Functions and iteration

Kylie A. Bemis

Northeastern University
Khoury College of Computer Sciences

Northeastern University

Goals for today

- Review of Python basics
- Iteration and iterables
- Comprehensions
- Functions and modules

REVIEW:
PYTHON BASICS

Vocabulary: Objects

- Programs manipulate **objects**
- Objects are the "things" that exist in a program
- Objects:
  - Are stored in **memory** with **value**(s) associated with them
  - Have a **data type** that defines what **operations** can be performed
  - Are frequently bound to **variable** names that identify them

Vocabulary: Variables

- Programs refer to **variables**
- A variable consists of:
    - Storage location in memory
    - Name
    - Value (a specific object)
- **Assignment** binds a **value** to a variable **name**

Binding variables in Python

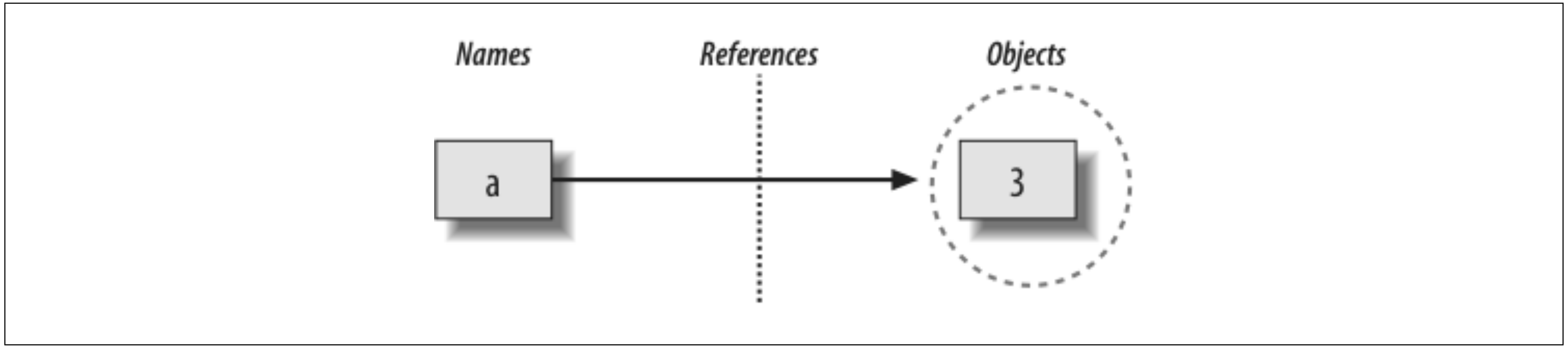
- Use equals sign (=) for variable assignment

```
          Name          Value
>>> pi = 3.14159265358979
```

- Creates a variable in memory
- Binds value to the variable name
- Variable name refers to bound value

Variables create **references**

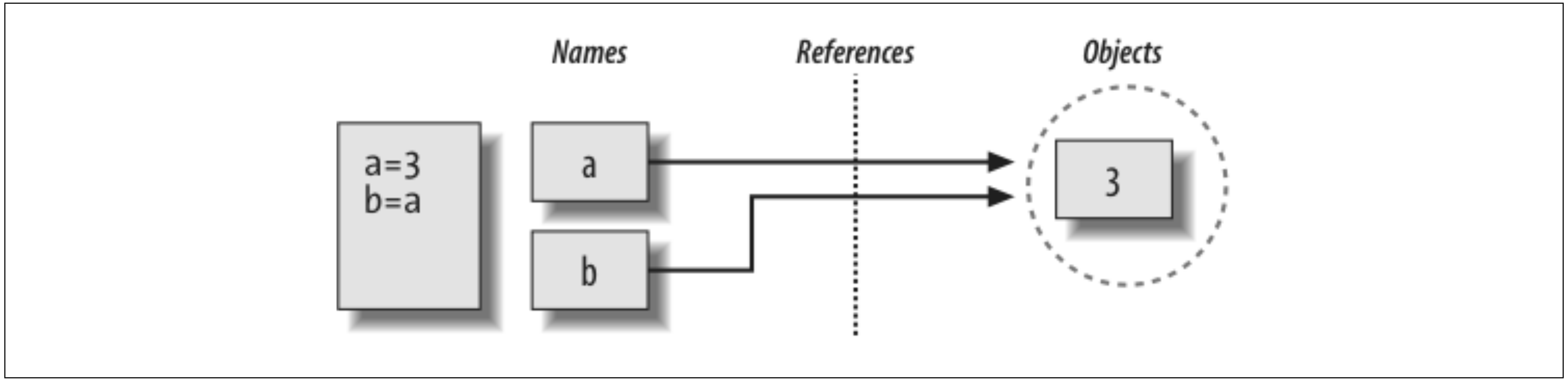
- Link between variable name and object
  - This link is called a ***reference***
  - An object may have multiple references
- Variables *point* to an object in memory



Learning Python. Mark Lutz. O'Reilly Media, 2013.

Shared references

- Multiple variables may reference the same object
  - Multiple variables may point to same location in memory
  - But only a single version of the object exists
- No additional memory is used



Learning Python. Mark Lutz. O'Reilly Media, 2013.

Types and values

- Objects have data **types**
- Types represent different kinds of values

```
>>> string1 = "Hello"
>>> string2 = "world"
>>> year = 2021
```

Strings (text)

Integer (number)

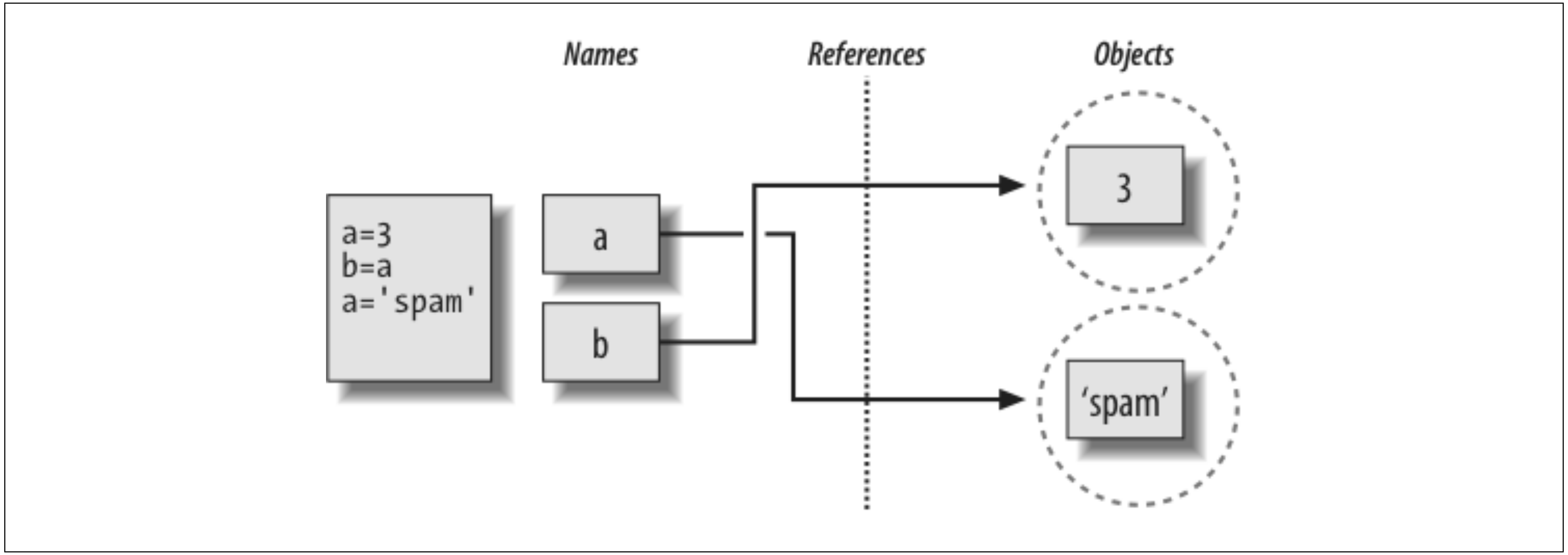
Types and operations

- Objects have data **types**
- Types define what operations are allowed

```
>>> string1 + " " + string2
"Hello world"

>>> string1 + " " + year
"Hello 2021"

>>> string1 * 3
"HelloHelloHello"

>>> string1 + 3
TypeError
```

Dynamic typing

- Variables may be re-bound to objects of different types
- Types belong to ***objects***, not variables



Learning Python. Mark Lutz. O'Reilly Media, 2013.

Python data types

Type	Example(s)
Integer	`1, 2, 3`
Float	`1.11, 2.22, 3.33`
String	`"Hello", "world"`
Boolean	`True, False`
NoneType	`None`

Python collections

- Lists
  - Ordered collection of arbitrary objects (mutable)
- Tuples
  - Ordered collection of arbitrary objects (immutable)
- Dictionaries
  - Unordered collection of key-value pairs
- Sets
  - Unordered collection of arbitrary objects

Lists

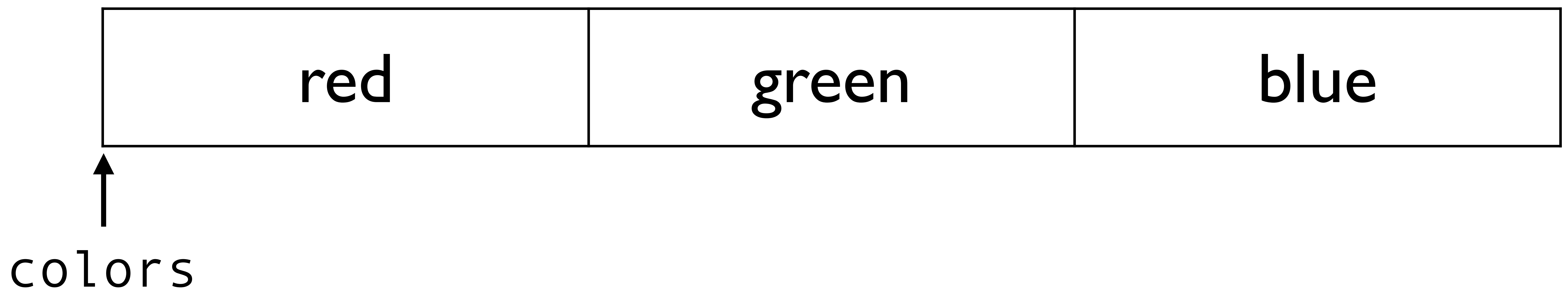
- Ordered collection of arbitrary objects
- Can be modified after creation
- Access elements by *offset*

`[]`	Empty list
`["red", "blue", 1, 2]`	List with 4 items
`["red", ["azure", "cyan"]]`	Nested list
`L[i]`	Access element at offset *i*

Indexing in Python

- A variable is a **pointer** to an object
- A pointer points to a location in memory
- A pointer to an *ordered collection* points to the *beginning* of the collection

```
colors = ["red", "green", "blue"]
```



Indexing in Python

```
colors = ["red", "green", "blue", "alpha"]
```

`colors[0]`	`colors[1]`	`colors[2]`	`colors[3]`
red	green	blue	alpha

`0` `1` `2` `3` `4`

Access elements by offset using brackets **[ ]**

Indexing

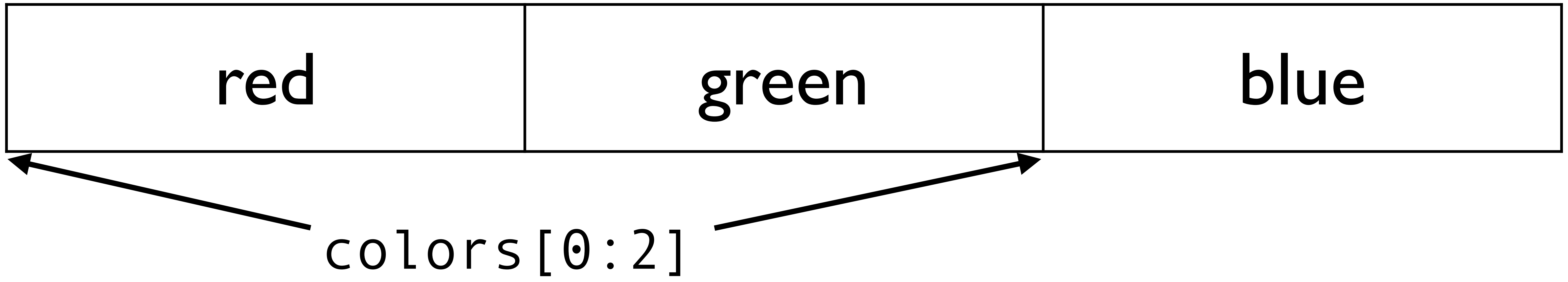
```
colors = ["red", "green", ["blue", "cyan", "indigo"]]
```

Expression	Value
`colors[0]`	"red"
`colors[2]`	["blue", "cyan", "indigo"]
`colors[2][1]`	"cyan"
`colors[-1]`	["blue", "cyan", "indigo"]
`colors[-2]`	green

Slicing in Python

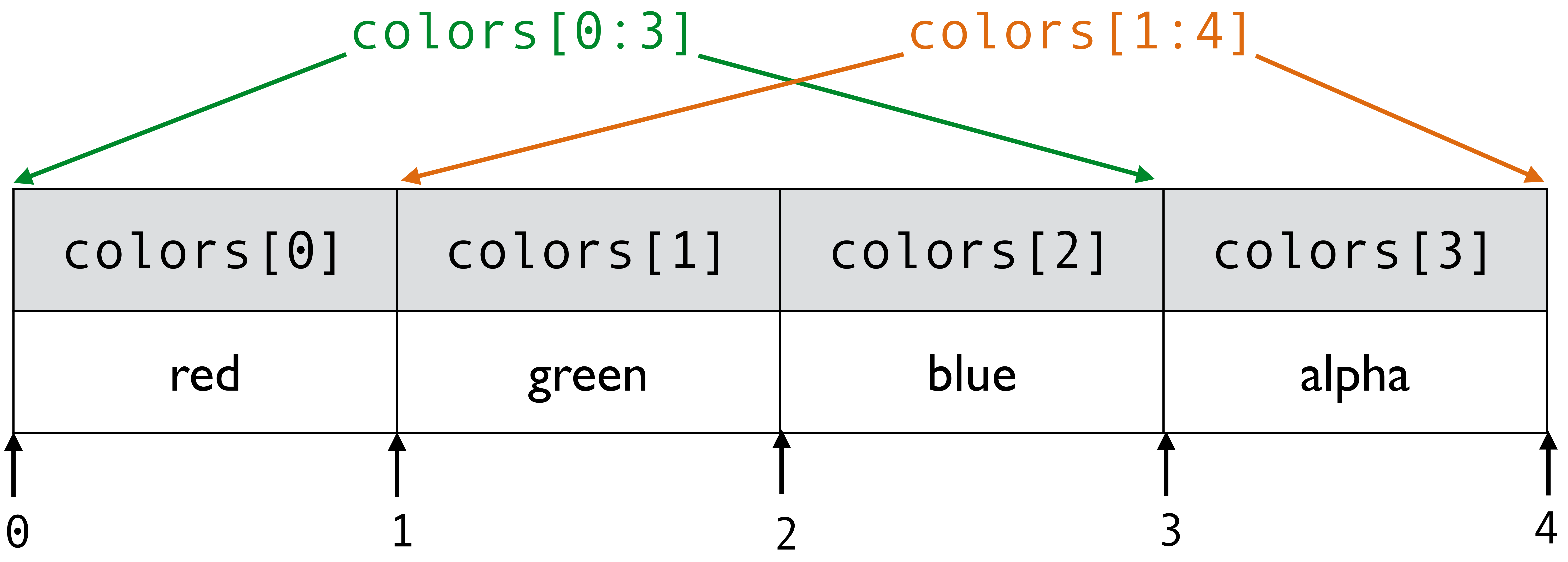
- **Slicing** is a powerful method of subsetting
- Access a subsequence of an ordered collection
- Slice a sequence using `start:end`

```
colors = ["red", "green", "blue"]
```



Slicing a list

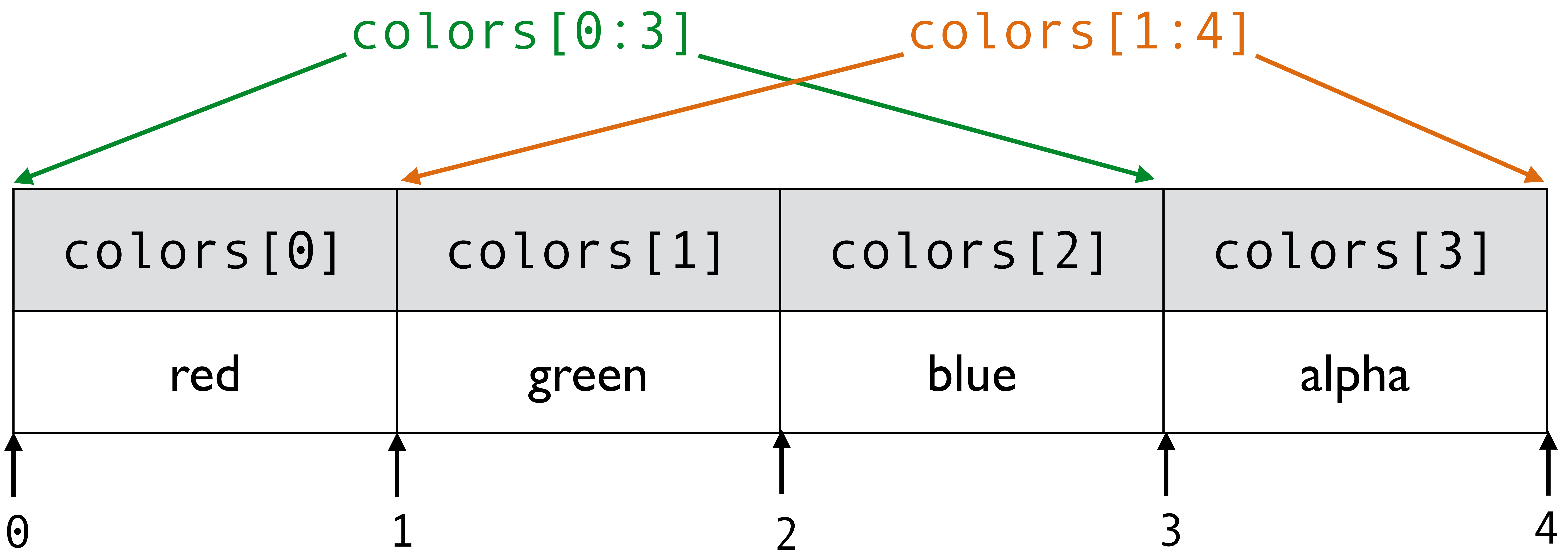
```
colors = ["red", "green", "blue", "alpha"]
```



Slice a sequence elements with `start:end`

Slicing a list

```
colors = ["red", "green", "blue", "alpha"]
```




```
>>> colors[0:3]
["red", "green", "blue"]
```

```
>>> colors[1:4]
["green", "blue", "alpha"]
```

Slicing

```
powers = [1, 2, 4, 8, 16, 32, 64, 128, 256, 512]
```

Expression	Value
`powers[0]`	1
`powers[-1]`	512
`powers[0:3]`	[1, 2, 4]
`powers[6:]`	[64, 128, 256, 512]
`powers[:-3]`	[1, 2, 4, 8, 16, 32, 64]

Functions and methods

- **Functions** are programming verbs
  - *Do something*, e.g., `print()`
  - *Return a value*, e.g., `len()`
- Some object types support specialized functions called ***methods***
  - Methods belong to the object
  - Methods may modify the object
  - Called via `object.method()`

List methods

```
fib = [1, 1, 2, 3, 5, 8, 13, 21, 34]
```

Method	**Description**
`fib.append(55)`	Append a value to the list
`fib.extend([55, 89, 144])`	Append a list (iterable) to the list
`fib.index(8)`	Return first index of a value
`fib.count(1)`	Count occurrences of a value
`fib.reverse()`	Reverse list *in-place*

Methods

- Find all available methods for a type
  - `help(list)`
- "Magic" methods surrounded by underscores
  - `__add__` implements `+`
  - `__mul__` implements `*`
  - More on magic methods later
- Methods may modify original object!

Tuples

- Ordered collection of arbitrary objects
- ***Cannot*** be modified after creation

`()`	Empty tuple
`(1,)`	Tuple with 1 items
`("red", "blue", 1, 2)`	Tuple with 4 items
`"red", "blue", 1, 2`	Tuple with 4 items (no parentheses)
`("red", ("azure", "cyan"))`	Nested tuple
`T[i]`	Access element at offset *i*

Mutable vs. immutable

- Mutable object (e.g., lists)
  - Can be modified after creation
  - More memory-efficient
  - Use for data that changes
- Immutable object (e.g., tuples)
  - Cannot be modified
  - Safer and provides integrity
  - Use for data that *doesn't* change

Shared references and mutability

Modifying a mutable object updates it everywhere!

```
>>> a = [1, 2, 3]
>>> b = a
>>> b
[1, 2, 3]
```

a and b share same reference.

```
>>> a[1] = 100
>>> a
[1, 100, 3]

>>> b
[1, 100, 3]
```

Both references see changes

Dictionaries

- Unordered collection of key-value pairs
- Keys must be immutable
- Can be modified after creation

`{}`	Empty dictionary
`{"name": "Kylie", "age": 31}`	Dictionary with 2 items
`dict(name="Kylie", age=31)`	Dictionary with 2 items
`D[key]`	Access element by key

Operations on a dictionary

```
trees = {"maple": 3, "pine": 7, "oak": 4, "spruce": 6}
```

Expression	Value
`trees["maple"]`	3
`trees["pine"]`	7
`"oak" in trees`	True
`"birch" in trees`	False
`trees.keys()`	["maple", "pine", "oak", "spruce"]
`trees.values()`	[3, 7, 4, 6]

Sets

- Unordered collection of unique objects
- Duplicates are not allowed
- Can be modified after creation

`set()`	Empty set
`{1, 2, 3}`	Set with 3 items
`{1, 1, 2, 3}`	Set with 3 items
`{"red", "blue", 1, 2}`	Set with 4 items
`x in S`	Test if element is in set

Python collections

✓ Lists

- Ordered collection of arbitrary objects (mutable)

✓ Tuples

- Ordered collection of arbitrary objects (immutable)

✓ Dictionaries

- Unordered collection of key-value pairs

✓ Sets

- Unordered collection of arbitrary objects

Conditionals

- Control the flow of program logic
- Branch between different choices
- `<condition>` is a boolean

```
if <condition>:
    <expression>
    <expression>
    ...
```

```
if <condition>:
    <expression>
    <expression>
    ...
else:
    <expression>
    <expression>
    ...
```

```
if <condition>:
    <expression>
    <expression>
    ...
elif <condition>:
    <expression>
    <expression>
    ...
else:
    <expression>
    <expression>
    ...
```

ITERATION AND ITERABLES

Loops

- Repeat a set of actions multiple times
- `while` loops
  - Repeat loop until a condition is (not) satisfied
- `for` loops
  - Iterate over elements of a sequence

`while` loops

- Repeat a set of actions until:
  - The condition is (not) satisfied
  - A `break` is encountered

```
while <condition>: # loop test
    <expression>   # loop body
    <expression>
    ...
else:  # if condition is False
    <expression>
    <expression>
    ...
```

`while` example

- Repeat a set of actions until:
  - The condition is (not) satisfied
  - A `break` is encountered

```
while True:
    print("Ctrl-C to escape!")
```

```
i = 0
while i < 5:
    print(i)
    i = i + 1
```

`for` loops

- Iterate over elements of a sequence:
  - Operate on each element in loop body
  - Continue until sequence is exhausted

```
for <variable> in <object>: # initialize loop
    <expression>            # loop body
    <expression>.           # use <variable>
    ...
else:  # if sequence is exhausted
    <expression>
    <expression>
    ...
```

`for` example

- Iterate over elements of a sequence:
  - Operate on each element in loop body
  - Continue until sequence is exhausted

```
for i in range(5):
    print(i)
```

is (roughly) equivalent to:

```
i = 0
while i < 5:
    print(i)
    i = i + 1
```

Loop vocabulary

- `break`
  - Exit out of the loop
- `continue`
  - Jump back to top of loop and continue iterating
- `pass`
  - Do nothing — empty statement placeholder

Iterating through a file

- Suppose we want to process each line of a file
  - If we can't fit the whole file in memory?
  - If we don't know how many lines in the file?
  - Unfortunately, `readlines()` loads whole file at once...

```
f = open("mtcars.csv")

data = []

line = f.readline()
while line:
    data.append(line.split(","))
    line = f.readline()
```

Empty strings evaluate to `False`

Use a file iterator

- Get an object that iterates through every line
  - Use `iter()` to get an iterator object
  - Use `next()` to get the next element

```
f = open("mtcars.csv")

data = []

I = iter(f)
while True:
    try:
        line = next(I)
    except StopIteration:
        break
    data.append(line.split(","))
```

Check for *exception* to stop iteration

Use a `for` loop

- A `for` loop will automatically use an iterator
  - Iterate through all items in a *iterable* collection
  - Individual items may not exist until requested
  - Simple and powerful!

```
f = open("mtcars.csv")

data = []

for line in f:
    data.append(line.split(","))
```

`for` loop automatically uses an iterator

Iterables

- *Iterable* objects can be iterated over
  - Lists, tuples, strings, files, etc.
  - Elements may be generated *on-demand*
  - Elements do not need to be realized all at once
  - Any object that implements `__iter__()`
- Combine with `for` loops for easy iteration
  - No need to handle the iterator directly
  - Loop construct handles the details

Using `range()`

- Iterate over a range of integers
- `range(stop)`
  - `range(4) → [0, 1, 2, 3]`
- `range(start, stop, step)`
  - `range(2, 10, 2) → [2, 4, 6, 8]`

```
# print ints 0 - 9

for i in range(10):
 print(i)
```

Magic of `range()`

- Does not create entire range of integers
- Provides an iterator to generate elements
  - Able to iterate over lists longer than memory
  - If we `break` early, future elements are never created

```
# do not run unattended!

for i in range(int(1e12)):
 print(i)
```

Would be > 1 TB
if realized as a list!

Using `enumerate()`

- Iterate over both offsets and elements
  - Returns an iterator over tuples
  - Useful when you need to operate on both

Tuples are "unpacked" when assigned to multiple variables

```
lst = ["red", "green", "blue"]

for i, elt in enumerate(lst):
    print(i, ":", elt)
```

Iterating over multiple items

- Use tuples to iterate over multiple items
  - Use multiple iterator variables in a `for` loop
  - Tuples are "unpacked" into multiple variable assignments

```
hotpink = {"red": 255, "green": 105, "blue": 180}

for col, val in hotpink.items():
    print(col, ":", val)
```

```
[("red", 255), ("green", 105), ("blue", 180)]
```

Using `zip()`

- Use `zip()` to iterate over multiple lists
  - Creates iterator that returns tuples of corresponding items
  - Tuples are "unpacked" into multiple variable assignments

```
colors = ["red", "green", "blue"]
values = [255, 105, 180]

for col, val in zip(colors, values):
    print(col, ":", val)
```

```
[("red", 255), ("green", 105), ("blue", 180)]
```

Using `zip(*)`

- Use `*` to unpack tuples into multiple arguments
  - Useful to programmatically pass a tuple of arguments
  - Can be used to (practically) perform the inverse of `zip()`

```
hotpink = {"red": 255, "green": 105, "blue": 180}

colors2, values2 = zip(*hotpink.items())
```

```
zip(("red", 255), ("green", 105), ("blue", 180))
```

COMPREHENSIONS

Processing a list

- Suppose we want to iterate over a list:
  - 1. Process each element of the list
  - 2. Return the results as a new list
- Try building the list using a `for` loop?

```
lst = [1, 2, 3, 4, 5, 6]

out = []
for x in lst:
    out.append(x ** 2)
```

List comprehensions

- **Create a list** using results of **iteration**
- Powerful list processing mechanism
- Syntax borrows from math set notation

```
lst = [1, 2, 3, 4, 5, 6]

[x ** 2 for x in lst]
```

```
[1, 4, 9, 16, 25, 36]
```

List comprehensions

- **Create a list** using results of **iteration**
- Embed a `for` loop inside brackets `[]`
- Efficiently returns list of elements

```
[<expression> for <variable> in <iterable>]
```

List comprehensions

- **Create a list** using results of **iteration**
- Embed a `for` loop inside brackets `[]`
- Efficiently returns list of elements

```
[<expression> for <variable> in <iterable>]
```

Become elements of the list

Variable can be referenced by `<expression>`

FUNCTIONS

Why functions?

- Code should be **reusable**!
- *Decomposition* creates structure
  - **Self-contained** chunk of code
  - **Coherent** and **organized** design
- Performs a **single task** using *input*
- Returns a value as *output*

Using functions

- We use many functions (e.g., `print()`)
- *Abstraction* supports usability
  - Functions are a "**black box**" for users
  - No need to know implementation details
- Supported usage should be **documented**
  - Function specification
  - Docstring

Function characteristics

- Functions in Python have:
  - Name
  - **Parameters** (0 or more)
  - **Docstring** (optional, but recommended)
  - Body (implementation)
  - **Return** value
- *Good* functions are intuitive to use

Defining a function in Python

```python
def mysum( x ):
    """
    Sums values of an iterable
    param x: An iterable to sum the values
    returns: The sum
    """
    xsum = 0
    for xi in x:
        xsum += xi
    return xsum
```

Defining a function in Python

Function keyword

Name

Parameter

```python
def mysum( x ):
    """
    Sums values of an iterable
    param x: An iterable to sum the values
    returns: The sum
    """
    xsum = 0
    for xi in x:
        xsum += xi
    return xsum
```

Docstring

Body

Return value

```python
mysum([1, 2, 3])
```

Usage (later in code)

Returning values in Python functions

- Use `return` to **return a value** from a function
- Returning a value <u>*immediately exits*</u> the function
- If missing, Python returns `None`
- Different from `print()`!

Exercise: Stem and leaf plot

- Create a function for making a stem plot
- A simple "old-school" histogram

44, 46, 47, 49, 63, 64, 66, 68, 68, 72, 72, 75, 76, 81, 84, 88, 106

```
Stem | Leaf
   4 | 4 6 7 9
   5 |
   6 | 3 4 6 8 8
   7 | 2 2 5 6
   8 | 1 4 8
   9 |
  10 | 6
```

MODULES

Python modules

- File of Python code with filename ending in ".py"
- Collection of Python definitions and statements
  - **Decompose** complex codebase into collection of related functions
  - Easier to **re-use** and **maintain**
- Everything in a module shares a **similar purpose**

Using modules

- Save your module as "my_module.py"
- Import module for use in another script
- Objects from module referred to by alias

```
import my_module

my_module.my_function()
```

Use module name as alias to prefix its functions

Import a module with an alias

- Save your module as "my_module.py"
- Import module for use in another script
- Objects from module referred to by alias

```
import my_module as my

my.my_function()
```

Specify a different alias to refer to module

Import specific objects from a module

- Save your module as "my_module.py"
- Import module for use in another script
- Import specific objects

```
from my_module import my_function
my_function()
```

No alias needed for specific function imports

Standard library modules

- math
- random
- itertools
- string
- datetime
- os
- sys
- etc.