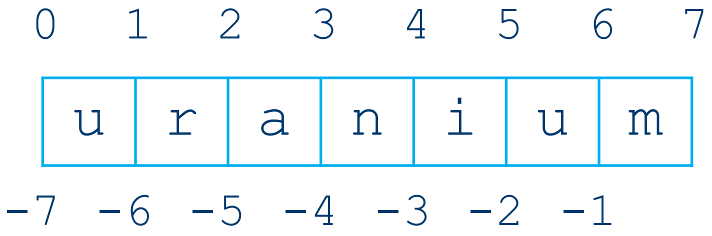
u r a n i u m

-7 -6 -5 -4 -3 -2 -1

Python checks bounds when indexing

But truncates when slicing

```
>>> element = 'uranium'
>>> print(element[400])
IndexError: string index out of range
>>>
```

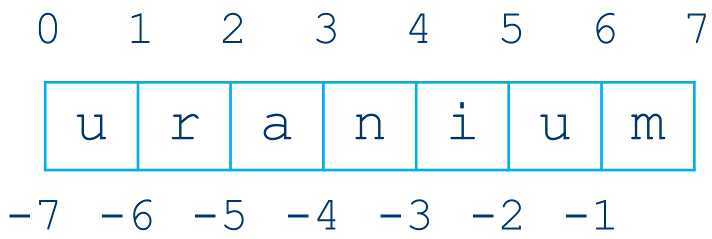


```
 0   1   2   3   4   5   6   7
 | u | r | a | n | i | u | m |
-7  -6  -5  -4  -3  -2  -1
```

Python checks bounds when indexing

But truncates when slicing

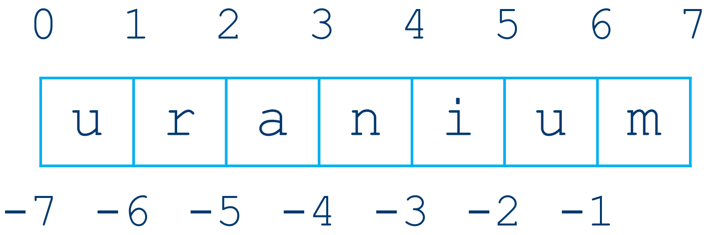
```
>>> element = 'uranium'
>>> print(element[400])
IndexError: string index out of range
>>> print(element[1:400])
>>> ranium
>>>
```



Python checks bounds when indexing

But truncates when slicing

```
>>> element = 'uranium'
>>> print(element[400])
IndexError: string index out of range
>>> print(element[1:400])
>>> ranium
>>>
```



```
0   1   2   3   4   5   6   7
  u   r   a   n   i   u   m
-7  -6  -5  -4  -3  -2  -1
```

"A foolish consistency is
the hobgoblin of little minds."

— *Ralph Waldo Emerson*

Python checks bounds when indexing

But truncates when slicing

```
>>> element = 'uranium'
>>> print(element[400])
IndexError: string index out of range
>>> print(element[1:400])
>>> ranium
>>>
```

```
 0   1   2   3   4   5   6   7
 | u | r | a | n | i | u | m |
-7  -6  -5  -4  -3  -2  -1
```

"A foolish consistency is
the hobgoblin of little minds."

— *Ralph Waldo Emerson*

"Aw, you're kidding me!"

— *programmers*

So `text[1:3]` is `0`, `1`, or `2` characters long

So `text[1:3]` is 0, 1, or 2 characters long

`''`	`''`
`'a'`	`''`
`'ab'`	`'b'`
`'abc'`	`'bc'`
`'abcdef'`	`'bc'`

For consistency, `text[1:1]` is the empty string

For consistency, `text[1:1]` is the empty string

- From index 1 up to (but not including) index 1

For consistency, `text[1:1]` is the empty string

- From index 1 up to (but not including) index 1

And `text[3:1]` is always the empty string

For consistency, `text[1:1]` is the empty string

- From index 1 up to (but not including) index 1

And `text[3:1]` is always the empty string

- *Not* the reverse of `text[1:3]`

For consistency, `text[1:1]` is the empty string

- From index 1 up to (but not including) index 1

And `text[3:1]` is always the empty string

- *Not* the reverse of `text[1:3]`

But `text[1:-1]` is everything except the first and last characters

Slicing always creates a new collection

Slicing always creates a new collection

Beware of aliasing

Slicing always creates a new collection

Beware of aliasing

```
>>> points = [[10, 10], [20, 20], [30, 30], [40, 40]]
>>>
```

Slicing always creates a new collection

Beware of aliasing

```
>>> points = [[10, 10], [20, 20], [30, 30], [40, 40]]
>>> middle = points[1:-1]
>>>
```

Slicing always creates a new collection

Beware of aliasing

```
>>> points = [[10, 10], [20, 20], [30, 30], [40, 40]]
>>> middle = points[1:-1]
>>> middle[0][0] = 'whoops'
>>>
```

Slicing always creates a new collection

Beware of aliasing

```
>>> points = [[10, 10], [20, 20], [30, 30], [40, 40]]
>>> middle = points[1:-1]
>>> middle[0][0] = 'whoops'
>>> middle[1][0] = 'aliasing'
>>>
```

Slicing always creates a new collection

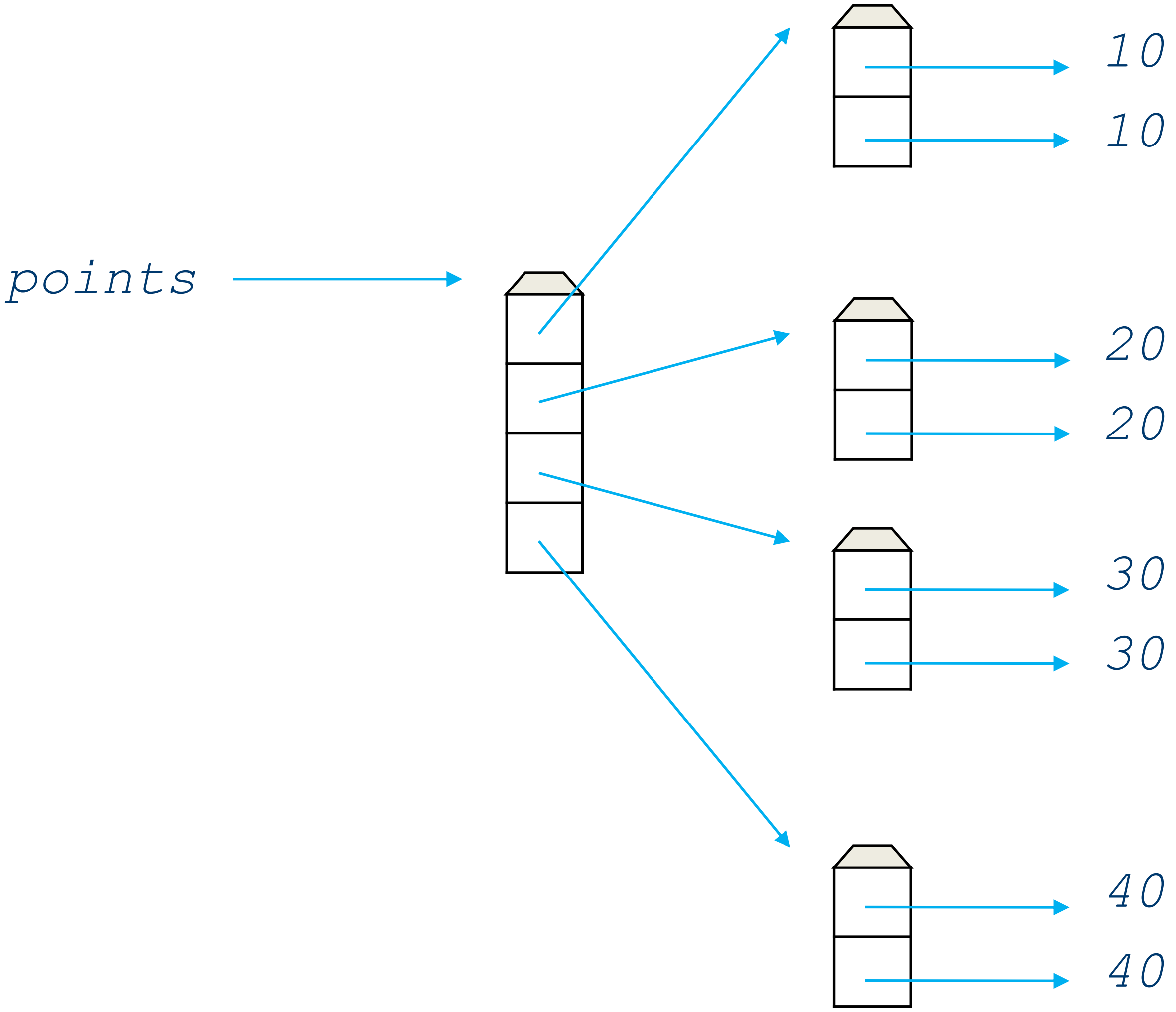
Beware of aliasing

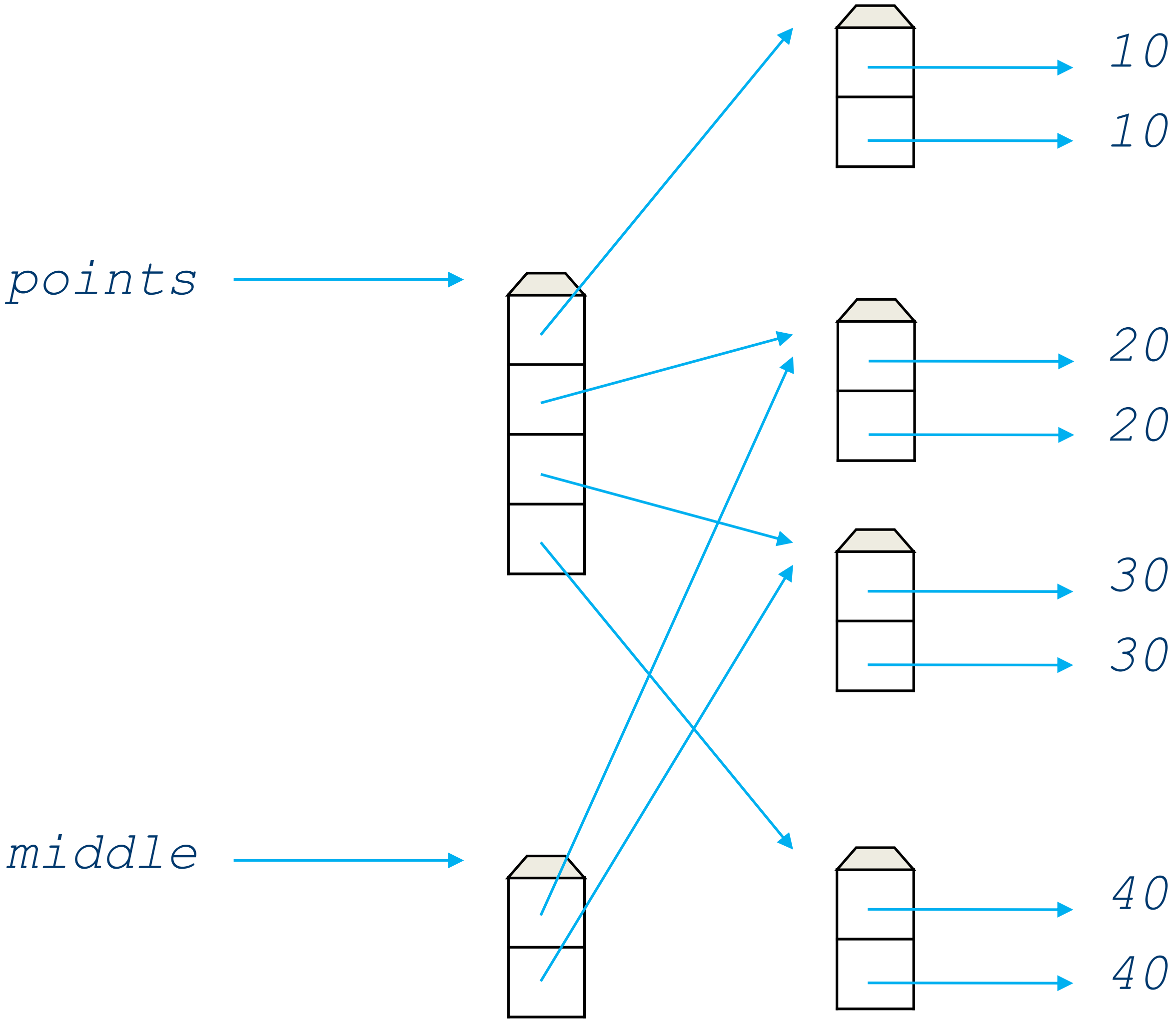
```
>>> points = [[10, 10], [20, 20], [30, 30], [40, 40]]
>>> middle = points[1:-1]
>>> middle[0][0] = 'whoops'
>>> middle[1][0] = 'aliasing'
>>> print(middle)
>>> [['whoops', 20], ['aliasing', 30]]
>>>
```

Slicing always creates a new collection

Beware of aliasing

```
>>> points = [[10, 10], [20, 20], [30, 30], [40, 40]]
>>> middle = points[1:-1]
>>> middle[0][0] = 'whoops'
>>> middle[1][0] = 'aliasing'
>>> print(middle)
>>> [['whoops', 20], ['aliasing', 30]]
>>> print(points)
[[10, 10], ['whoops', 20], ['aliasing', 30], [40, 40]]
>>>
```


points
10
10
20
20
30
30
40
40



points

middle

10

10

20

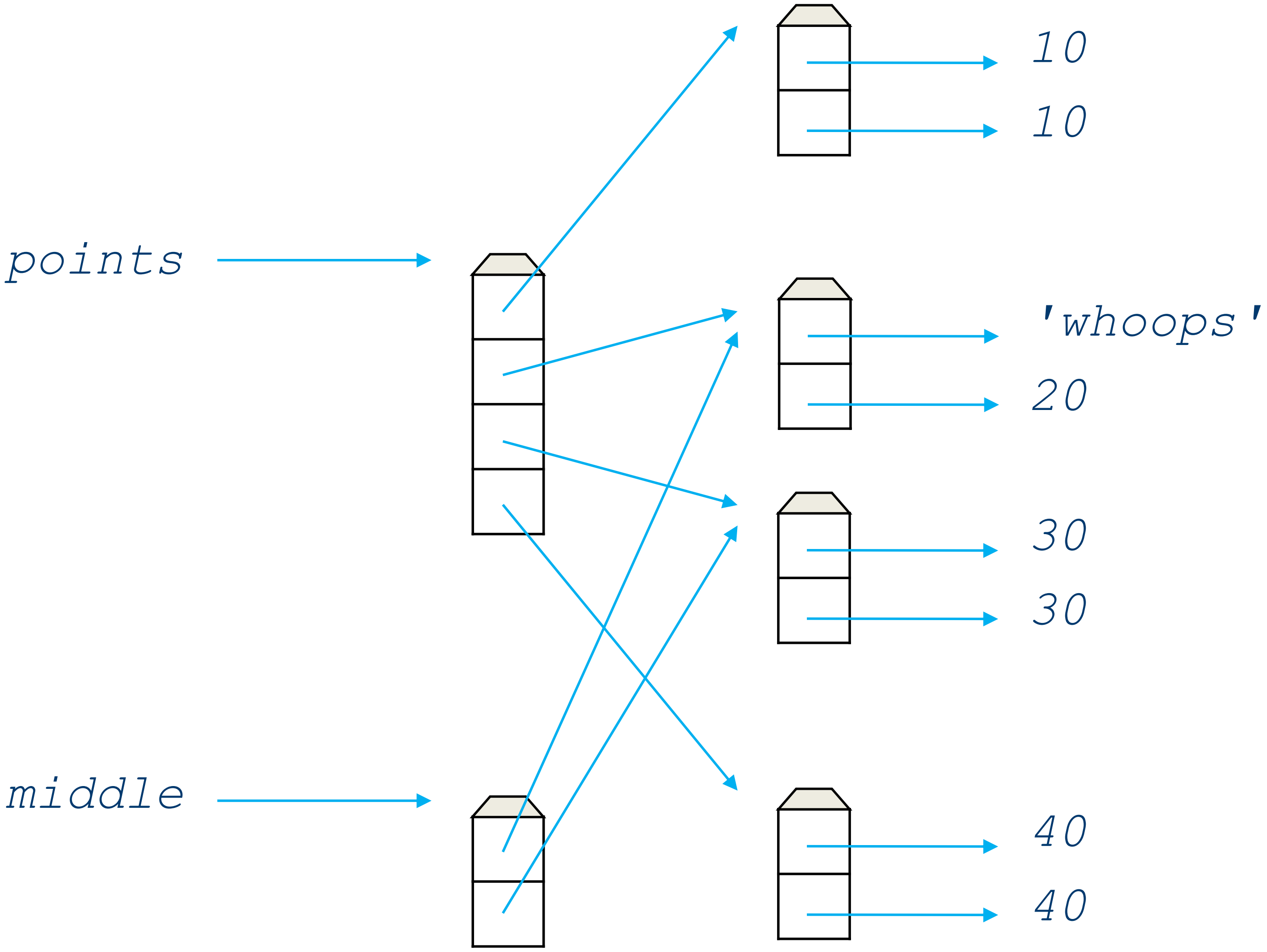
20

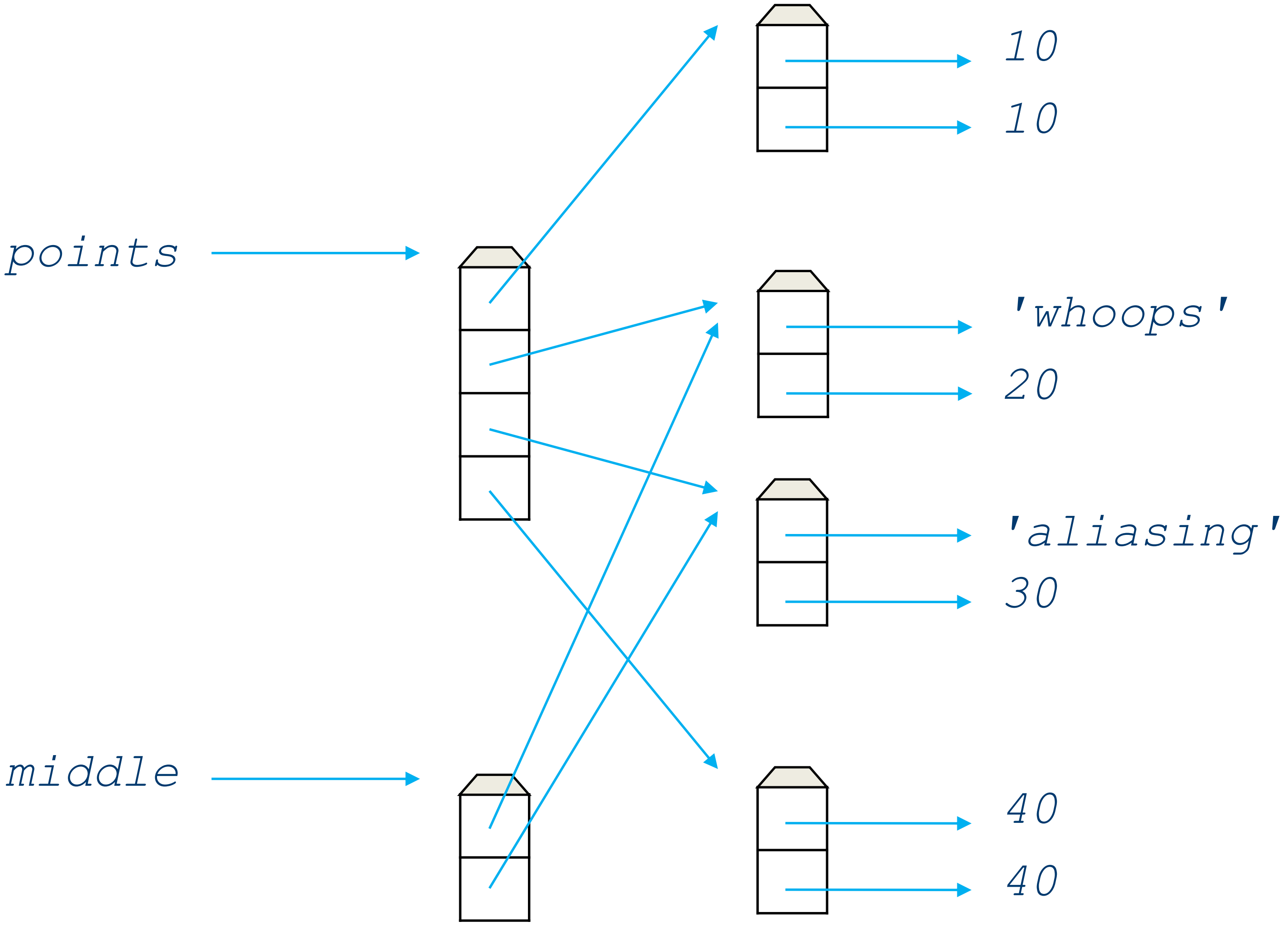
30

30

40

40


points
middle
10
10
'whoops'
20
30
30
40
40


points
middle
10
10
'whoops'
20
'aliasing'
30
40
40