ECE-203 – Programming for Engineers

Contact

Dr. James Shackleford
shack@drexel.edu
Bossone 211

Office Hours: 3 – 4 pm (Tuesday)
Course Website: http://learn.dcollege.net

Textbook

Think Python
by Allen Downey
O'Reilly Press, 2015
ISBN-13: 978-1449330729
(Freely available in PDF format, check course website)

Grading

- 10% In-lab Programming Assignments
- 10% Take-Home Programming Assignments
- 35% Mid-term Exam
- 45% Final Exam

ANNOUNCEMENT

MIDTERM EXAM

NEXT WEEK IN CLASS

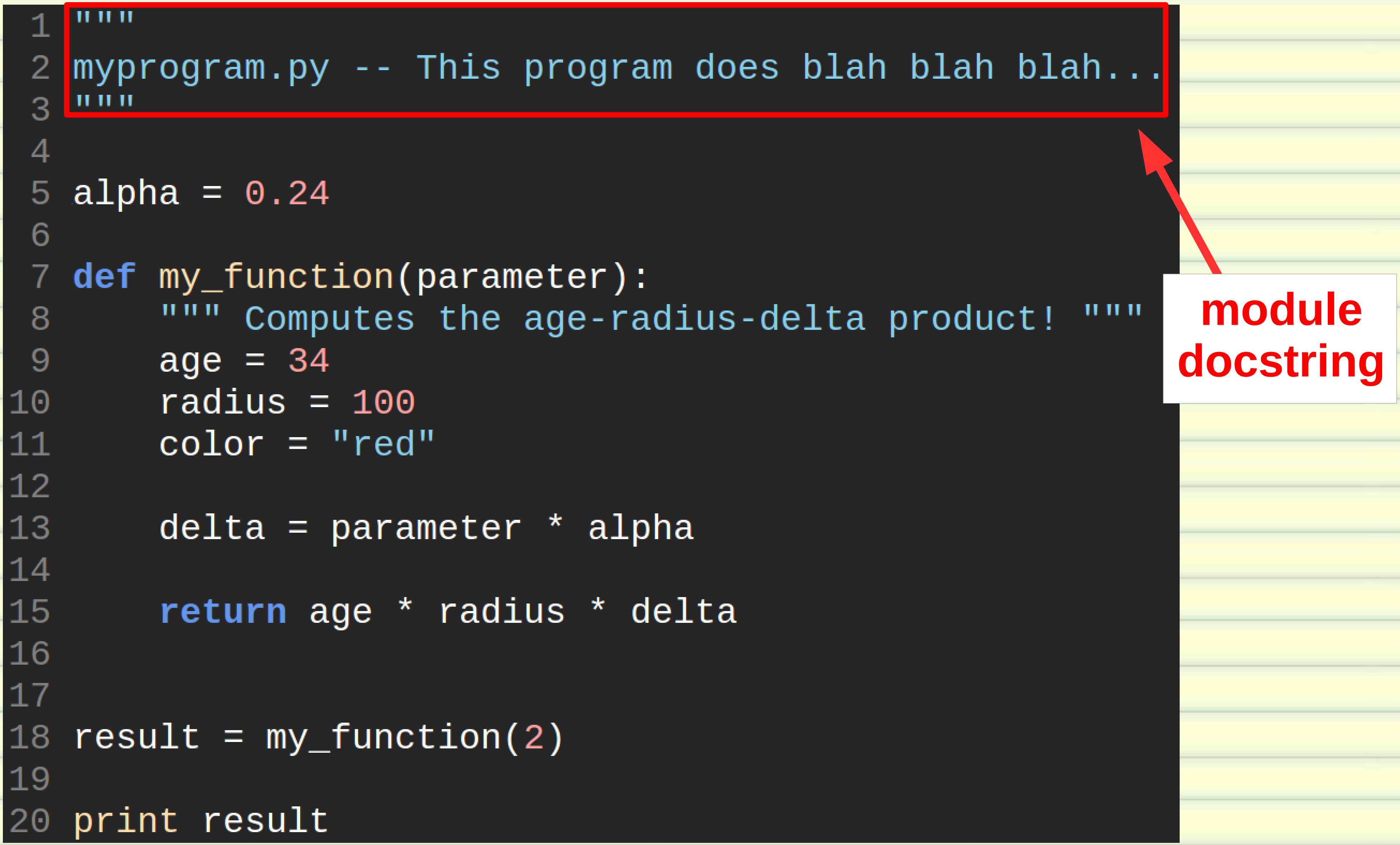
THURSDAY, FEB. 16th

ALSO:
NO HOMEWORK THIS WEEK
(aside from midterm preparation)

Review Stuff

REVIEW STUFF!!!

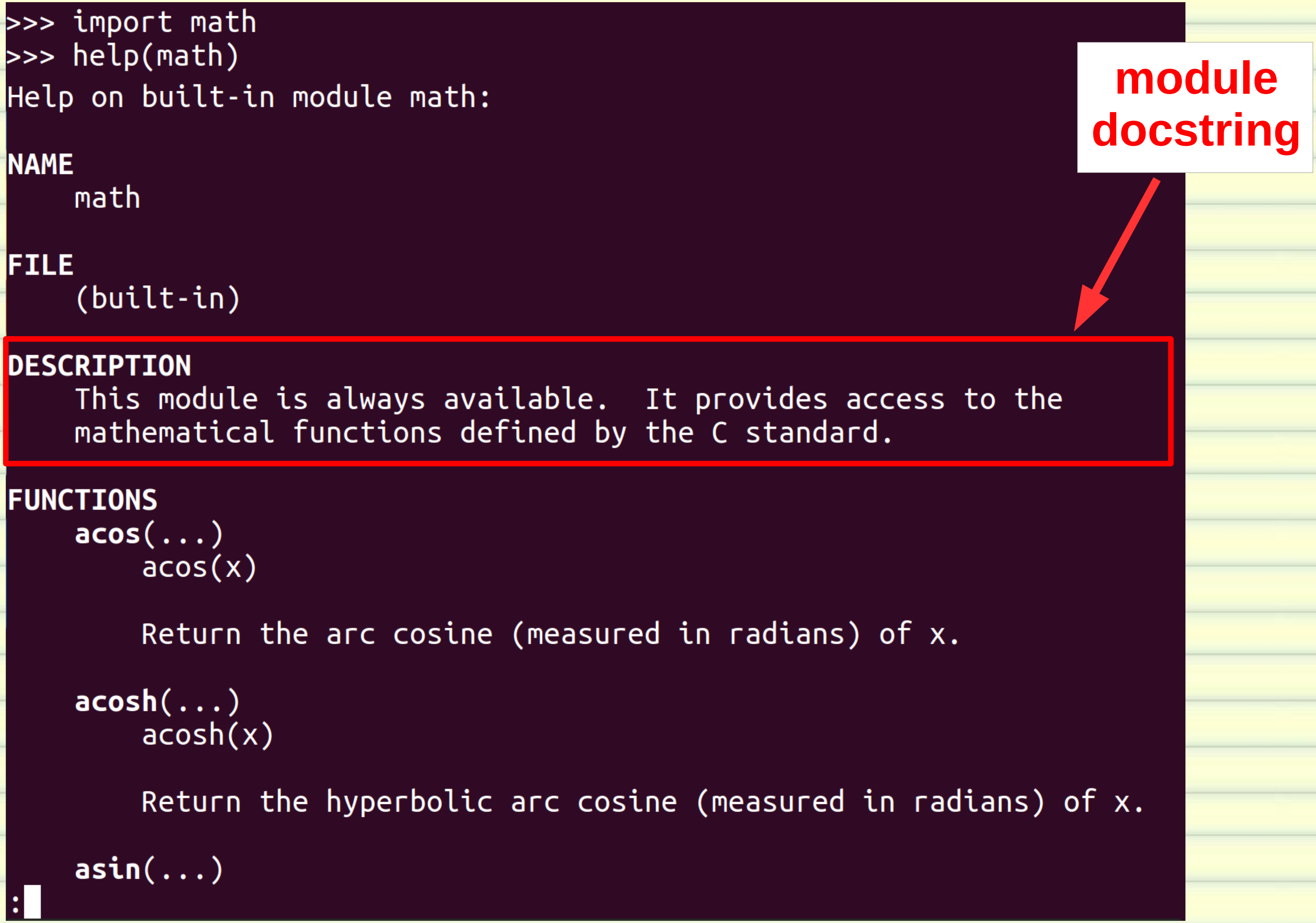
Anatomy of an (almost) “proper” Python program



```python
"""
myprogram.py -- This program does blah blah blah...
"""

alpha = 0.24

def my_function(parameter):
    """ Computes the age-radius-delta product! """
    age = 34
    radius = 100
    color = "red"

    delta = parameter * alpha

    return age * radius * delta


result = my_function(2)

print result
```

module docstring

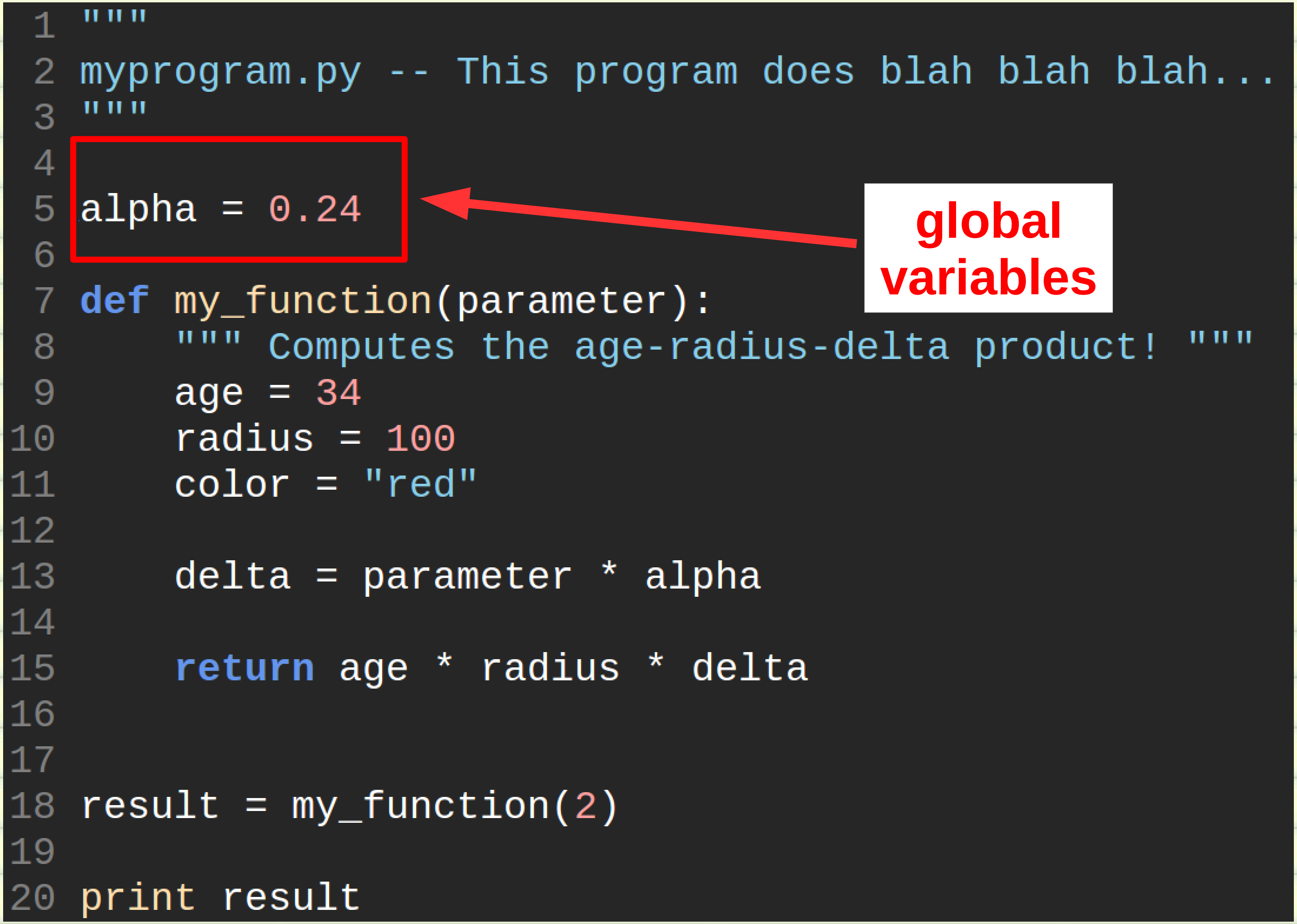
Anatomy of an (almost) “proper” Python program



```
>>> import math
>>> help(math)
Help on built-in module math:

NAME
    math

FILE
    (built-in)

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)

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

    asin(...)
:
```

module docstring

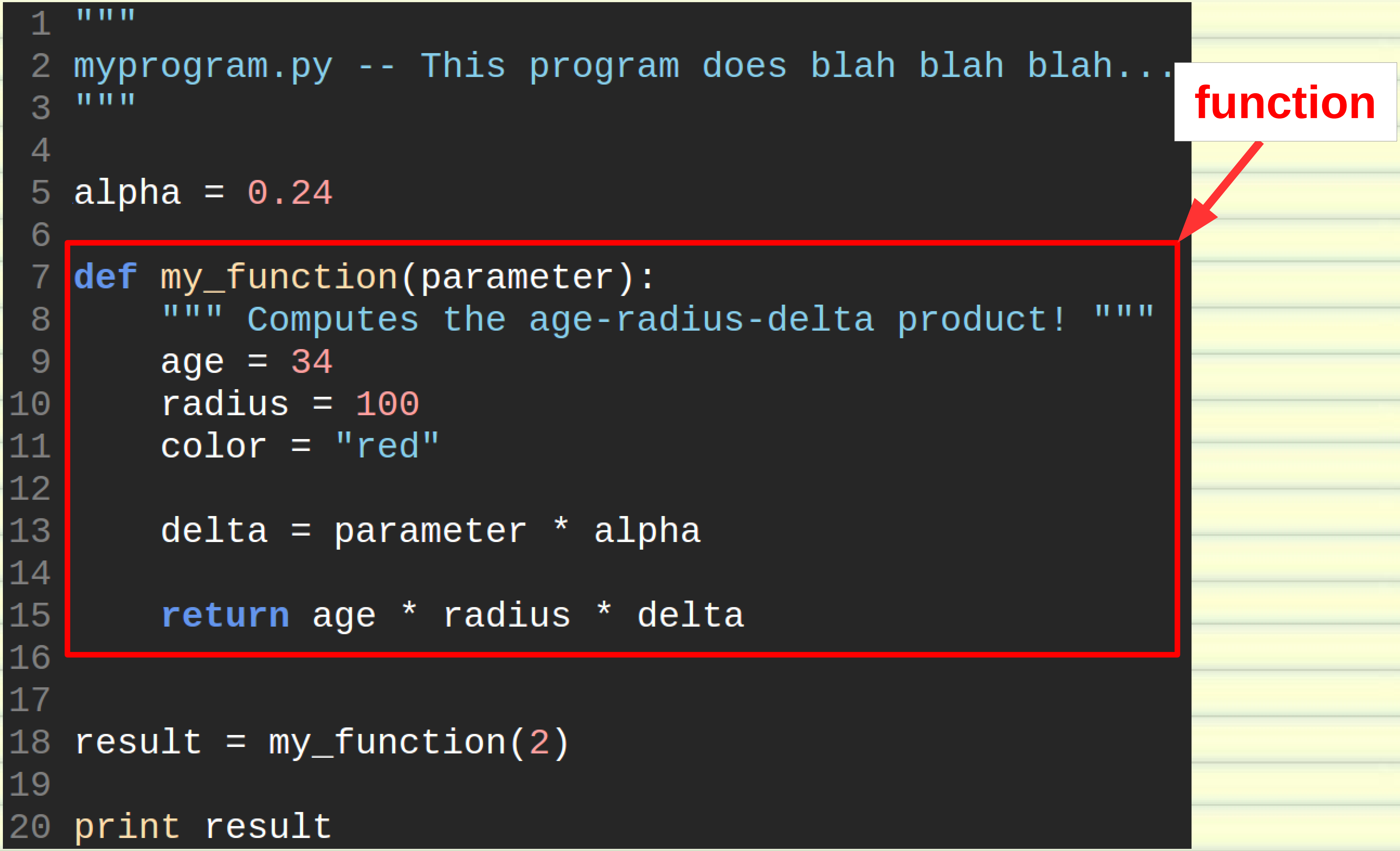
Anatomy of an (almost) “proper” Python program



```
"""
myprogram.py -- This program does blah blah blah...
"""

alpha = 0.24

def my_function(parameter):
    """ Computes the age-radius-delta product! """
    age = 34
    radius = 100
    color = "red"

    delta = parameter * alpha

    return age * radius * delta


result = my_function(2)

print result
```

global variables

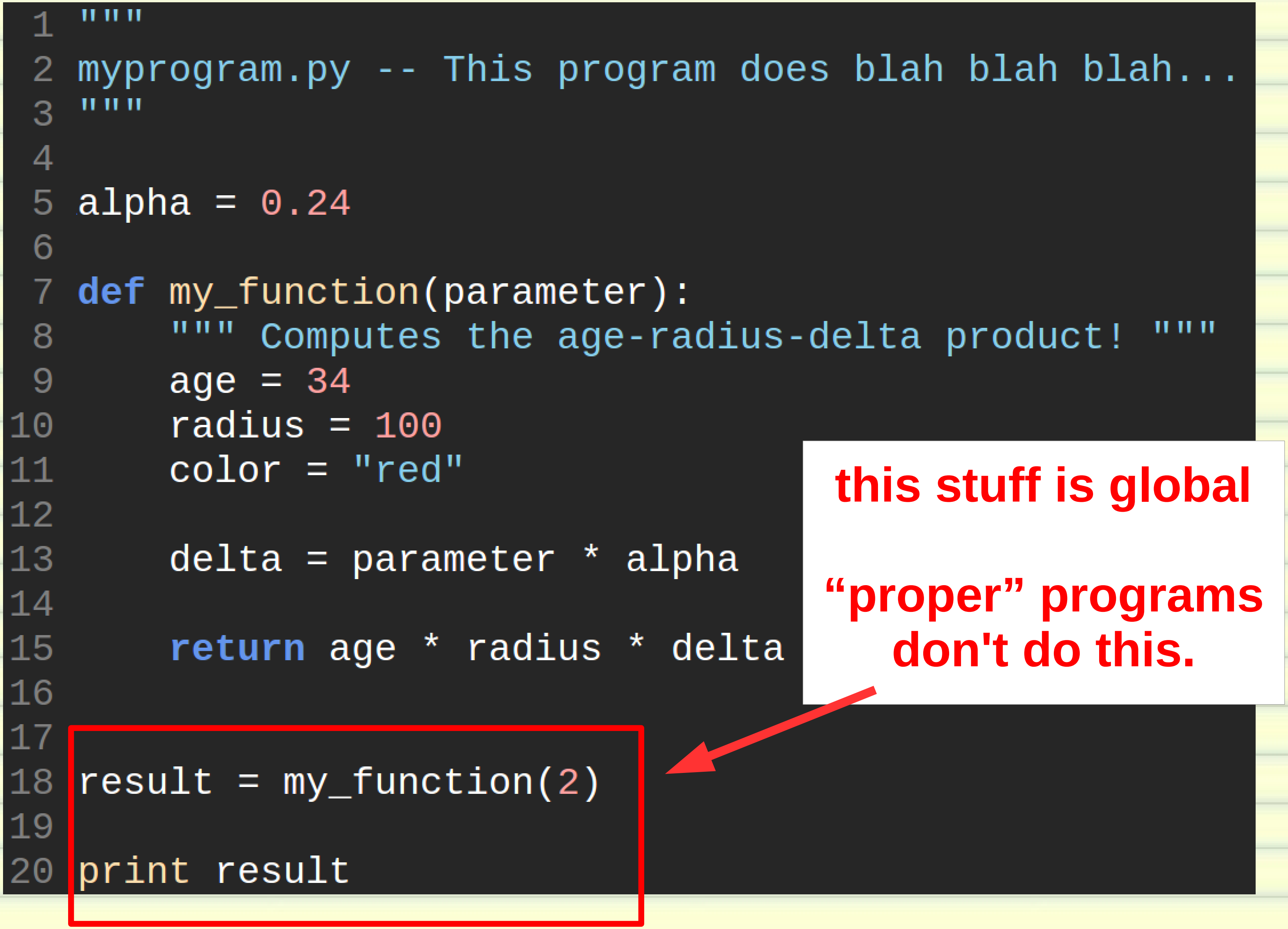
Anatomy of an (almost) “proper” Python program



```
"""
myprogram.py -- This program does blah blah blah...
"""

alpha = 0.24

def my_function(parameter):
    """ Computes the age-radius-delta product! """
    age = 34
    radius = 100
    color = "red"

    delta = parameter * alpha

    return age * radius * delta


result = my_function(2)

print result
```

function

Anatomy of an (almost) “proper” Python program

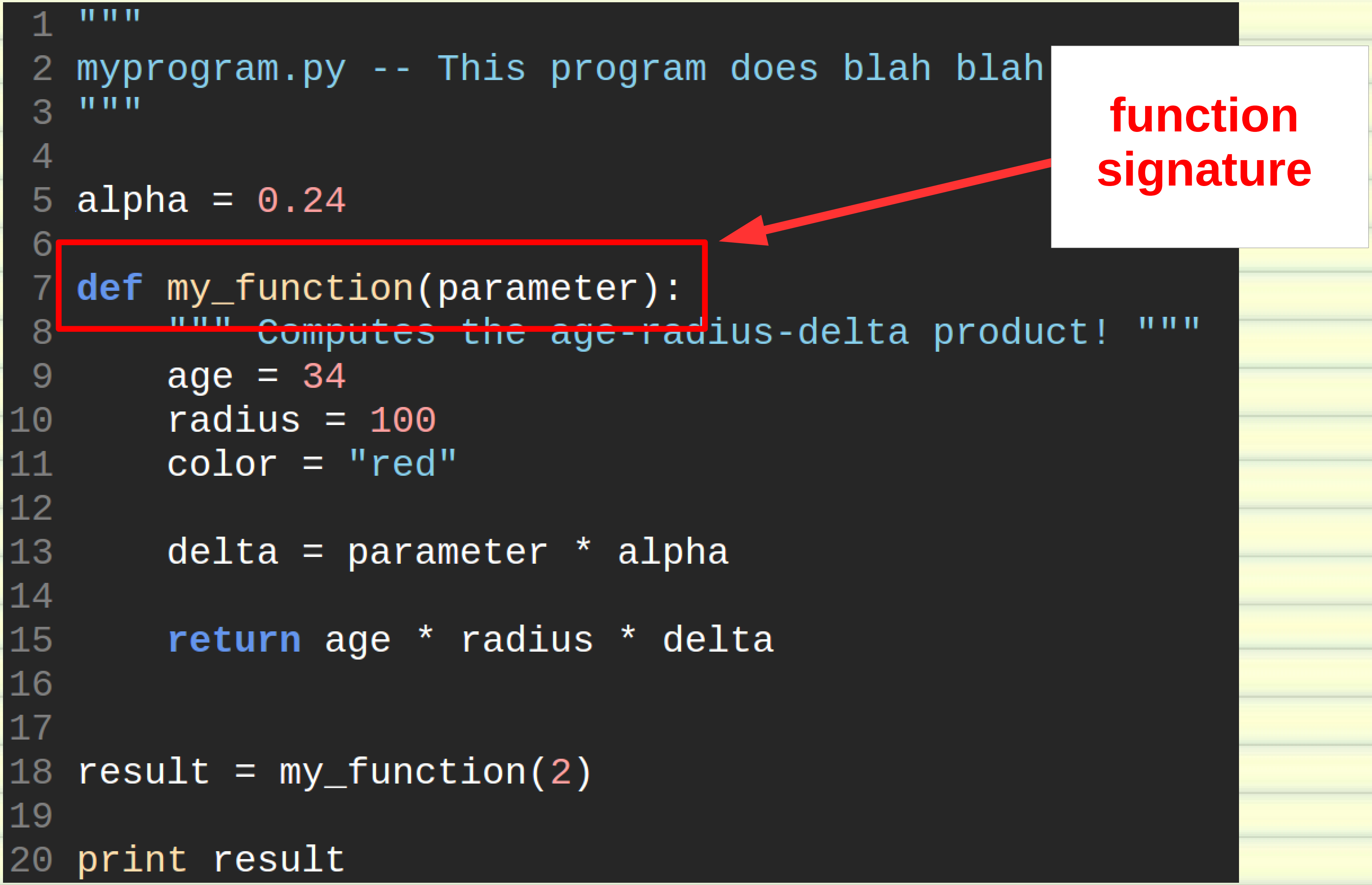


```python
"""
myprogram.py -- This program does blah blah blah...
"""

alpha = 0.24

def my_function(parameter):
    """ Computes the age-radius-delta product! """
    age = 34
    radius = 100
    color = "red"

    delta = parameter * alpha

    return age * radius * delta


result = my_function(2)

print result
```

this stuff is global

“proper” programs don't do this.

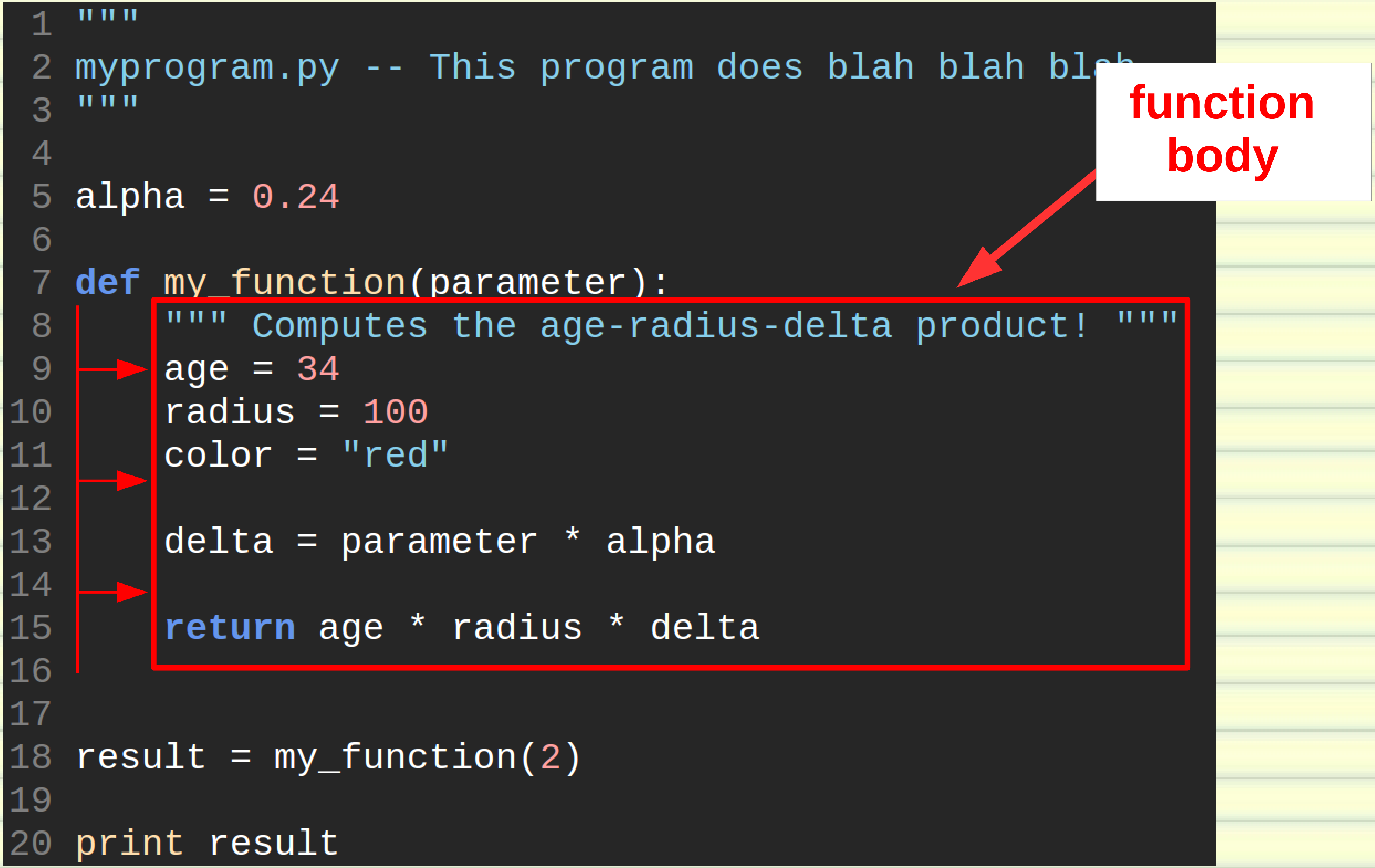
Anatomy of a Python function



```
"""
myprogram.py -- This program does blah blah
"""

alpha = 0.24

def my_function(parameter):
    """ Computes the age-radius-delta product! """
    age = 34
    radius = 100
    color = "red"

    delta = parameter * alpha

    return age * radius * delta


result = my_function(2)

print result
```

function signature

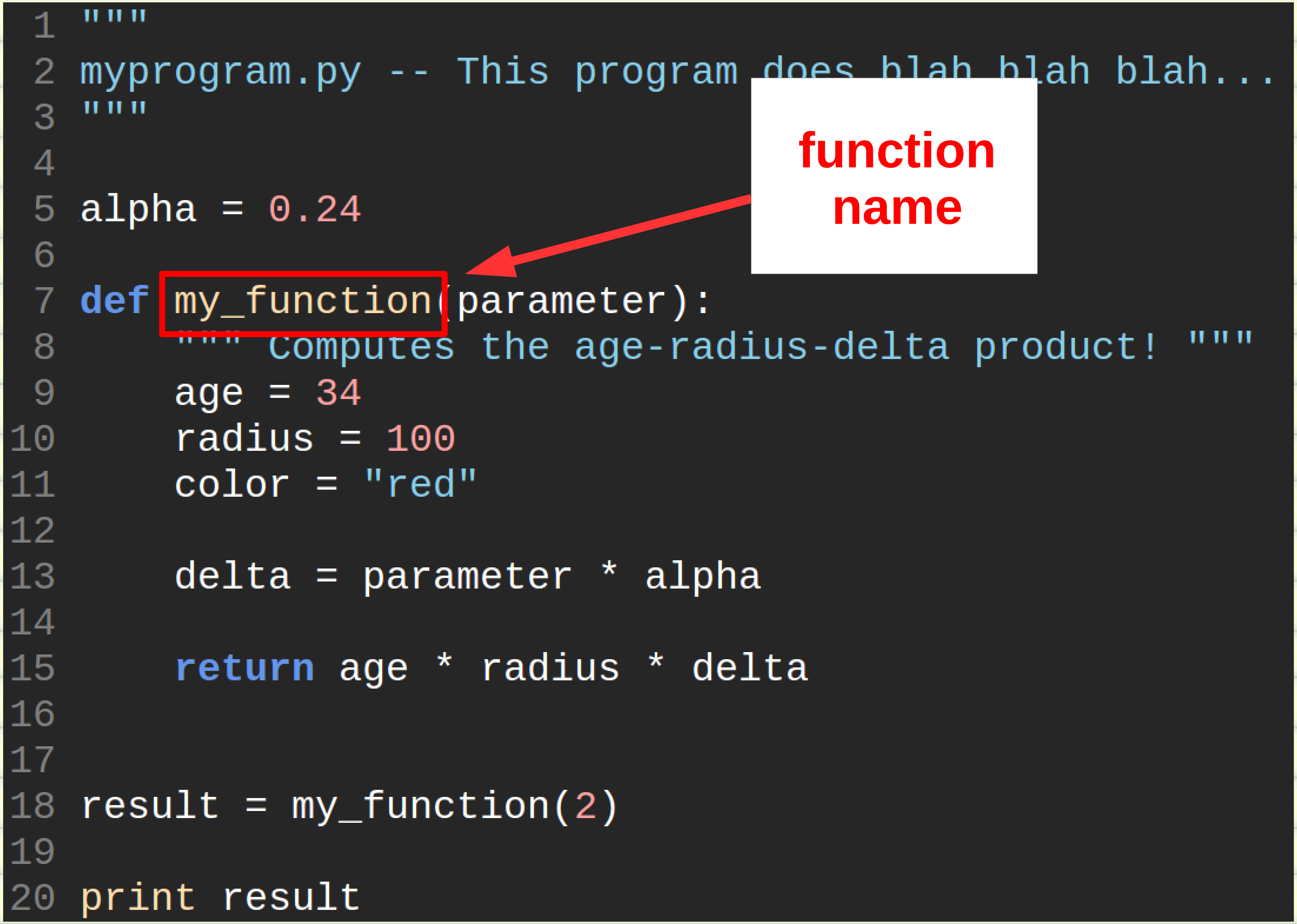
Anatomy of a Python function



```
"""
myprogram.py -- This program does blah blah blah
"""

alpha = 0.24

def my_function(parameter):
    """ Computes the age-radius-delta product! """
    age = 34
    radius = 100
    color = "red"

    delta = parameter * alpha

    return age * radius * delta


result = my_function(2)

print result
```

function body

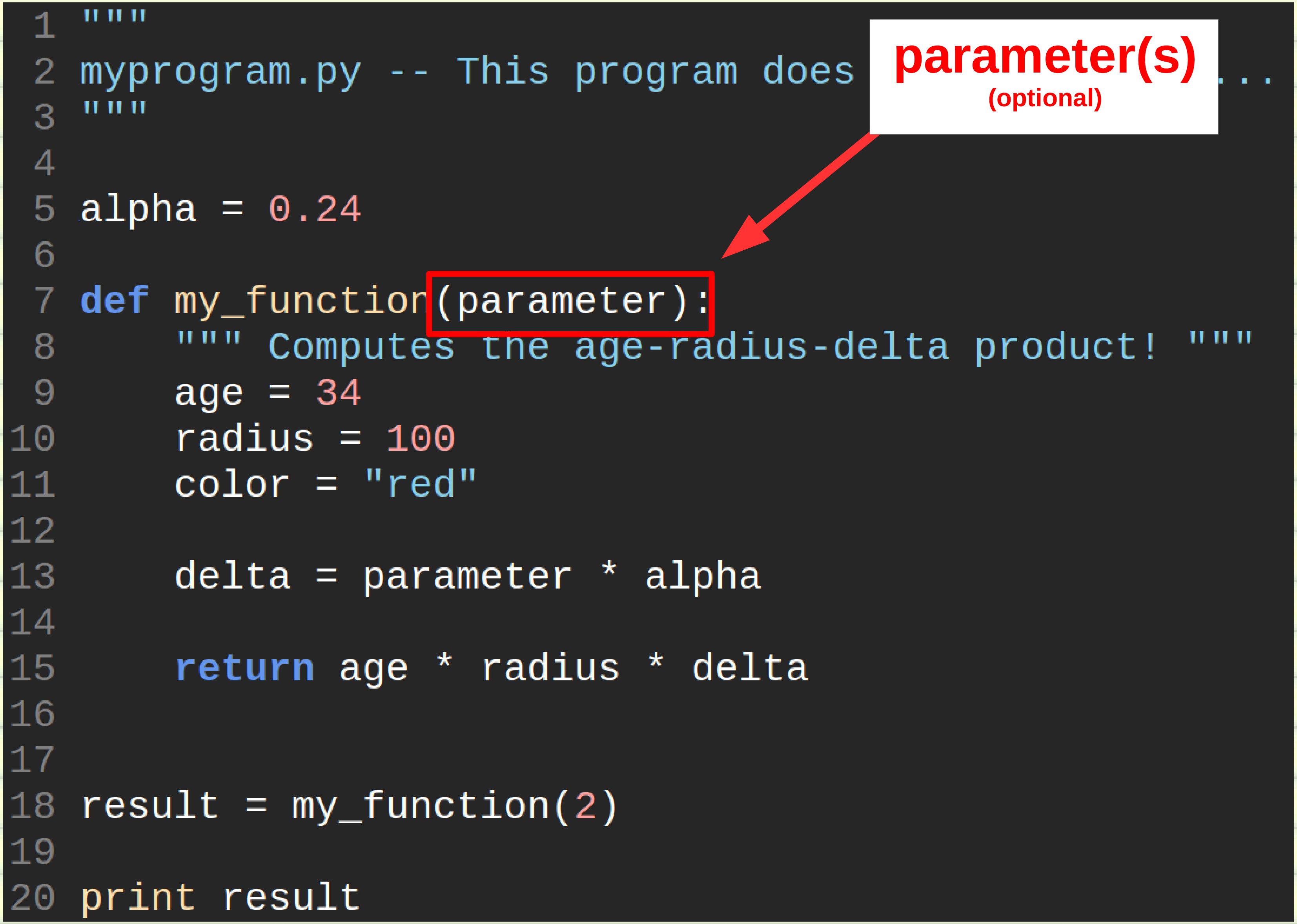
Anatomy of a Python function



```
"""
myprogram.py -- This program does blah blah blah...
"""

alpha = 0.24

def my_function(parameter):
    """ Computes the age-radius-delta product! """
    age = 34
    radius = 100
    color = "red"

    delta = parameter * alpha

    return age * radius * delta


result = my_function(2)

print result
```

function name

Anatomy of a Python function

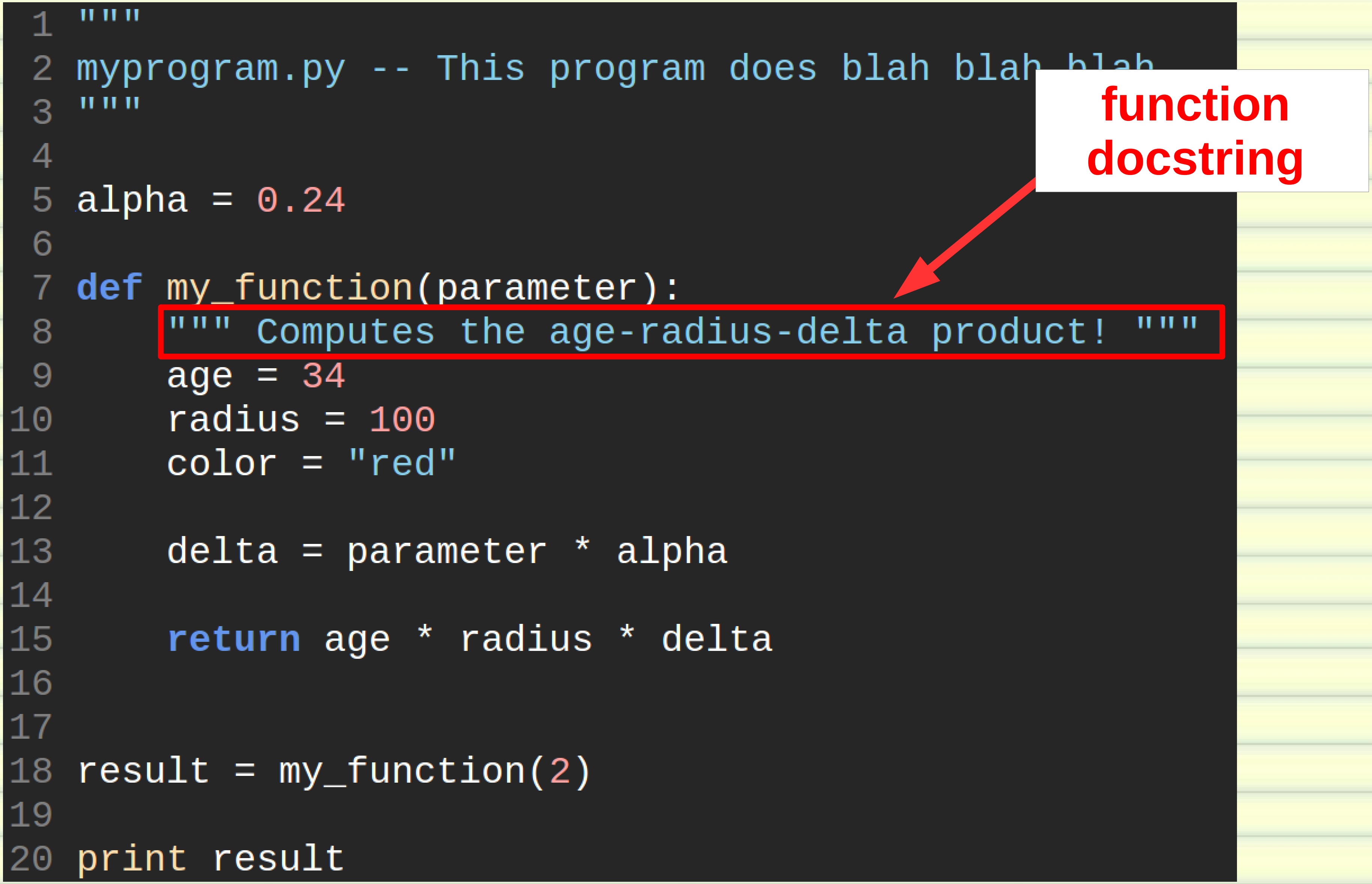


```
"""
myprogram.py -- This program does ...
"""

alpha = 0.24

def my_function(parameter):
    """ Computes the age-radius-delta product! """
    age = 34
    radius = 100
    color = "red"

    delta = parameter * alpha

    return age * radius * delta


result = my_function(2)

print result
```

parameter(s)

(optional)

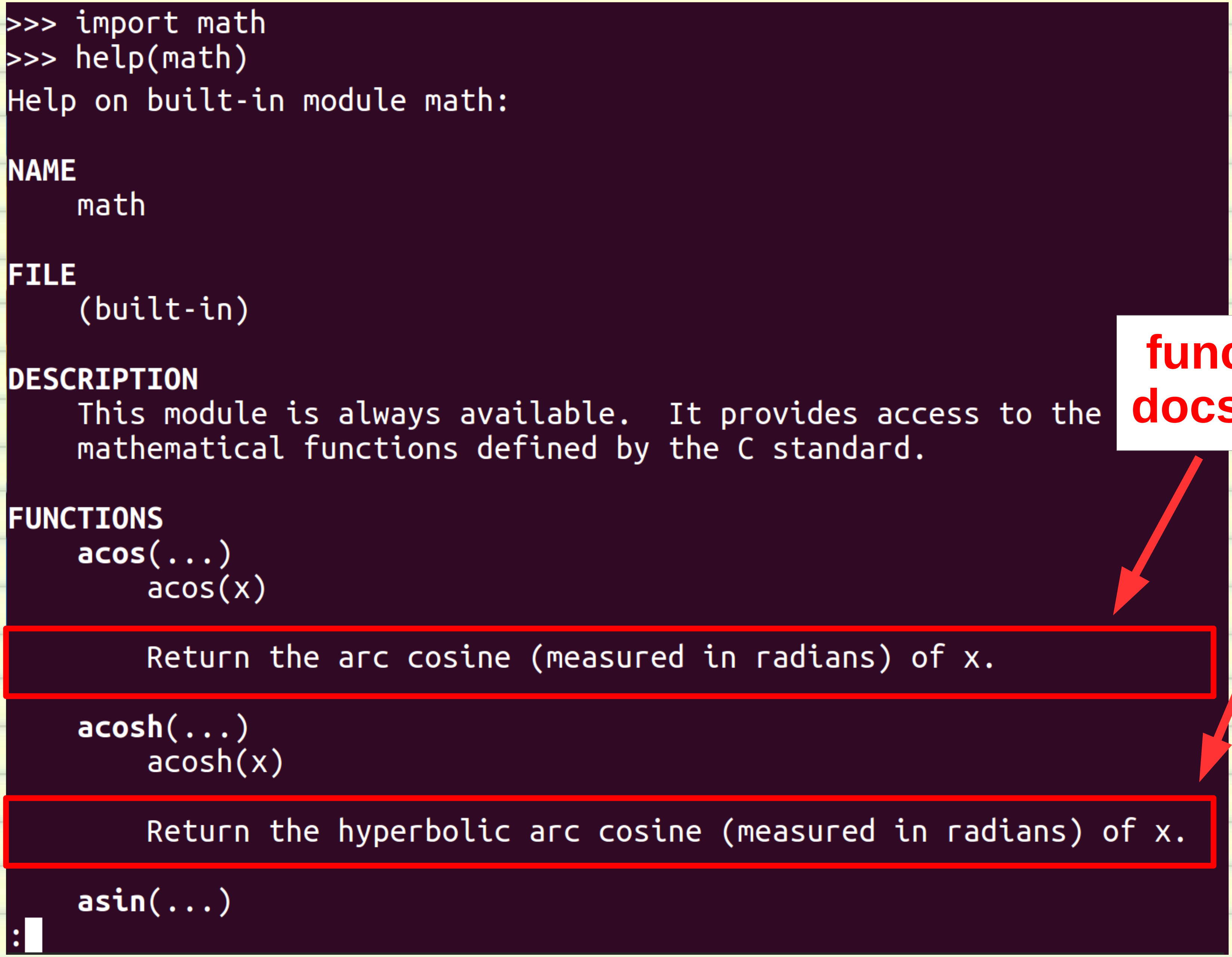
Anatomy of a Python function



```
"""
myprogram.py -- This program does blah blah blah
"""

alpha = 0.24

def my_function(parameter):
    """ Computes the age-radius-delta product! """
    age = 34
    radius = 100
    color = "red"

    delta = parameter * alpha

    return age * radius * delta


result = my_function(2)

print result
```

function docstring

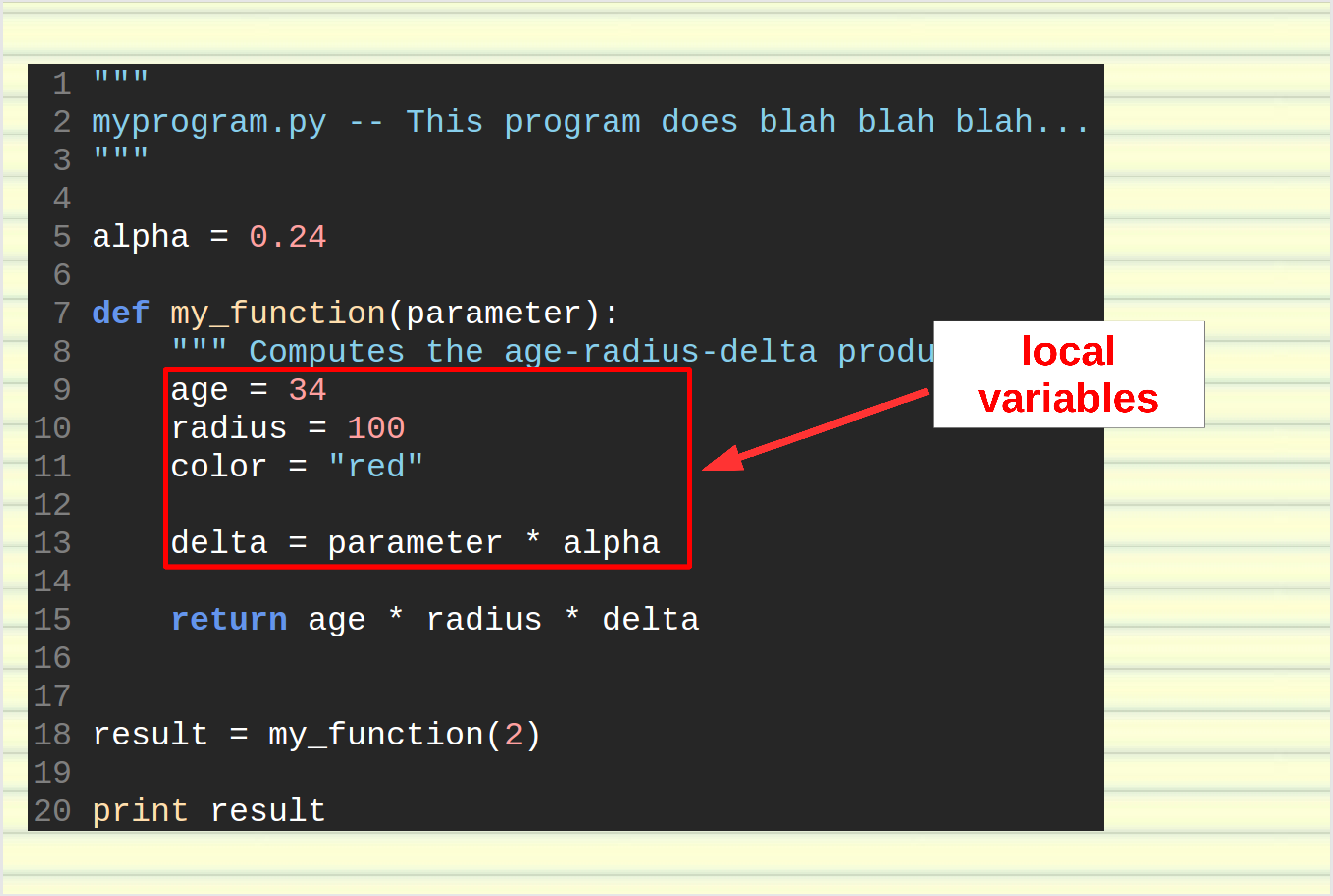
Anatomy of an (almost) “proper” Python program



```
>>> import math
>>> help(math)
Help on built-in module math:

NAME
    math

FILE
    (built-in)

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)

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

    asin(...)
:
```

function docstring

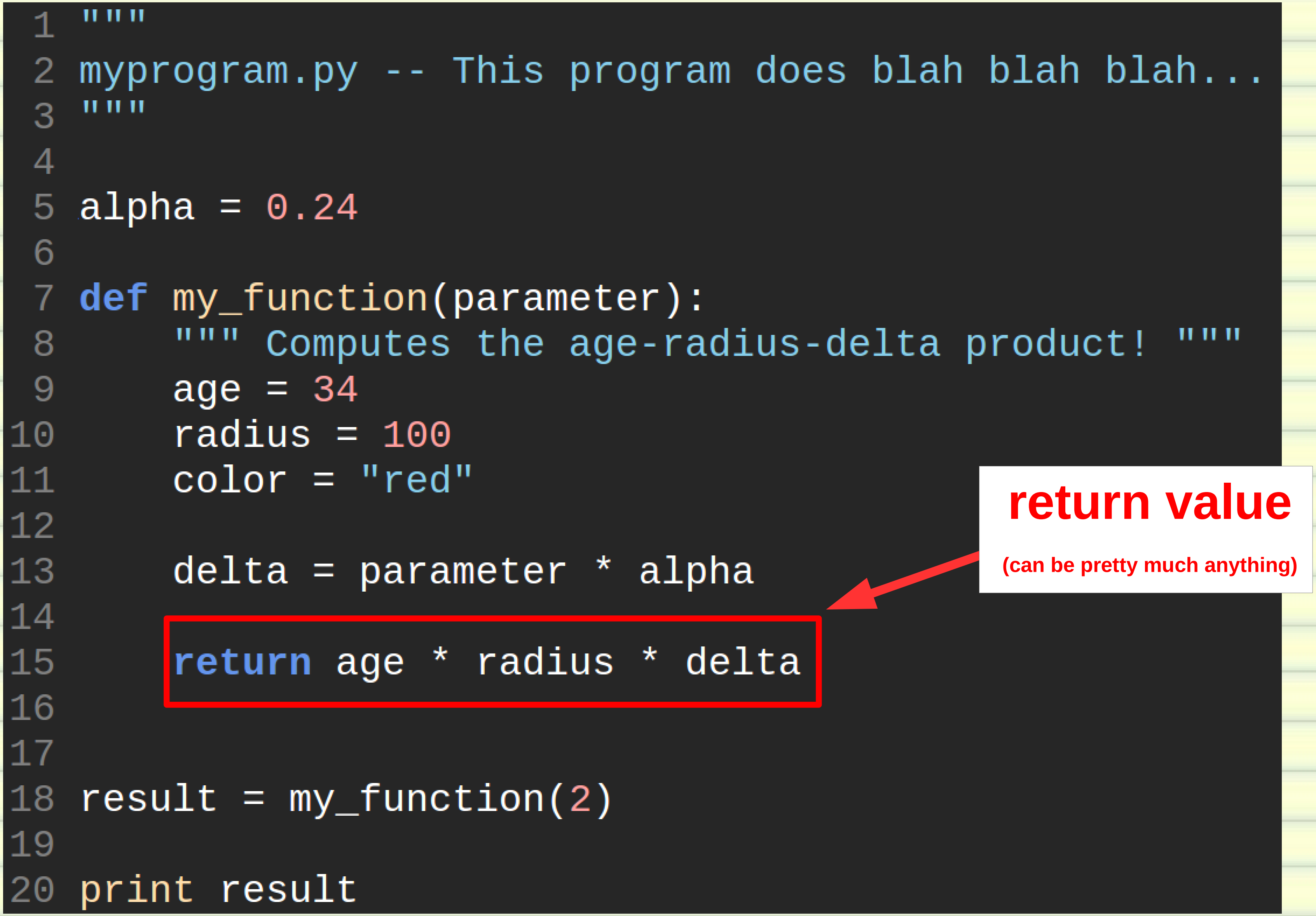
Anatomy of a Python function



```
"""
myprogram.py -- This program does blah blah blah...
"""

alpha = 0.24

def my_function(parameter):
    """ Computes the age-radius-delta produ
    age = 34
    radius = 100
    color = "red"

    delta = parameter * alpha

    return age * radius * delta


result = my_function(2)

print result
```

local variables

Anatomy of a Python function



```
"""
myprogram.py -- This program does blah blah blah...
"""

alpha = 0.24

def my_function(parameter):
    """ Computes the age-radius-delta product! """
    age = 34
    radius = 100
    color = "red"

    delta = parameter * alpha

    return age * radius * delta


result = my_function(2)

print result
```

return value

(can be pretty much anything)

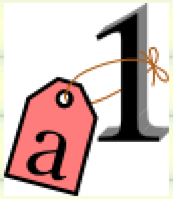
A Few Notes on Representations

In the previous lecture, we learned about *objects* and how to <u>bind</u> *names* to objects

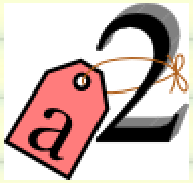
>> Name binding is kinda unique to Python
>> You can later define your own custom objects (later...)

<u>Python</u>

```
a = 1
```

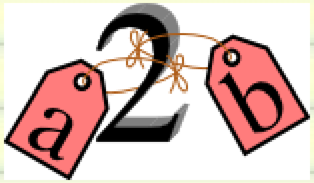


```
a = 2
```



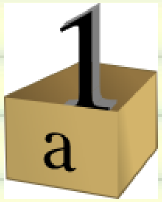
Gets "garbage collected" (no bindings)

```
b = a
```

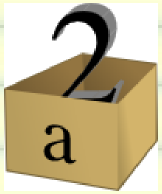


<u>Most Other Languages</u>

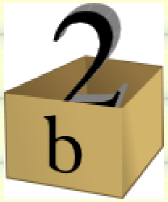
```
int a = 1;
```

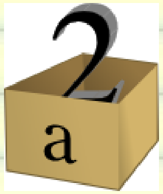


```
int a = 2;
```



```
int b = a;
```





Fundamental Datatypes

Mutable (adj.) – State ***can*** be changed after creation.

Immutable (adj.) – State **cannot** be changed after creation.

Mutable Python Types

- **list**

  Similar to a vector in MATLAB, but not confined to just numbers. Can also be heterogeneous!

  example:

  ```
  >>> A = [3.24, 78, 'foo', 1103]
  >>> A[1:3]
  [78, 'foo']
  ```

- **dictionary**

  An associative array.

  example:

  ```
  >>> A = {'age': 34, 'gender': 'female'}
  >>> A['gender']
  'female'
  ```

Immutable Python Types

- **int, float, long, complex**

- **tuple**

  Similar to a **list**, but values cannot be changed after creation. Consequently, a bit faster.

  example:

  ```
  >>> A = (32, 'bar', 32.22)
  >>> A[0:2]
  (32, 'bar')
  ```

- **str**

  A string of characters

  example:

  ```
  >>> A = “Hello World!”
  >>> A[3:9]
  'lo Wor'
  ```

Sequence Attributes of Strings

A Bit More on Strings

Strings are one of the three *Sequence Types* in Python

(the other two are tuples and lists)

★ **Strings are immutable – you can't change strings**

(but you can create new strings from other strings)

Sequence Attributes of Strings

A Bit More on Strings

Strings are one of the three *Sequence Types* in Python

(the other two are tuples and lists)

- ★ **Strings are <u>immutable</u> – you can't change strings**

  (but you can create new strings from other strings)

- ★ **Strings are defined using quotes – (“, ', or “““)**

```
>> my_string = “Hello World”      # This and
>> my_string = 'Hello World'      #       ...this are the same

>> my_string = “““This is a multi-line
string that uses triple quotes”””
```

Sequence Attributes of Strings

A Bit More on Strings

Strings are one of the three *Sequence Types* in Python

(the other two are tuples and lists)

★ **Strings are immutable – you can't change strings**

(but you can create new strings from other strings)

★ **Strings are defined using quotes – (“, ', or “““)**

```
>> my_string = “Hello World”      # This and
>> my_string = 'Hello World'      #       ...this are the same

>> my_string = ““““This is a multi-line
string that uses triple quotes”””
```

★ **The different quotes allow you to actually use quotes in your strings!**

```
>> my_string = 'Shackleford said, “Learn Python”'

>> print my_string
Shackleford said, “Learn Python”
```

Sequence Attributes of Strings

A Bit More on Strings

Strings are one of the three *Sequence Types* in Python

(the other two are tuples and lists)

★ All Sequence Types can be indexed – this includes Strings

```
>> my_string = “Hello World”
>> first_character = my_string[0]
>> print first_character
H

>> print my_string[3]
l
```

(you played with this a bit in lab)

Sequence Attributes of Strings

A Bit More on Strings

Strings are one of the three *Sequence Types* in Python

(the other two are tuples and lists)

★ All Sequence Types can be indexed – this includes Strings

```
>> my_string = “Hello World”
>> first_character = my_string[0]
>> print first_character
H

>> print my_string[3]
l
```

(you played with this a bit in lab)

★ Negative indices “wrap around” to the end and “go backwards”

```
>> my_string = “Hello World”
>> last_character = my_string[-1]
>> print last_character
d

>> print my_string[-3]
r
```

Sequence Attributes of Strings

A Bit More on Strings

Strings are one of the three *Sequence Types* in Python

(the other two are tuples and lists)

★ All Sequence Types can be sliced – this includes Strings

This creates a new string object containing a subset of a string

```
>> my_string = “Hello World”

>> my_string[1:4]
'ell'
```

Sequence Attributes of Strings

A Bit More on Strings

Strings are one of the three *Sequence Types* in Python

(the other two are tuples and lists)

★ All Sequence Types can be sliced – this includes Strings

This creates a new string object containing a subset of a string

```
>> my_string = “Hello World”

>> my_string[1:4]
'ell'

>> my_string[1:-1]
'ello Worl'
```

Sequence Attributes of Strings

A Bit More on Strings

Strings are one of the three *Sequence Types* in Python

(the other two are tuples and lists)

★ All Sequence Types can be sliced – this includes Strings

This creates a new string object containing a subset of a string

```
>> my_string = “Hello World”

>> my_string[1:4]
'ell'

>> my_string[1:-1]
'ello Worl'

>> my_string[:5]
'Hello'
```

Sequence Attributes of Strings

A Bit More on Strings

Strings are one of the three *Sequence Types* in Python

(the other two are tuples and lists)

★ All Sequence Types can be sliced – this includes Strings

This creates a new string object containing a subset of a string

```
>> my_string = “Hello World”

>> my_string[1:4]
'ell'

>> my_string[1:-1]
'ello Worl'

>> my_string[:5]
'Hello'

>> my_string[6:]
'World'
```

Sequence Attributes of Strings

A Bit More on Strings

Strings are one of the three *Sequence Types* in Python

(the other two are tuples and lists)

★ **All Sequence Types can be sliced – this includes Strings**

This creates a new string object containing a subset of a string

```
>> my_string = “Hello World”

>> my_string[1:4]
'ell'

>> my_string[1:-1]
'ello Worl'

>> my_string[:5]
'Hello'

>> my_string[6:]
'World'

>> my_string[:]
'Hello World'
```

More Attributes of Strings

A Bit More on Strings

Strings are one of the three *Sequence Types* in Python

(the other two are tuples and lists)

★ Subsequence Matching using the `in` operator

```
>>> my_string = “Hello World”

>>> 'Wor' in my_string
True
```

More Attributes of Strings

A Bit More on Strings

Strings are one of the three *Sequence Types* in Python

(the other two are tuples and lists)

★ Subsequence Matching using the `in` operator

```
>>> my_string = “Hello World”

>>> 'Wor' in my_string
True

>>> 'Wr' in my_string
False
```

More Attributes of Strings

A Bit More on Strings

Strings are one of the three *Sequence Types* in Python

(the other two are tuples and lists)

★ Subsequence Matching using the `in` operator

```
>>> my_string = “Hello World”

>>> 'Wor' in my_string
True

>>> 'Wr' in my_string
False

>>> if 'Hello' in my_string:
...     print 'Great Success!'
...
Great Success!
```

More Attributes of Strings

A Bit More on Strings

Strings are one of the three *Sequence Types* in Python

(the other two are tuples and lists)

★ **Strings can be concatenated**

```
>>> var1 = “Hello” + “ ” + “World”

>>> print var1
Hello World!
```

More Attributes of Strings

A Bit More on Strings

Like everything in Python, Strings are *objects*

THIS MEANS THEY HAVE METHODS

★ **Strings have lots of powerful methods!**

```
>>> foo = “i am a string!”
>>> foo.upper()
'I AM A STRING!'
```

https://docs.python.org/2/library/stdtypes.html#string-methods

More Attributes of Strings

A Bit More on Strings

Like everything in Python, Strings are *objects*

THIS MEANS THEY HAVE METHODS

★ **Strings have lots of powerful methods!**

```
>>> foo = “i am a string!”
>>> foo.upper()
'I AM A STRING!'

>> bar = “this is also valid!”.upper()
>> print bar
THIS IS ALSO VALID!
```

https://docs.python.org/2/library/stdtypes.html#string-methods

More Attributes of Strings

A Bit More on Strings

Like everything in Python, Strings are *objects*

THIS MEANS THEY HAVE METHODS

★ **Strings have lots of powerful methods!**

```
>>> foo = “i am a string!”
>>> foo.upper()
'I AM A STRING!'

>> bar = “this is also valid!”.upper()
>> print bar
THIS IS ALSO VALID!

>>> 'chapter 3: python makes programming fun'.title()
'Chapter 3: Python Makes Programming Fun'
```

https://docs.python.org/2/library/stdtypes.html#string-methods

More Attributes of Strings

A Bit More on Strings

Like everything in Python, Strings are *objects*

THIS MEANS THEY HAVE METHODS

★ **Strings have lots of powerful methods!**

```
>>> foo = “i am a string!”
>>> foo.upper()
'I AM A STRING!'

>> bar = “this is also valid!”.upper()
>> print bar
THIS IS ALSO VALID!

>>> 'chapter 3: python makes programming fun'.title()
'Chapter 3: Python Makes Programming Fun'

>>> “data1,label,data2,foo,bar”.split(“,”)
['data1', 'label', 'data2', 'foo', 'bar']
```

https://docs.python.org/2/library/stdtypes.html#string-methods

More Attributes of Strings

A Bit More on Strings

Like everything in Python, Strings are *objects*

THIS MEANS THEY HAVE METHODS

★ **Strings have lots of powerful methods!**

```
>>> foo = “i am a string!”
>>> foo.upper()
'I AM A STRING!'

>> bar = “this is also valid!”.upper()
>> print bar
THIS IS ALSO VALID!

>>> 'chapter 3: python makes programming fun'.title()
'Chapter 3: Python Makes Programming Fun'

>>> “data1,label,data2,foo,bar”.split(“,”)
['data1', 'label', 'data2', 'foo', 'bar']

>>> “-”.join( [“join”, “a”, “list”, “of”, “strings”] )
'join-a-list-of-strings'
```

https://docs.python.org/2/library/stdtypes.html#string-methods

More Attributes of Strings

A Bit More on Strings

Finally, Strings have the Formatting Operator %

Used to build a string by “filling in the blanks” with a single value, tuple, or dictionary

```
>>> speed = 10
>>> fuel = 5.23
>>> color = “blue”

>>> foo = “speed = %i” % speed
>>> print foo
'speed = 10'
```

For a table of conversion types (i.e. `%i`, `%f`, `%s`, etc) visit:

https://docs.python.org/2/library/stdtypes.html#string-formatting-operations

More Attributes of Strings

A Bit More on Strings

Finally, Strings have the Formatting Operator %

Used to build a string by “filling in the blanks” with a single value, tuple, or dictionary

```
>>> speed = 10
>>> fuel = 5.23
>>> color = “blue”

>>> foo = “speed = %i” % speed
>>> print foo
'speed = 10'

>> bar = “speed = %i, fuel = %f, and color is %s” % (speed, fuel, color)
>> print bar
'speed = 10, fuel = 5.230000, and color is blue'
```

For a table of conversion types (i.e. `%i`, `%f`, `%s`, etc) visit:

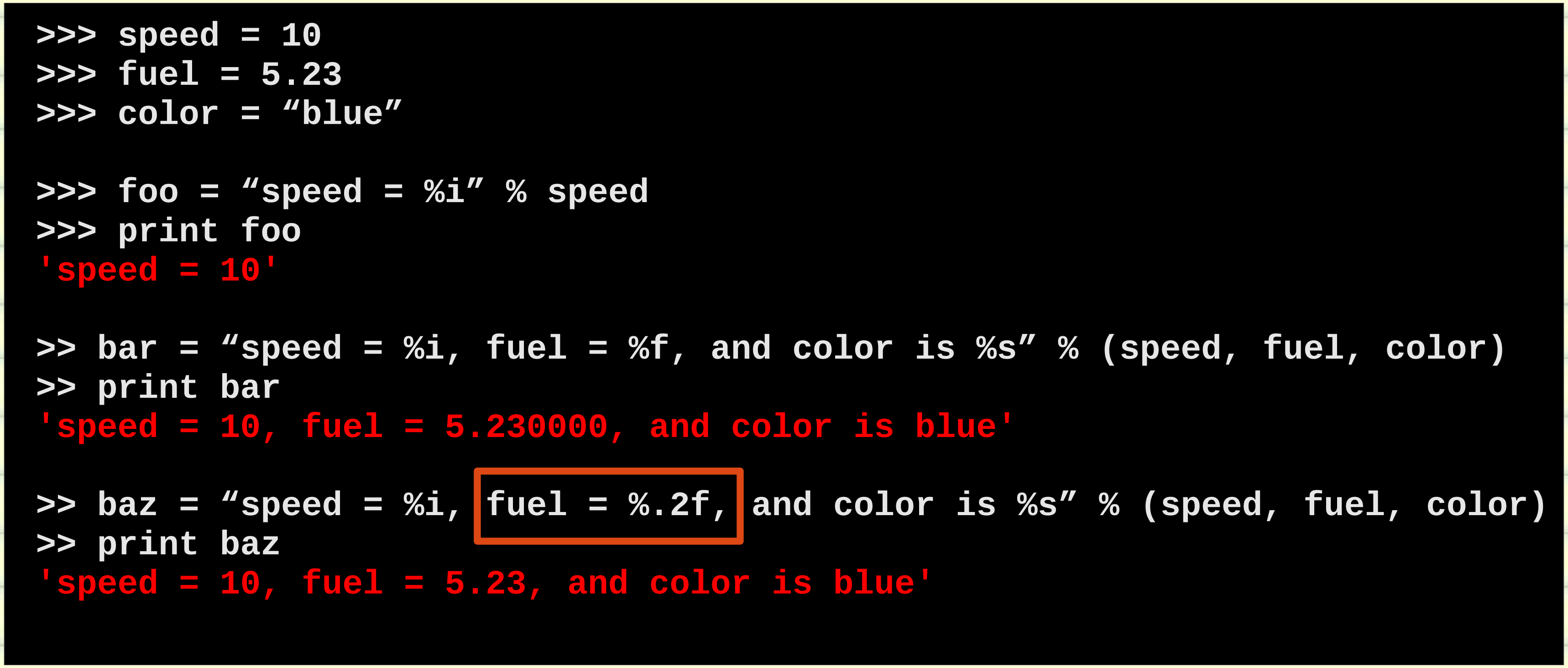
https://docs.python.org/2/library/stdtypes.html#string-formatting-operations

More Attributes of Strings

A Bit More on Strings

Finally, Strings have the Formatting Operator %

Used to build a string by "filling in the blanks" with a single value, tuple, or dictionary



```
>>> speed = 10
>>> fuel = 5.23
>>> color = “blue”

>>> foo = “speed = %i” % speed
>>> print foo
'speed = 10'

>> bar = “speed = %i, fuel = %f, and color is %s” % (speed, fuel, color)
>> print bar
'speed = 10, fuel = 5.230000, and color is blue'

>> baz = “speed = %i, fuel = %.2f, and color is %s” % (speed, fuel, color)
>> print baz
'speed = 10, fuel = 5.23, and color is blue'
```

For a table of conversion types (i.e. %i, %f, %s, etc) visit:

https://docs.python.org/2/library/stdtypes.html#string-formatting-operations

Sequence Attributes of Lists

Introduction to Lists

Lists are one of the three *Sequence Types* in Python

(the other two are tuples and strings)

If you "get" strings, lists are very similar

★ **Lists are defined using square brackets [ ] – items are comma separated**

```
>> my_list = [43, 23, 10, 5, 91]
>> print my_list
[43, 23, 10, 5, 91]
```

(you played with this a bit in lab)

Sequence Attributes of Lists

Introduction to Lists

Lists are one of the three *Sequence Types* in Python

(the other two are tuples and strings)

If you "get" strings, lists are very similar

★ **Lists are defined using square brackets [ ] – items are comma separated**

```
>> my_list = [43, 23, 10, 5, 91]
>> print my_list
[43, 23, 10, 5, 91]
```

(you played with this a bit in lab)

★ **A single list can contain many different types of items**

```
>> my_list = [84, “some words”, 1.234, 600, 'test']
>> print my_list
[84, 'some words', 1.234, 600, 'test']
```

Sequence Attributes of Lists

Introduction to Lists

Lists are one of the three *Sequence Types* in Python

(the other two are tuples and strings)

If you "get" strings, lists are very similar

★ All Sequence Types can be indexed – this includes Lists

```
>> my_list = [43, 23, 10, 5, 91]
>> first_item = my_list[0]
>> print first_item
43

>> print my_string[3]
5
```

(you played with this a bit in lab)

Sequence Attributes of Lists

Introduction to Lists

Lists are one of the three *Sequence Types* in Python

(the other two are tuples and strings)

If you "get" strings, lists are very similar

★ All Sequence Types can be indexed – this includes Lists

```
>> my_list = [43, 23, 10, 5, 91]
>> first_item = my_list[0]
>> print first_item
43

>> print my_string[3]
5
```

(you played with this a bit in lab)

★ Negative indices "wrap around" to the end and "go backwards"

```
>> my_list = [43, 23, 10, 5, 91]
>> last_item = my_list[-1]
>> print last_item
91

>> print my_list[-3]
10
```

List Methods

Introduction to Lists

Lists are one of the three *Sequence Types* in Python

(the other two are tuples and strings)

If you "get" strings, lists are very similar

★ **A new empty string can be defined easily**

```
>> my_list = []
```

List Methods

Introduction to Lists

Lists are one of the three *Sequence Types* in Python

(the other two are tuples and strings)

If you "get" strings, lists are very similar

★ A new empty string can be defined easily

```
>> my_list = []
```

★ Adding new items to a list at runtime is also easy!

```
>> my_list = []
>> my_list.append(40)
>> my_list.append('foo')
>> my_list.append(32.234)
>> print my_list
[40, 'foo', 32.234]

>> my_list.append('more words')
>> print my_list
[40, 'foo', 32.234, 'more words']
```

List Methods

Introduction to Lists

Lists are one of the three *Sequence Types* in Python

(the other two are tuples and strings)

If you "get" strings, lists are very similar

★ **Removing arbitrary items from a list is simple**

```
>> my_list = [98, 23, 'time of day', 32.99]
>> my_list.remove(23)
>> print my_list
[98, 'time of day', 32.99]

>> my_list.remove('time of day')
>> print my_list
[09, 32.99]
```

List Methods

Introduction to Lists

Lists are one of the three *Sequence Types* in Python

(the other two are tuples and strings)

If you "get" strings, lists are very similar

★ Removing arbitrary items from a list is simple

```
>> my_list = [98, 23, 'time of day', 32.99]
>> my_list.remove(23)
>> print my_list
[98, 'time of day', 32.99]

>> my_list.remove('time of day')
>> print my_list
[09, 32.99]
```

★ Lists are also sortable and reversible "in place"

```
>> my_list = [23, 40, 100, 21, 1, 59]
>> my_list.sort()
>> print my_list
[1, 21, 23, 40, 59, 100]

>> my_list.reverse()
>> print my_list
[100, 59, 40, 23, 21, 1]
```

A Very Brief Look at Tuples

Defined Upon Creation (called “Packing”)

```
>>> test = (8, 23, 99, 4, 61)
>>> print test
(8, 23, 99, 4, 61)
```

A Very Brief Look at Tuples

Defined Upon Creation (called “Packing”)

```
>>> test = (8, 23, 99, 4, 61)
>>> print test
(8, 23, 99, 4, 61)
```

Parenthesis Optional (but customary)

```
>>> test = 8, 23, 99, 4, 61
>>> print test
(8, 23, 99, 4, 61)
```

A Very Brief Look at Tuples

Defined Upon Creation (called “Packing”)

```
>>> test = (8, 23, 99, 4, 61)
>>> print test
(8, 23, 99, 4, 61)
```

Parenthesis Optional (but customary)

```
>>> test = 8, 23, 99, 4, 61
>>> print test
(8, 23, 99, 4, 61)
```

Indexable

```
>>> test = (8, 23, 99, 4, 61)
>>> print test[2]
99
```

A Very Brief Look at Tuples

Immutable – Cannot Change!

```
>>> test = (8, 23, 99, 4, 61)
>>> test[2] = 327
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

A Very Brief Look at Tuples

Immutable – Cannot Change!

```
>>> test = (8, 23, 99, 4, 61)
>>> test[2] = 327
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

Immutable – Cannot Append!

```
>>> test = (8, 23, 99, 4, 61)
>>> test.append(690)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'tuple' object has no attribute 'append'
```

A Very Brief Look at Tuples

Tuples can be **Unpacked** as easily as they are **Packed**

```
>>> employee = ('bob', 'male', 42, 'engineer')
```

A Very Brief Look at Tuples

Tuples can be **Unpacked** as easily as they are **Packed**

```
>>> employee = ('bob', 'male', 42, 'engineer')
>>> name, sex, age, job = employee
```

unpacking

A Very Brief Look at Tuples

Tuples can be **Unpacked** as easily as they are **Packed**

```
>>> employee = ('bob', 'male', 42, 'engineer')
>>> name, sex, age, job = employee
>>> print name, sex, age, job
bob male 42 engineer
```

unpacking

A Very Brief Look at Tuples

Tuples can be **Unpacked** as easily as they are **Packed**

```
>>> employee = ('bob', 'male', 42, 'engineer')
>>> name, sex, age, job = employee
>>> print name, sex, age, job
bob male 42 engineer
```

unpacking

Non-”Pythonic” Variable Swap

```
>>> a = 5
>>> b = 9

>>> tmp = a
>>> a = b
>>> b = tmp
```

A Very Brief Look at Tuples

Tuples can be **Unpacked** as easily as they are **Packed**

```
>>> employee = ('bob', 'male', 42, 'engineer')
>>> name, sex, age, job = employee
>>> print name, sex, age, job
bob male 42 engineer
```

unpacking

Non-”Pythonic” Variable Swap

```
>>> a = 5
>>> b = 9

>>> tmp = a
>>> a = b
>>> b = tmp
```

”Pythonic” Variable Swap

```
>>> a = 5
>>> b = 9

>>> b, a = a, b
```

A Very Brief Look at Tuples

Tuples can be **Unpacked** as easily as they are **Packed**

```
>>> employee = ('bob', 'male', 42, 'engineer')
>>> name, sex, age, job = employee
>>> print name, sex, age, job
bob male 42 engineer
```

Non-”Pythonic” Variable Swap

```
>>> a = 5
>>> b = 9

>>> tmp = a
>>> a = b
>>> b = tmp
```

”Pythonic” Variable Swap

```
>>> a = 5
>>> b = 9

>>> b, a = a, b
```

packing

(creates an “anonymous” tuple object)

A Very Brief Look at Tuples

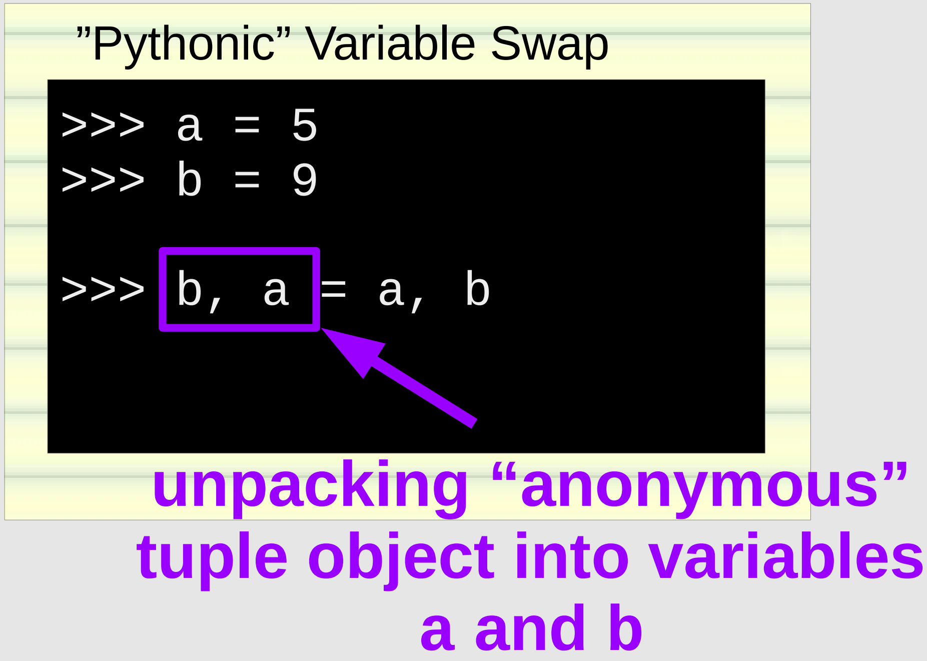
Tuples can be **Unpacked** as easily as they are **Packed**

```
>>> employee = ('bob', 'male', 42, 'engineer')
>>> name, sex, age, job = employee
>>> print name, sex, age, job
bob male 42 engineer
```

Non-”Pythonic” Variable Swap

```
>>> a = 5
>>> b = 9

>>> tmp = a
>>> a = b
>>> b = tmp
```

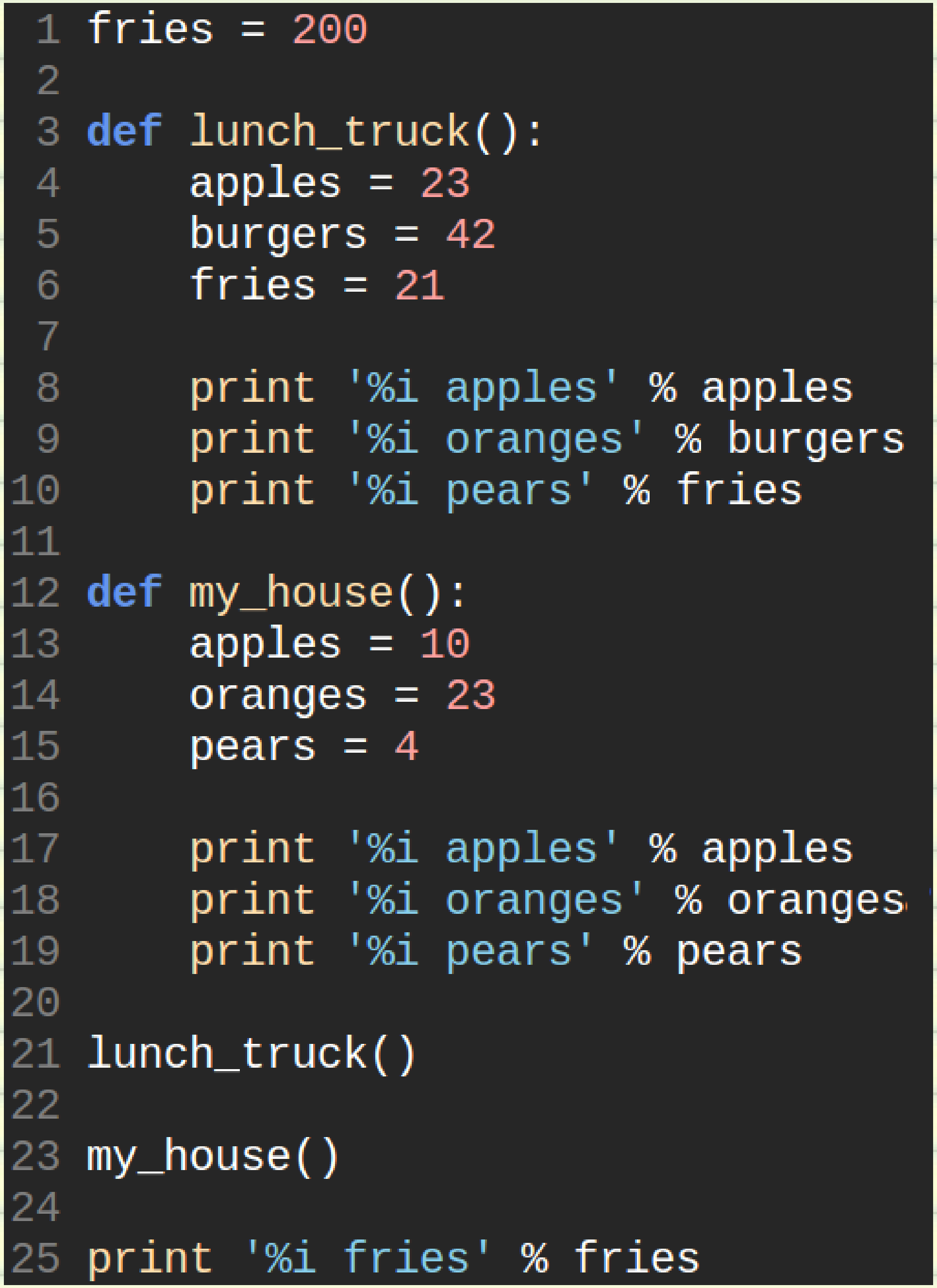
”Pythonic” Variable Swap



```
>>> a = 5
>>> b = 9

>>> b, a = a, b
```

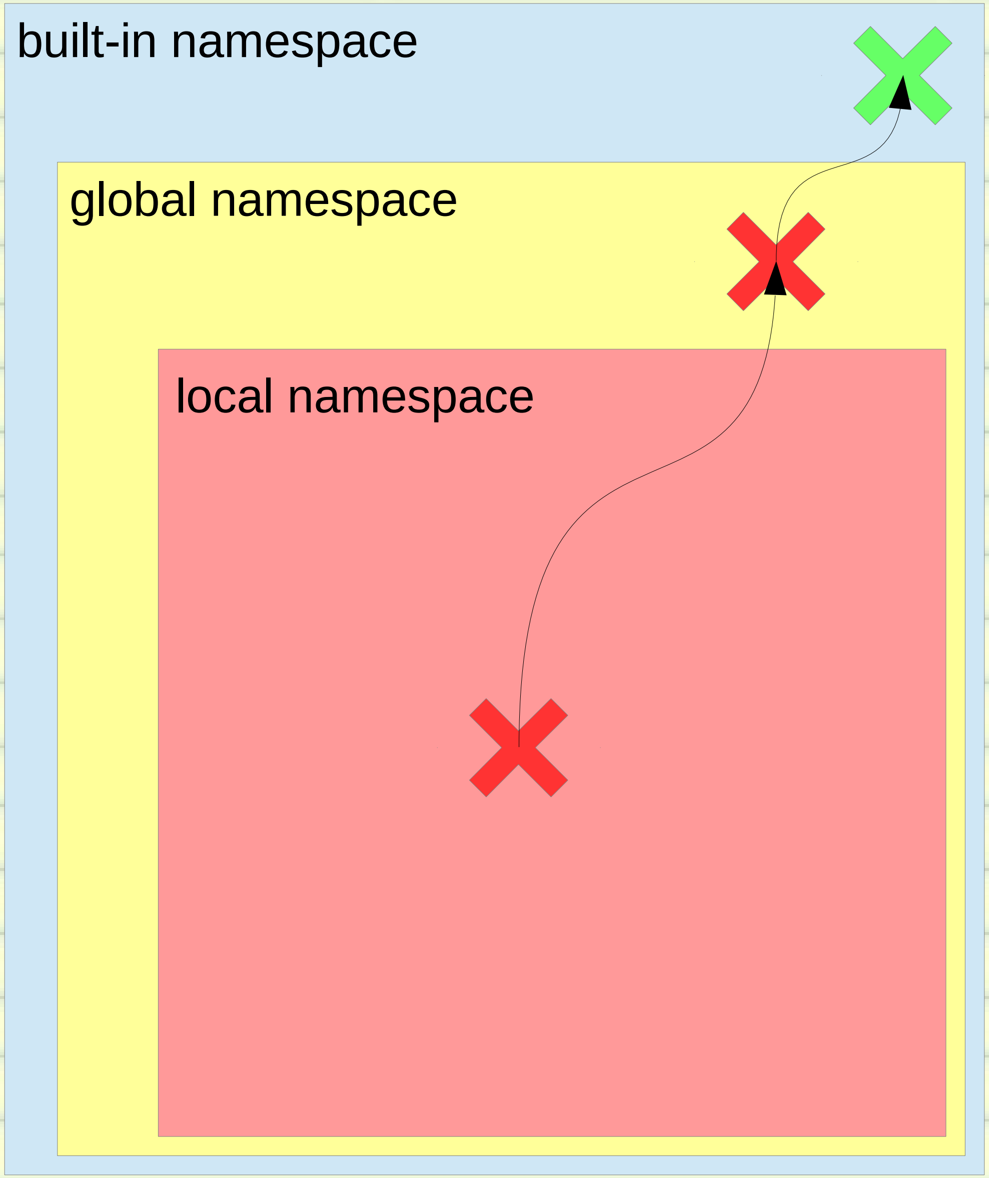
unpacking “anonymous” tuple object into variables a and b

Namespaces & Variable Scope



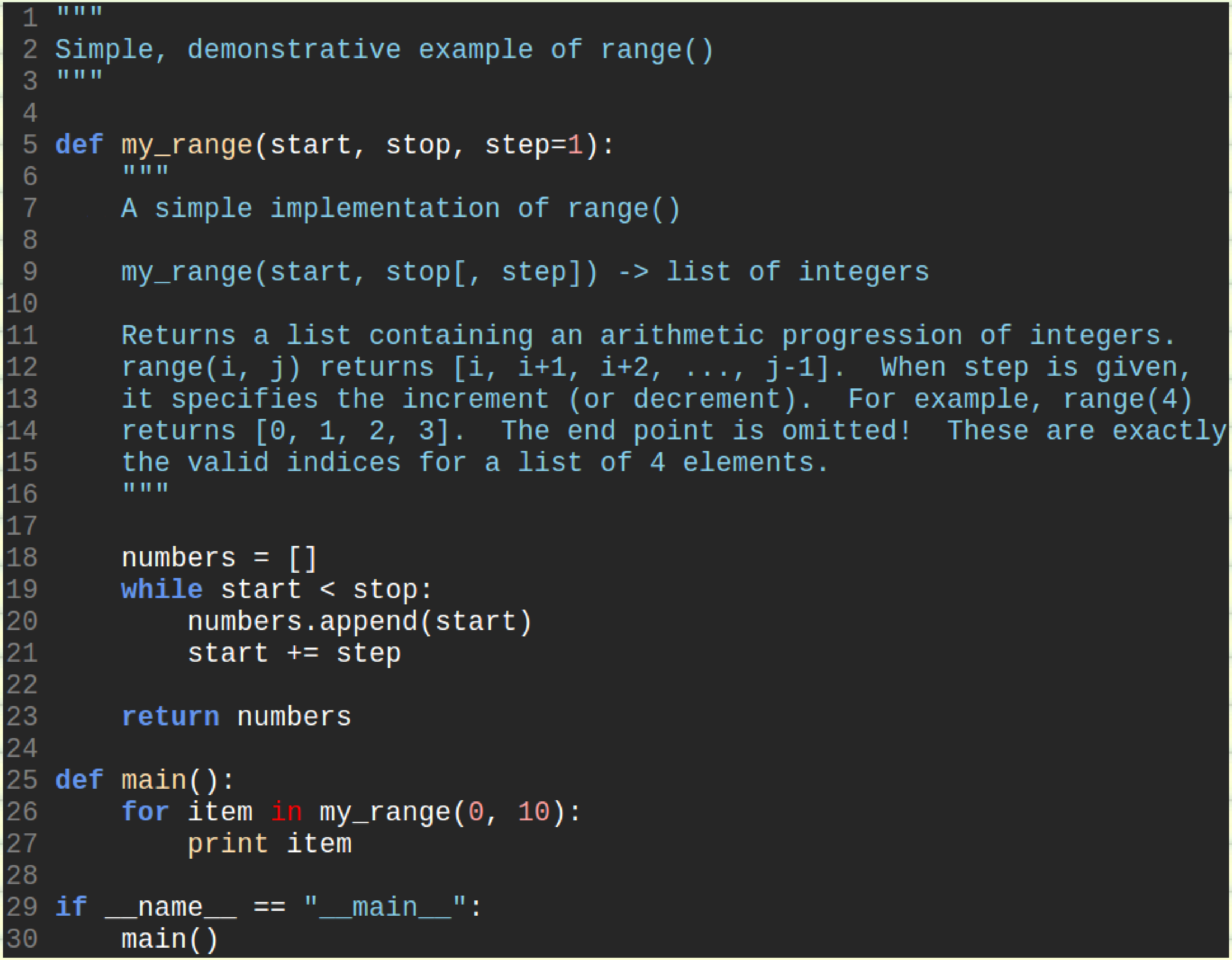
```python
fries = 200

def lunch_truck():
    apples = 23
    burgers = 42
    fries = 21

    print '%i apples' % apples
    print '%i oranges' % burgers
    print '%i pears' % fries

def my_house():
    apples = 10
    oranges = 23
    pears = 4

    print '%i apples' % apples
    print '%i oranges' % oranges
    print '%i pears' % pears

lunch_truck()

my_house()

print '%i fries' % fries
```

Name search looks like this:



Using Functions -- import

example.py



```python
"""
Simple, demonstrative example of range()
"""

def my_range(start, stop, step=1):
    """
    A simple implementation of range()

    my_range(start, stop[, step]) -> list of integers

    Returns a list containing an arithmetic progression of integers.
    range(i, j) returns [i, i+1, i+2, ..., j-1].  When step is given,
    it specifies the increment (or decrement).  For example, range(4)
    returns [0, 1, 2, 3].  The end point is omitted!  These are exactly
    the valid indices for a list of 4 elements.
    """

    numbers = []
    while start < stop:
        numbers.append(start)
        start += step

    return numbers

def main():
    for item in my_range(0, 10):
        print item

if __name__ == "__main__":
    main()
```

New Stuff!!!

NEW STUFF!!!

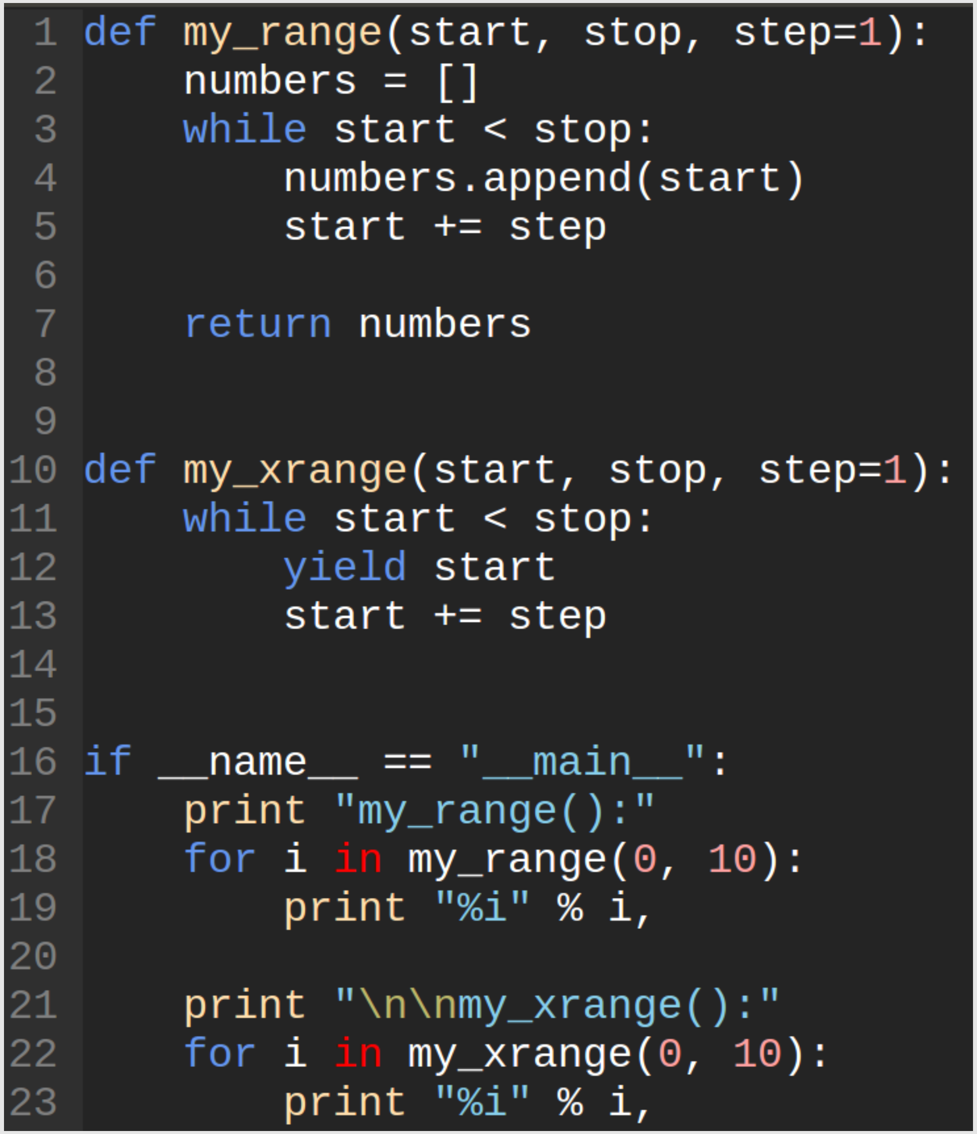
New Stuff - Generators

GENERATOR FUNCTIONS

Compute On-Demand

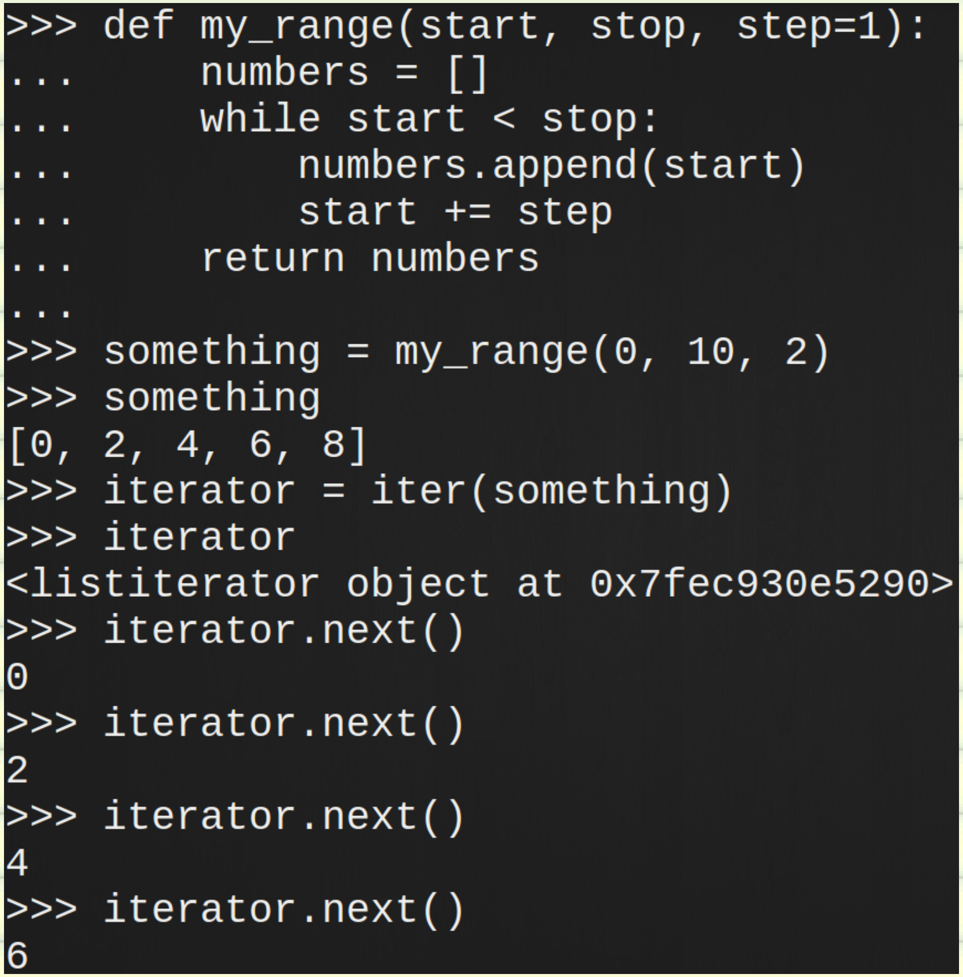
Reduce Memory Usage
Increase Speed

New Stuff - Generators



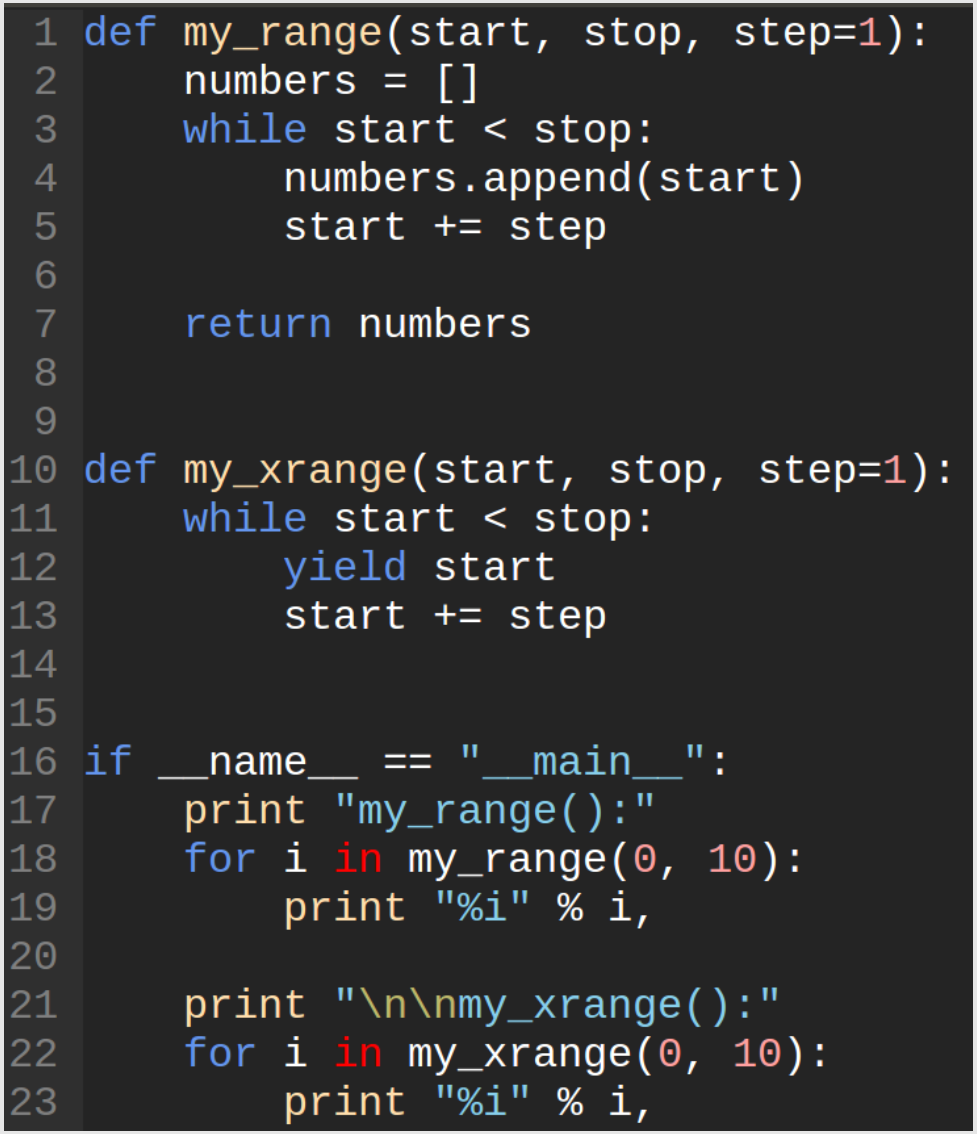
```python
def my_range(start, stop, step=1):
    numbers = []
    while start < stop:
        numbers.append(start)
        start += step

    return numbers


def my_xrange(start, stop, step=1):
    while start < stop:
        yield start
        start += step


if __name__ == "__main__":
    print "my_range():"
    for i in my_range(0, 10):
        print "%i" % i,

    print "\n\nmy_xrange():"
    for i in my_xrange(0, 10):
        print "%i" % i,
```

Let's Experiment:



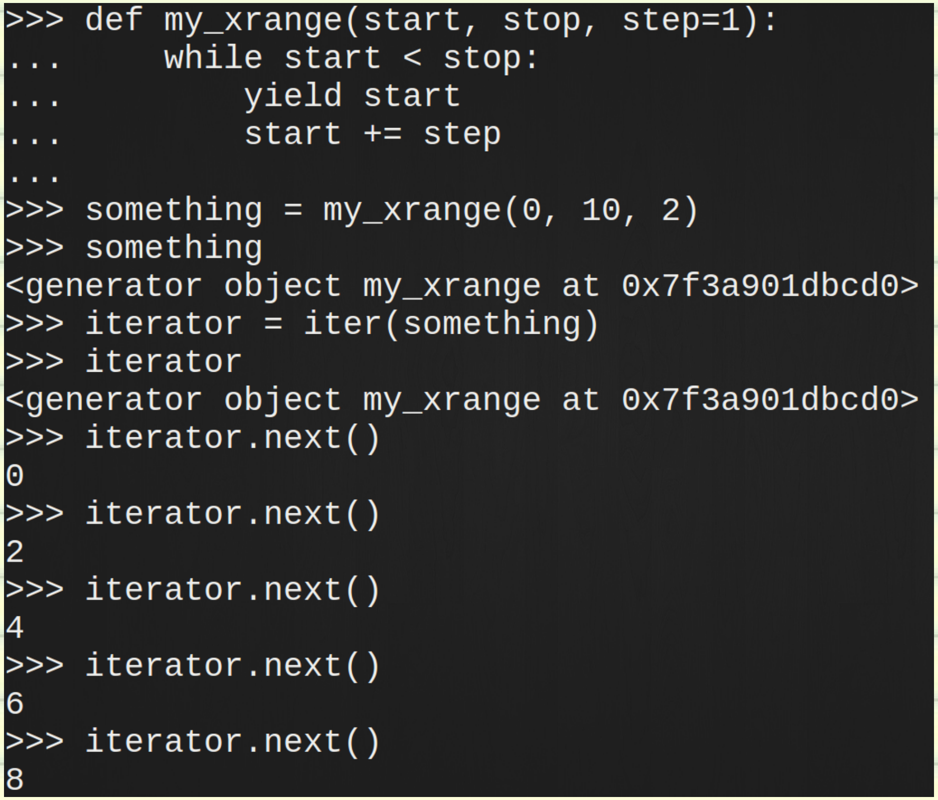
```
>>> def my_range(start, stop, step=1):
...     numbers = []
...     while start < stop:
...         numbers.append(start)
...         start += step
...     return numbers
...
>>> something = my_range(0, 10, 2)
>>> something
[0, 2, 4, 6, 8]
>>> iterator = iter(something)
>>> iterator
<listiterator object at 0x7fec930e5290>
>>> iterator.next()
0
>>> iterator.next()
2
>>> iterator.next()
4
>>> iterator.next()
6
```

New Stuff - Generators



```python
def my_range(start, stop, step=1):
    numbers = []
    while start < stop:
        numbers.append(start)
        start += step

    return numbers


def my_xrange(start, stop, step=1):
    while start < stop:
        yield start
        start += step


if __name__ == "__main__":
    print "my_range():"
    for i in my_range(0, 10):
        print "%i" % i,

    print "\n\nmy_xrange():"
    for i in my_xrange(0, 10):
        print "%i" % i,
```

Let's Experiment:



```
>>> def my_xrange(start, stop, step=1):
...     while start < stop:
...         yield start
...         start += step
...
>>> something = my_xrange(0, 10, 2)
>>> something
<generator object my_xrange at 0x7f3a901dbcd0>
>>> iterator = iter(something)
>>> iterator
<generator object my_xrange at 0x7f3a901dbcd0>
>>> iterator.next()
0
>>> iterator.next()
2
>>> iterator.next()
4
>>> iterator.next()
6
>>> iterator.next()
8
```

New Stuff – List Comprehensions

LIST COMPREHENSIONS!

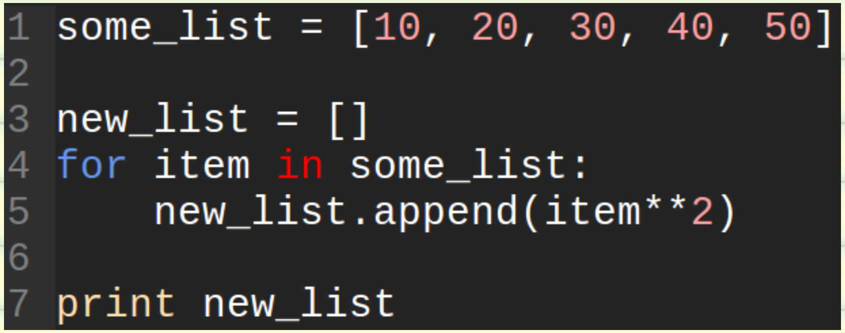
New Stuff – List Comprehensions

LIST COMPREHENSIONS!

A powerful feature of Python

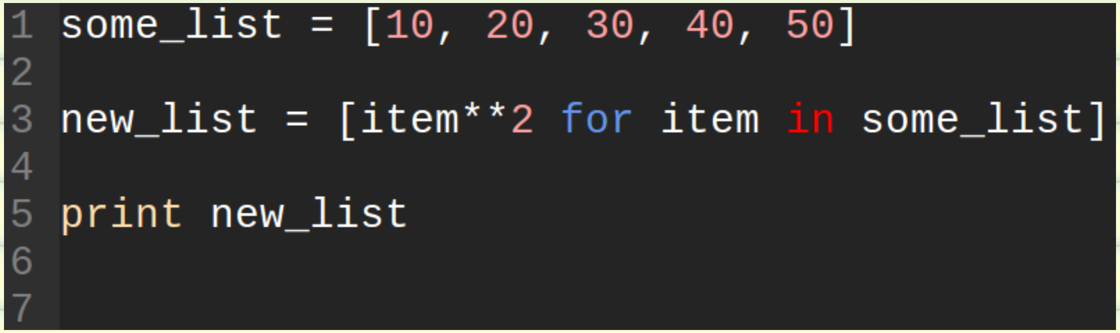
List Comprehensions

“Normal” for-loop

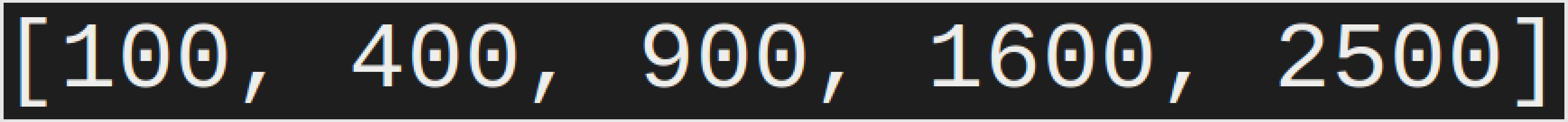


```
some_list = [10, 20, 30, 40, 50]

new_list = []
for item in some_list:
    new_list.append(item**2)

print new_list
```

List Comprehension



```
some_list = [10, 20, 30, 40, 50]

new_list = [item**2 for item in some_list]

print new_list
```



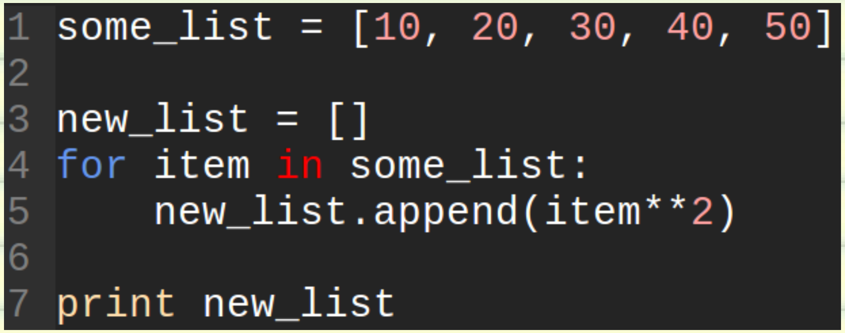
```
[100, 400, 900, 1600, 2500]
```

Standard Form

[expression for name in list]

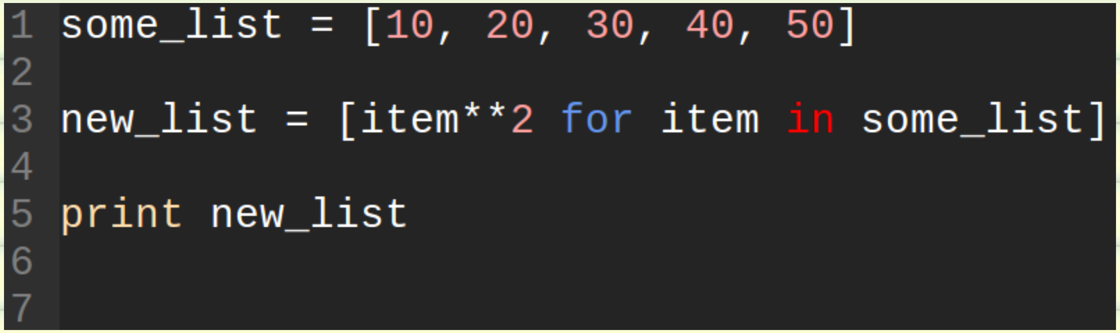
List Comprehensions

“Normal” for-loop

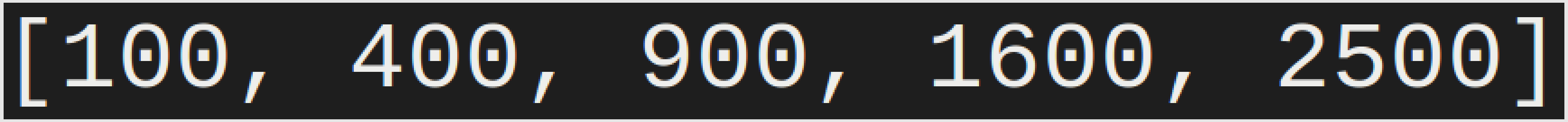


```
some_list = [10, 20, 30, 40, 50]

new_list = []
for item in some_list:
    new_list.append(item**2)

print new_list
```

List Comprehension

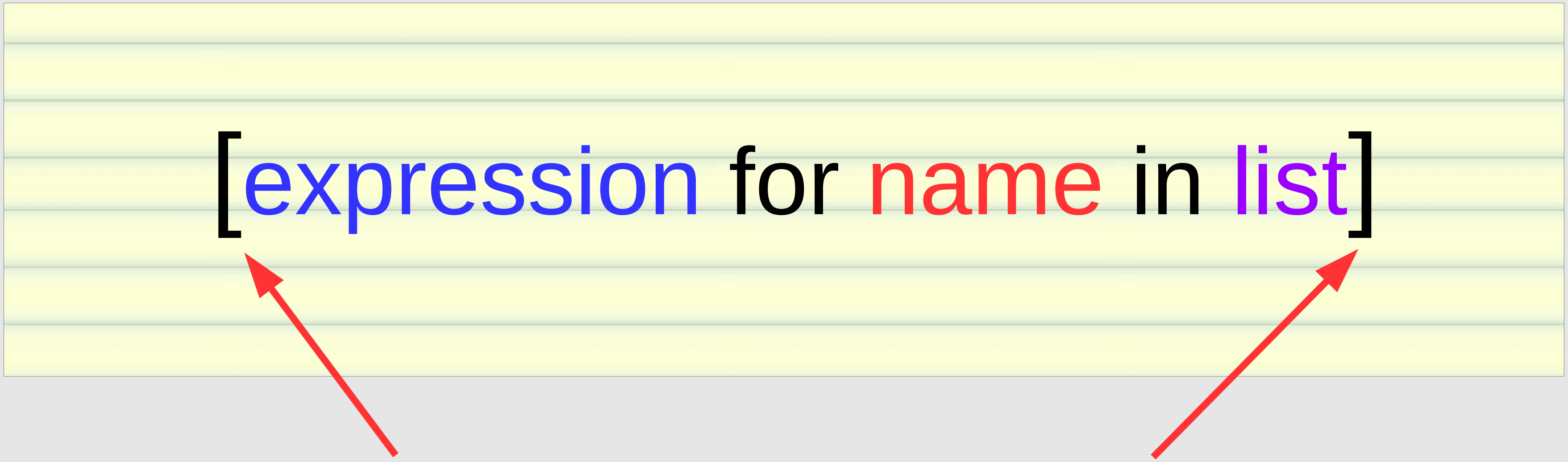


```
some_list = [10, 20, 30, 40, 50]

new_list = [item**2 for item in some_list]

print new_list


```



```
[100, 400, 900, 1600, 2500]
```

Standard Form

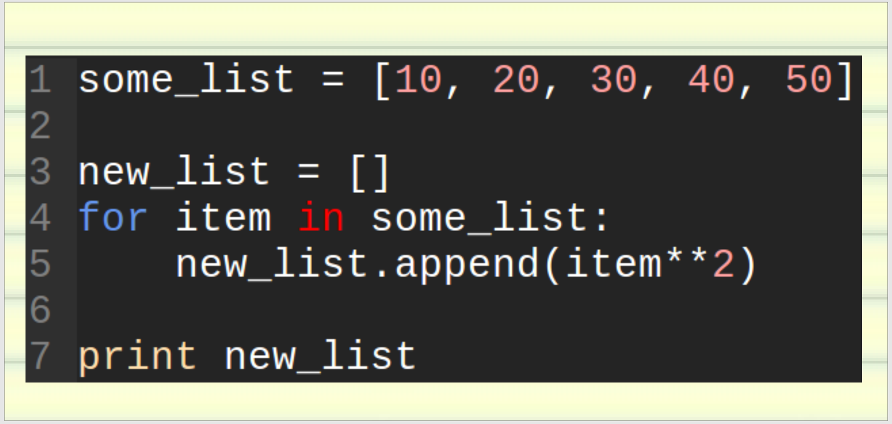


[expression for name in list]

implies you are creating a list

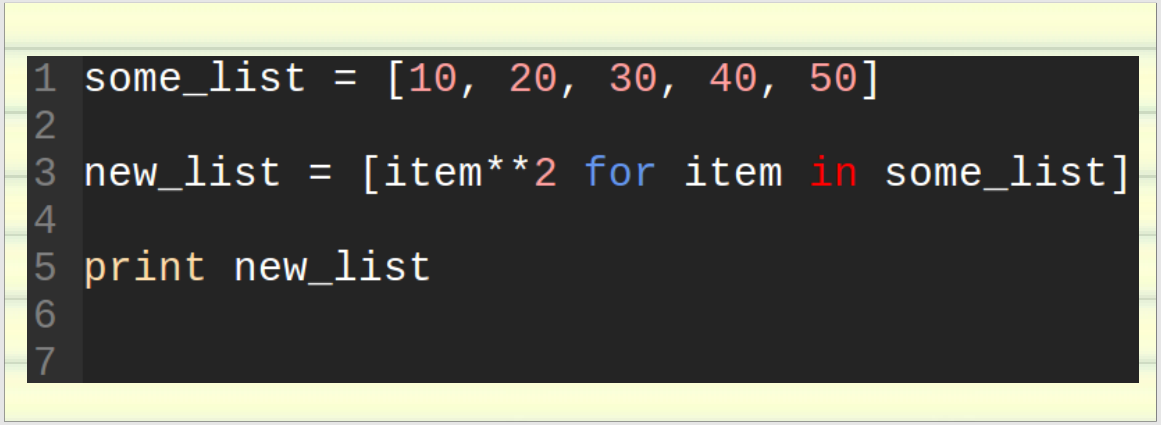
List Comprehensions

“Normal” for-loop

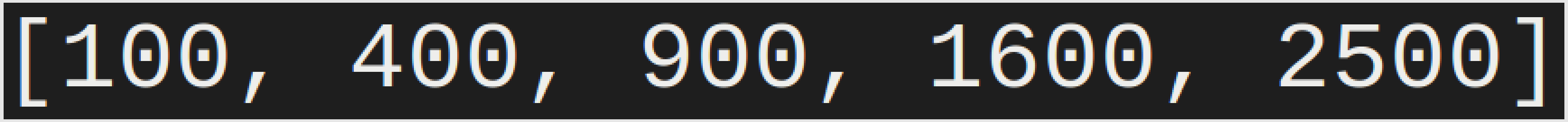


```
some_list = [10, 20, 30, 40, 50]

new_list = []
for item in some_list:
    new_list.append(item**2)

print new_list
```

List Comprehension

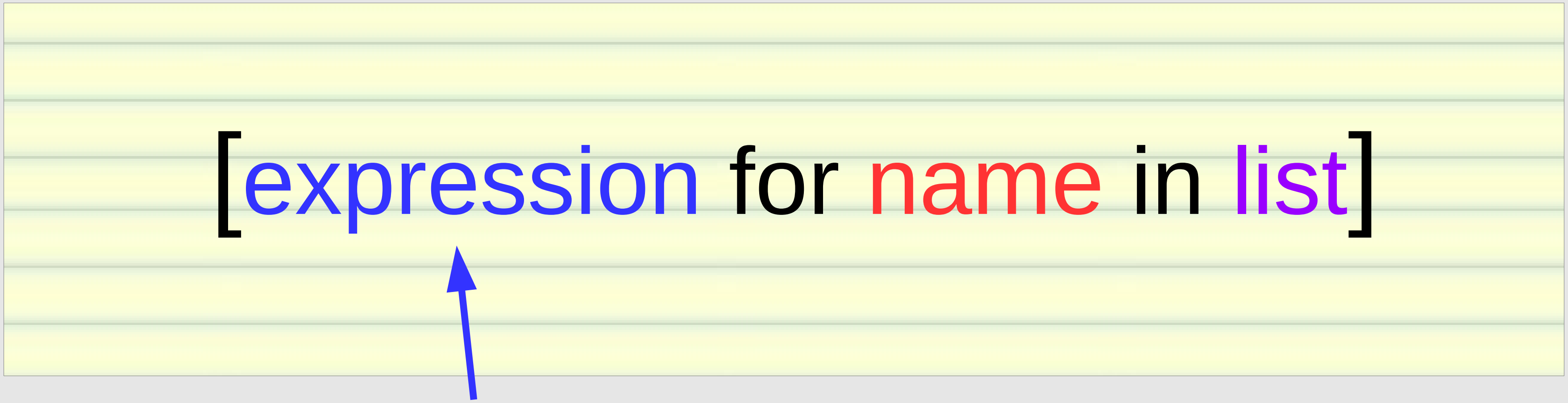


```
some_list = [10, 20, 30, 40, 50]

new_list = [item**2 for item in some_list]

print new_list
```



```
[100, 400, 900, 1600, 2500]
```

Standard Form

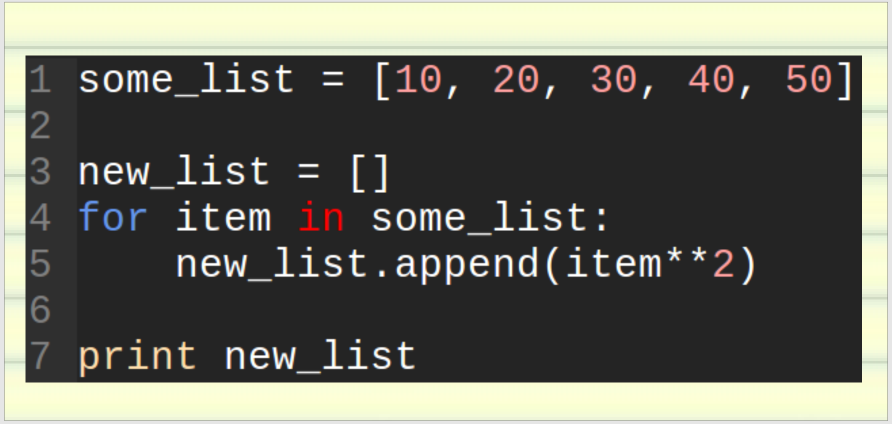


[expression for name in list]

elements in new list are each formed by the expression, which *can* use name

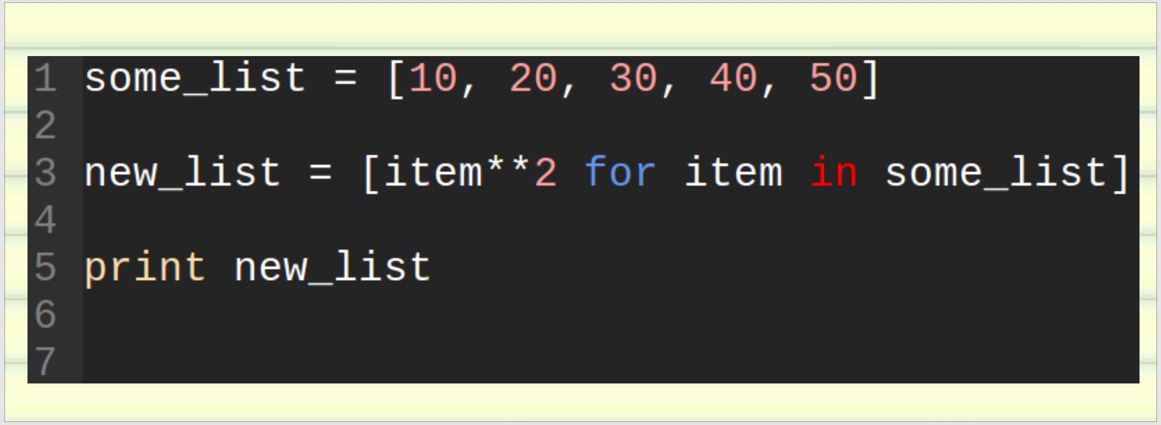
List Comprehensions

"Normal" for-loop

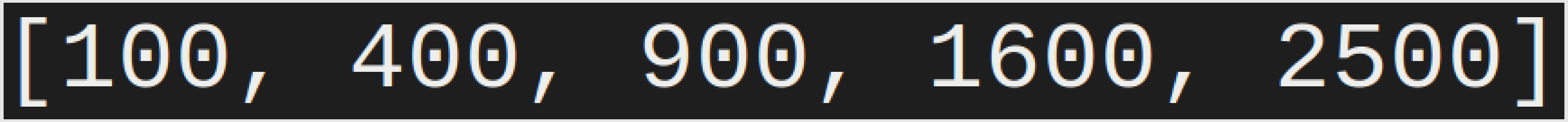


```
some_list = [10, 20, 30, 40, 50]

new_list = []
for item in some_list:
    new_list.append(item**2)

print new_list
```

List Comprehension

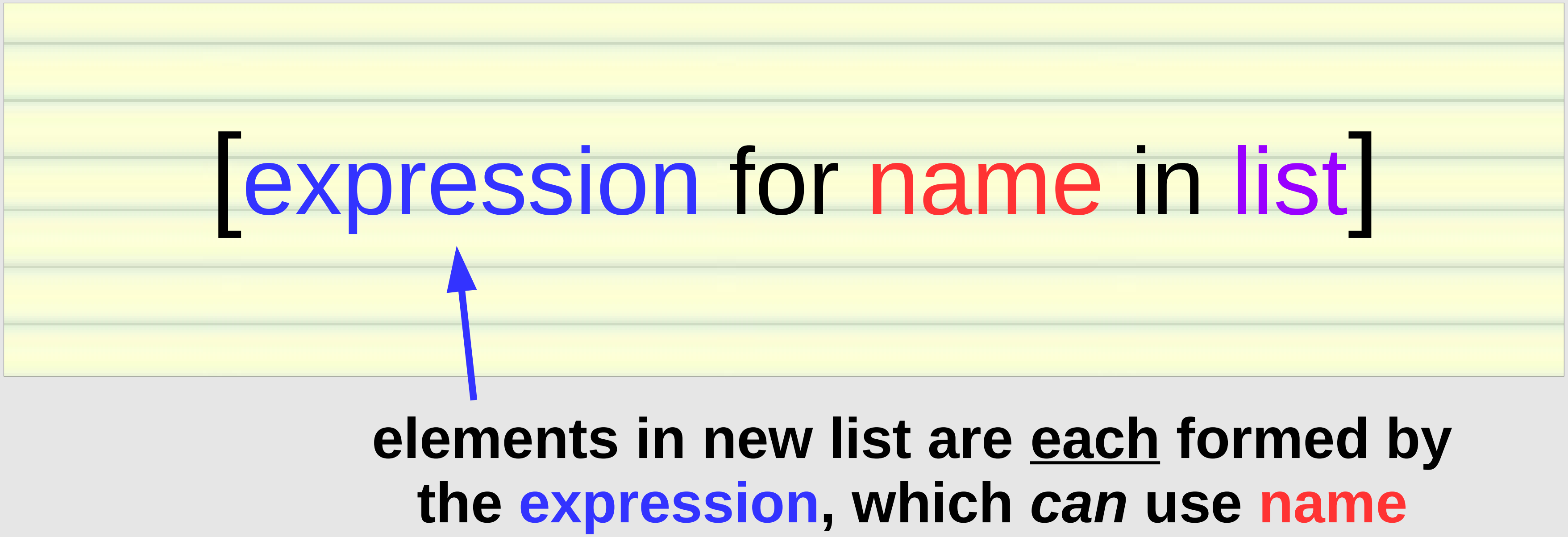


```
some_list = [10, 20, 30, 40, 50]

new_list = [item**2 for item in some_list]

print new_list

```



```
[100, 400, 900, 1600, 2500]
```

Standard Form



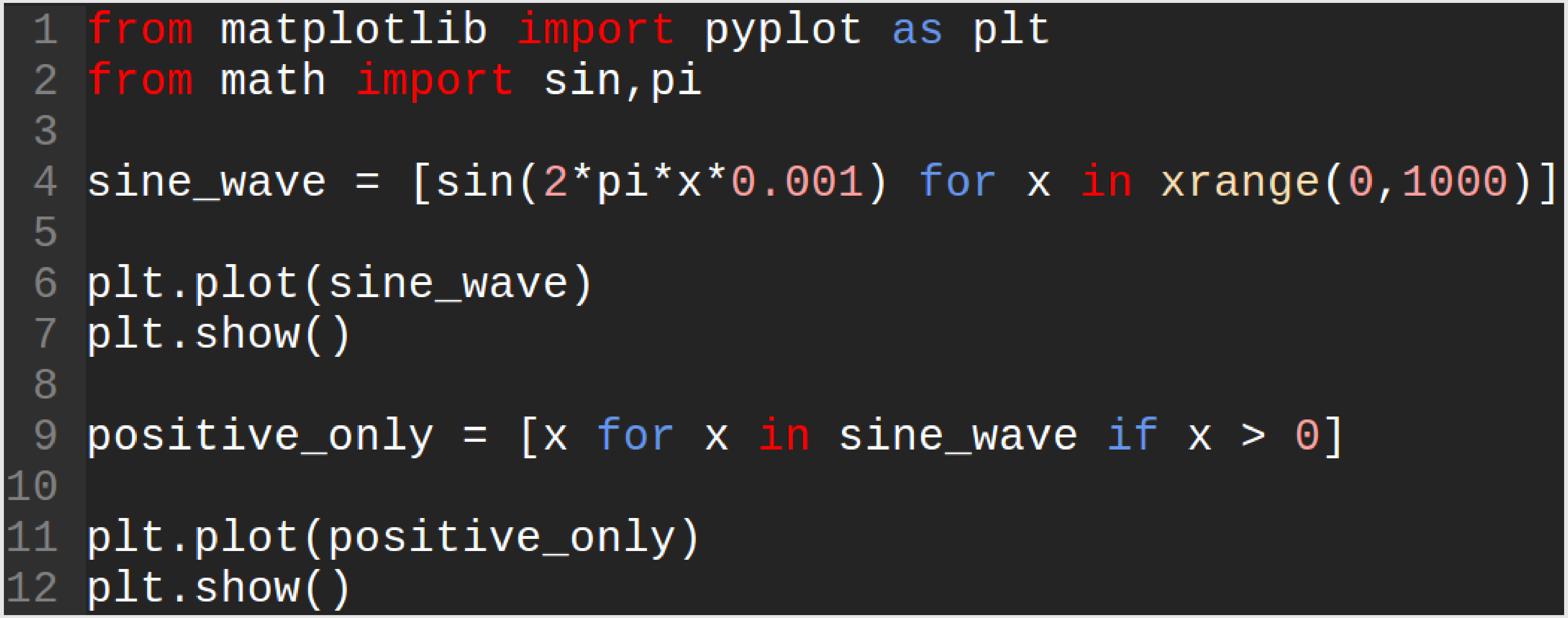
[expression for name in list]

elements in new list are <u>each</u> formed by the expression, which *can* use name

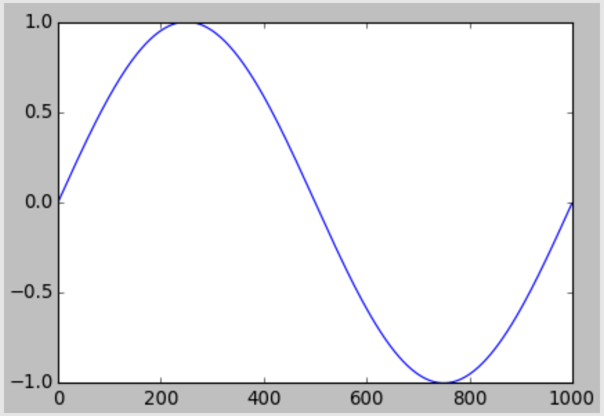
List Comprehensions

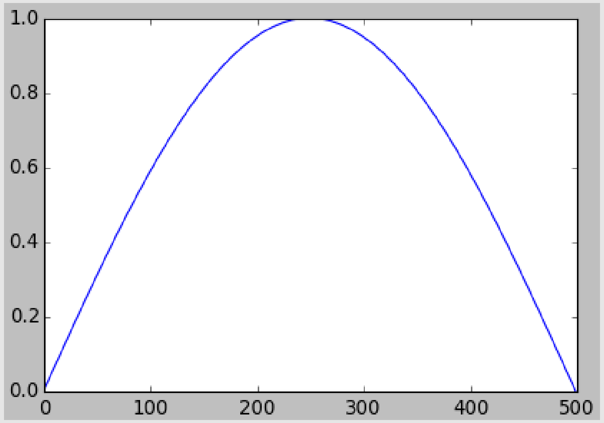
Standard(est) Form

[expression for name in list if filter]



```
from matplotlib import pyplot as plt
from math import sin,pi

sine_wave = [sin(2*pi*x*0.001) for x in xrange(0,1000)]

plt.plot(sine_wave)
plt.show()

positive_only = [x for x in sine_wave if x > 0]

plt.plot(positive_only)
plt.show()
```

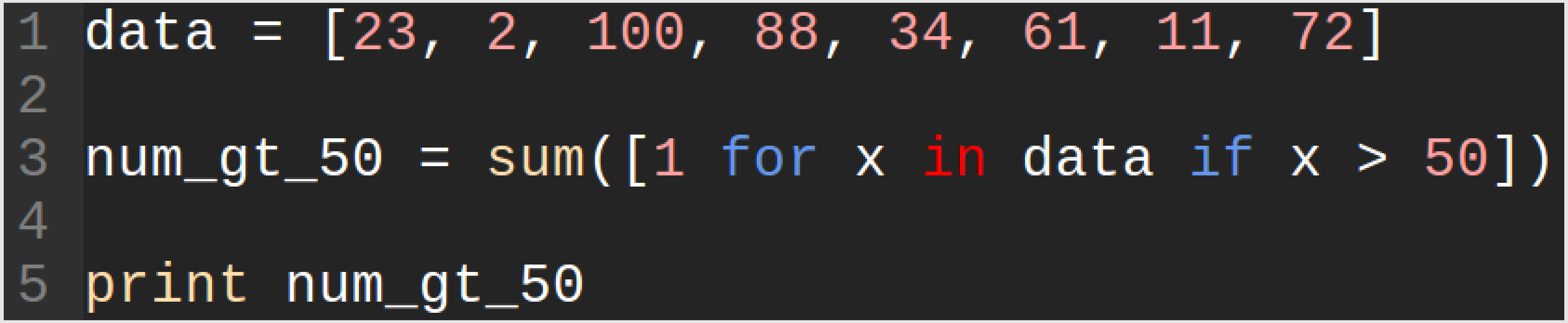




List Comprehensions

Standard(est) Form

[expression for name in list if filter]

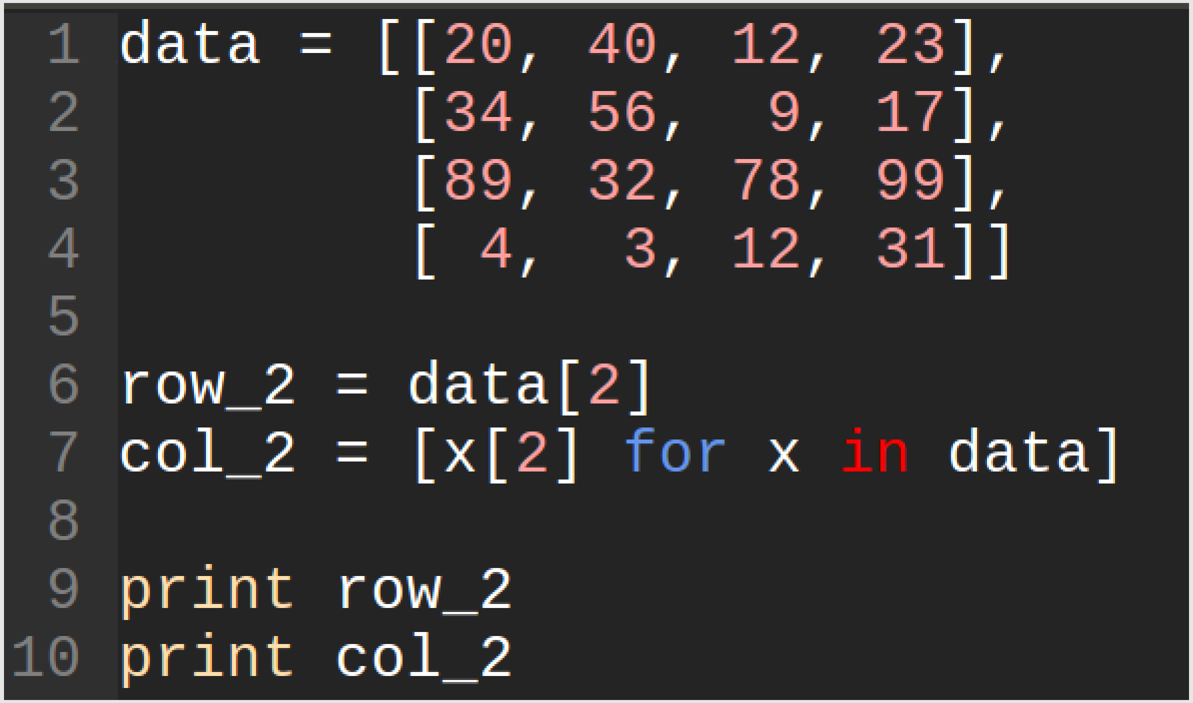


```
data = [23, 2, 100, 88, 34, 61, 11, 72]

num_gt_50 = sum([1 for x in data if x > 50])

print num_gt_50
```

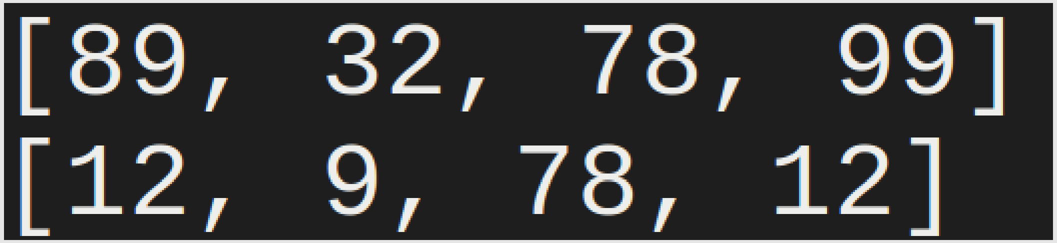
List Comprehensions

Standard(est) Form

[expression for name in list if filter]



```
data = [[20, 40, 12, 23],
        [34, 56,  9, 17],
        [89, 32, 78, 99],
        [ 4,  3, 12, 31]]

row_2 = data[2]
col_2 = [x[2] for x in data]

print row_2
print col_2
```



```
[89, 32, 78, 99]
[12, 9, 78, 12]
```