Basics of Programming with Python

Kylie A. Bemis

Northeastern University
Khoury College of Computer Sciences

Northeastern University

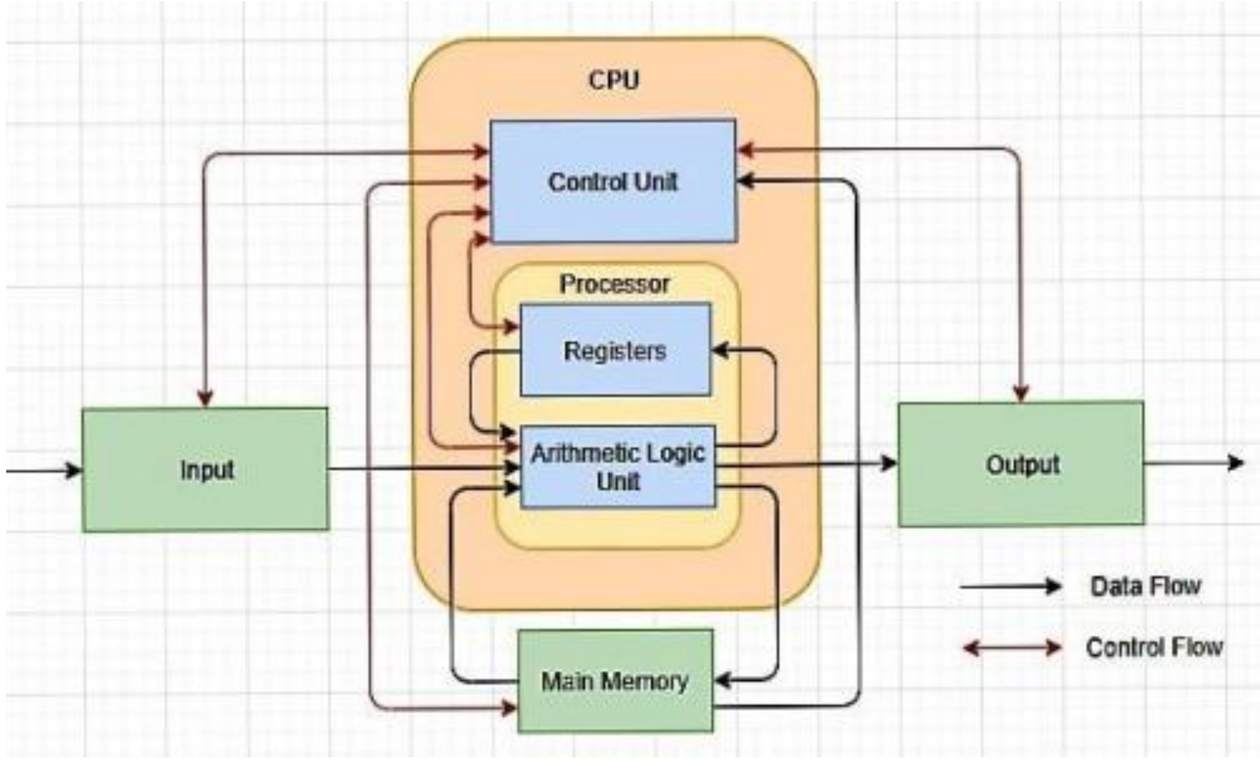
Goals for today

- How Python works
- Variables and data types
- Collection data structures
- Control flow
- File I/O

HOW PYTHON WORKS

How computers work

- **Central processing unit** (CPU)
- **Machine code** instructions (specific to architecture)
- **Assembly language** provides a human-readable version
- Intel x86, ARM, etc.

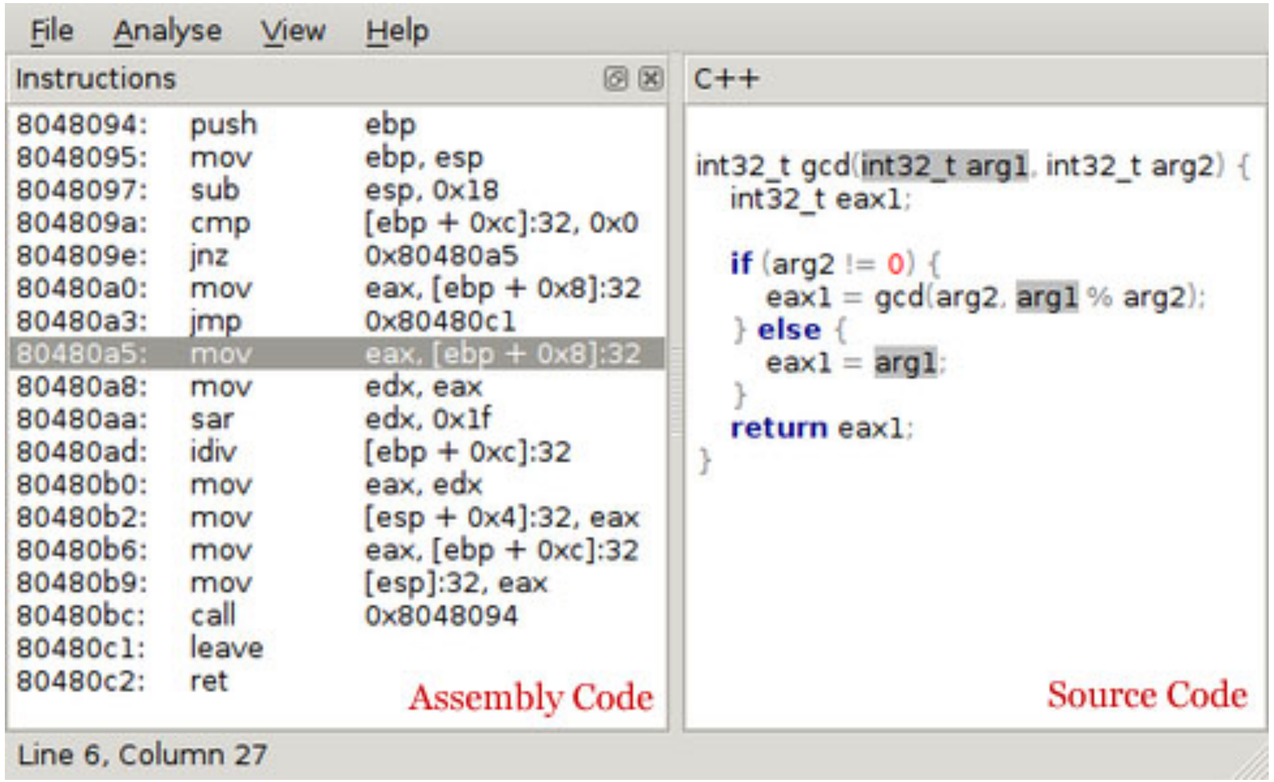


Assembly vs. machine code

Machine code bytes	Assembly language statements
	`foo:`
`B8 22 11 00 FF`	`movl $0xFF001122, %eax`
`01 CA`	`addl %ecx, %edx`
`31 F6`	`xorl %esi, %esi`
`53`	`pushl %ebx`
`8B 5C 24 04`	`movl 4(%esp), %ebx`
`8D 34 48`	`leal (%eax,%ecx,2), %esi`
`39 C3`	`cmpl %eax, %ebx`

Compiled languages

- Compiler converts **source** code to **machine** code
- Entire program must be compiled to run
- Runs directly on CPU
- C, C++, Java, Haskell, Rust



Interpreted languages

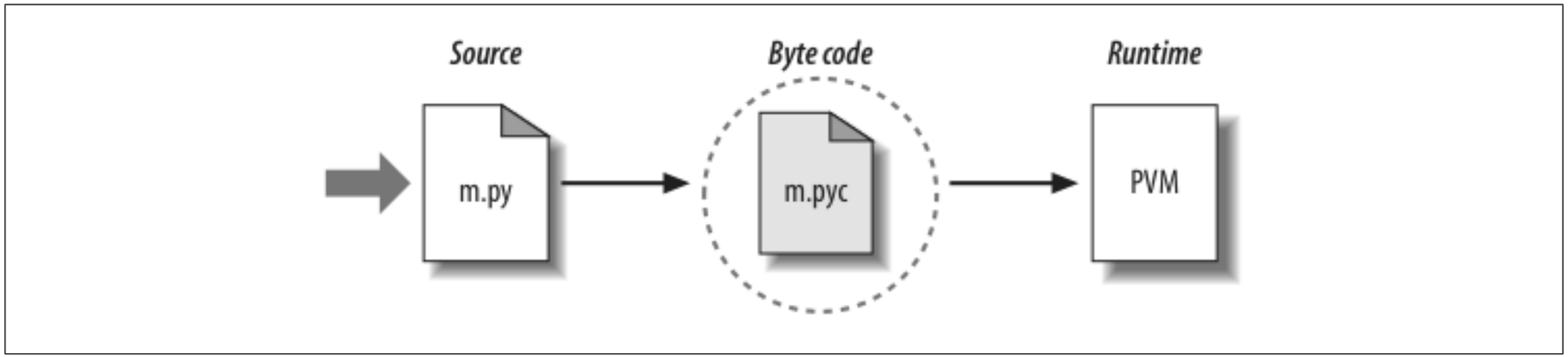
- Source code is executed by an **interpreter**
- Program can be parsed and run line-by-line
- Not run directly on CPU
- Python, R, JavaScript, Perl, Ruby

Compiled vs. interpreted

- Compiled languages
  - Machine code runs directly on CPU
  - Faster and more efficient
  - Non-interactive, complex, requires compilation
- Interpreted languages
  - Source code parsed and run by interpreter
  - Slower and less efficient
  - Interactive, flexible, rapid prototyping

Python interpreter

- On execution, Python interpreter compiles **source** code to **byte** code
- Byte code is low-level "intermediate" code
- Runs on Python Virtual Machine (PVM)



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

Python byte code

```
In [1]: def hello_world():
   ...:         print("Hello, world!")
   ...:

In [2]: import dis

In [3]: dis.dis(hello_world)
  2           0 LOAD_GLOBAL              0 (print)
              2 LOAD_CONST               1 ('Hello, world!')
              4 CALL_FUNCTION            1
              6 POP_TOP
              8 LOAD_CONST               0 (None)
             10 RETURN_VALUE
In [4]: hello_world()
Hello, world!
```

VARIABLES AND TYPES

Vocabulary: Objects

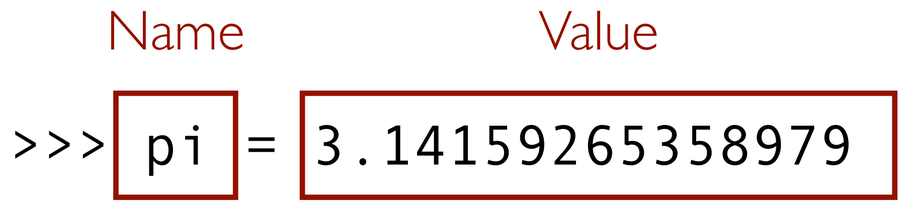
- Programs manipulate **objects**
- Objects are the "things" that exist in a program
- Objects:
  - Are stored in **memory** with **value**(s) associated with them
  - Have a **data type** that defines what **operations** can be performed
  - Are frequently bound to **variable** names that identify them

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



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

Using variables

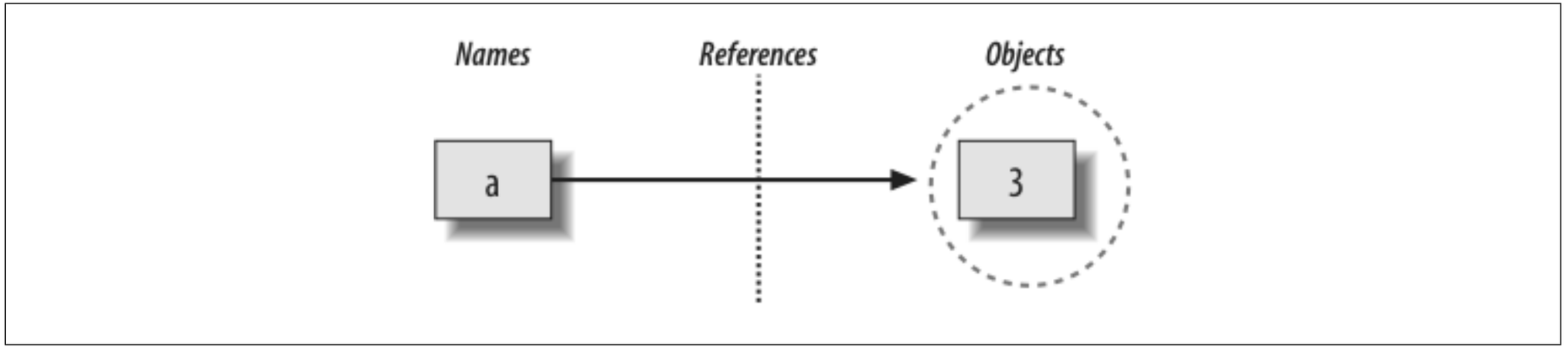
- Use variables for clear, expressive code

```
>>> pi = 3.14159265358979
>>> radius = 2.22
>>> area = pi * (radius**2)
```

- Easier to read
- Reusable, portable code
- Arithmetic is an **operation**

Variables create **references**

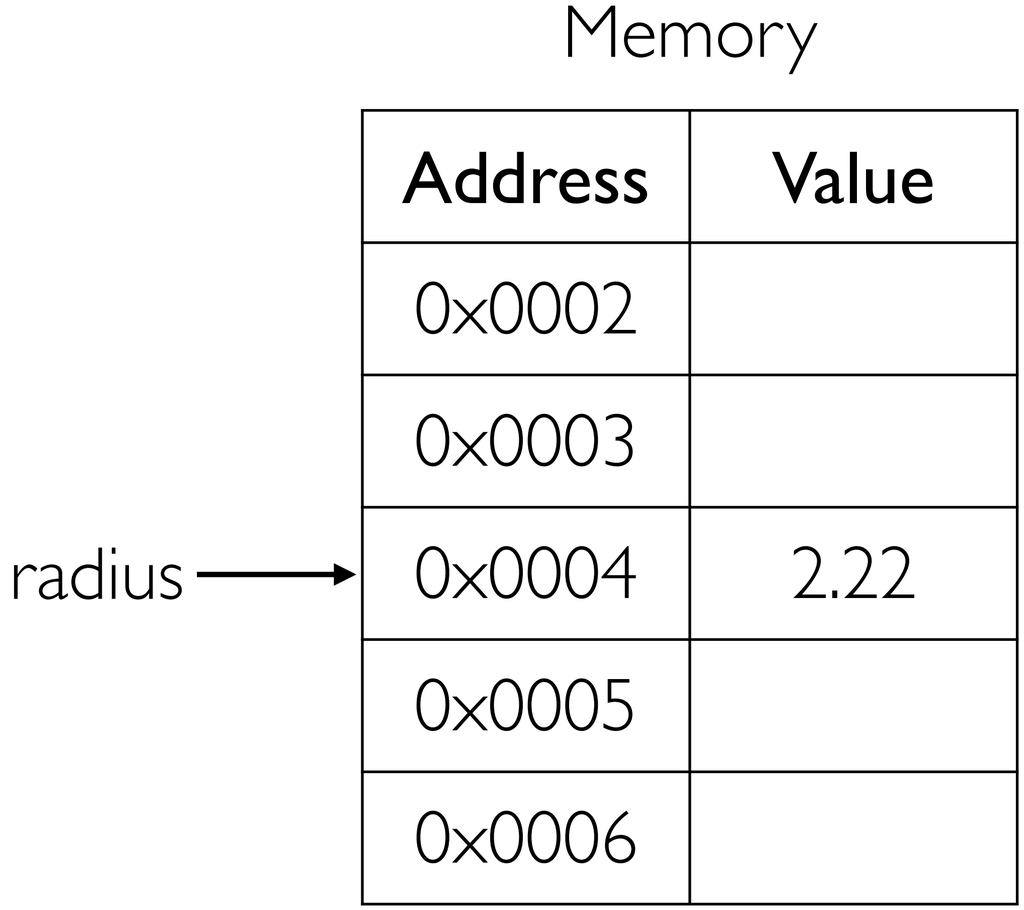
- Link between variable name and object
  - This link is called a ***reference***
  - An object may have multiple references
- Variables *point* to an object in memory



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

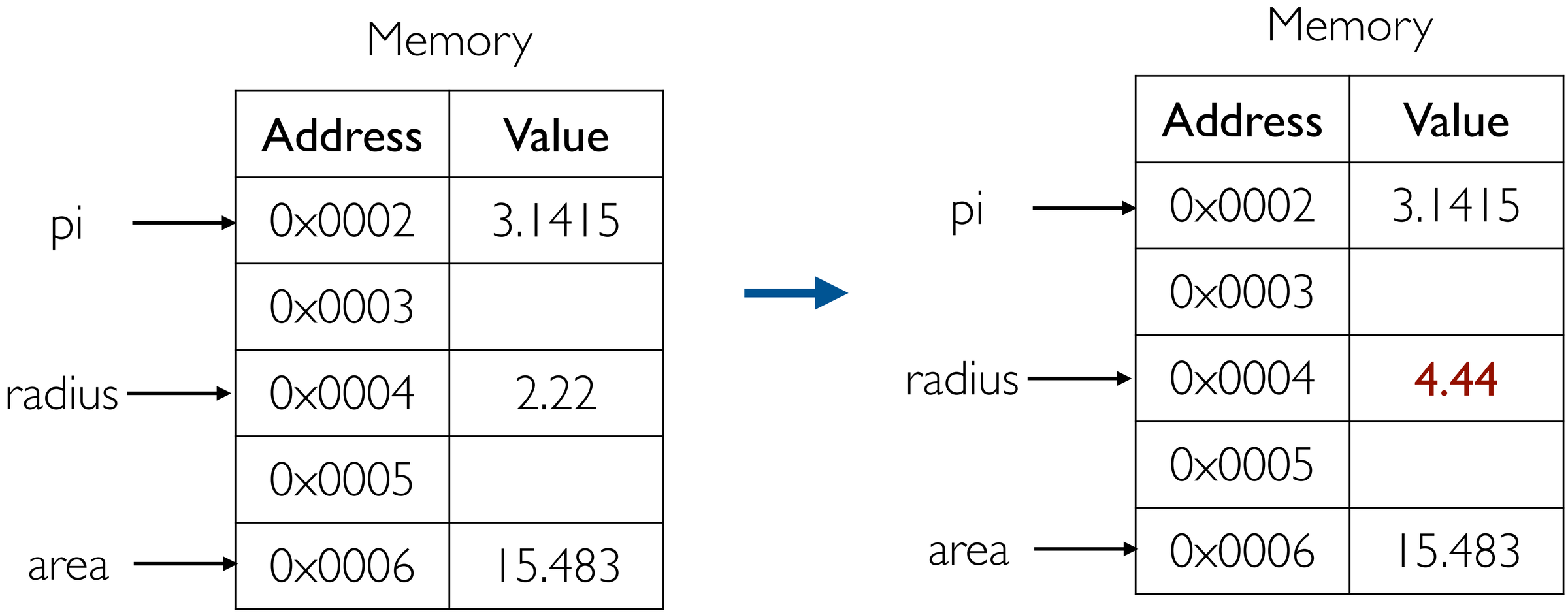
Re-binding variables

- **Values** of variables can be **changed**
- Location in memory may be preserved (or not)
- In a **high-level language** (e.g., Python), we don't need to think about location in memory



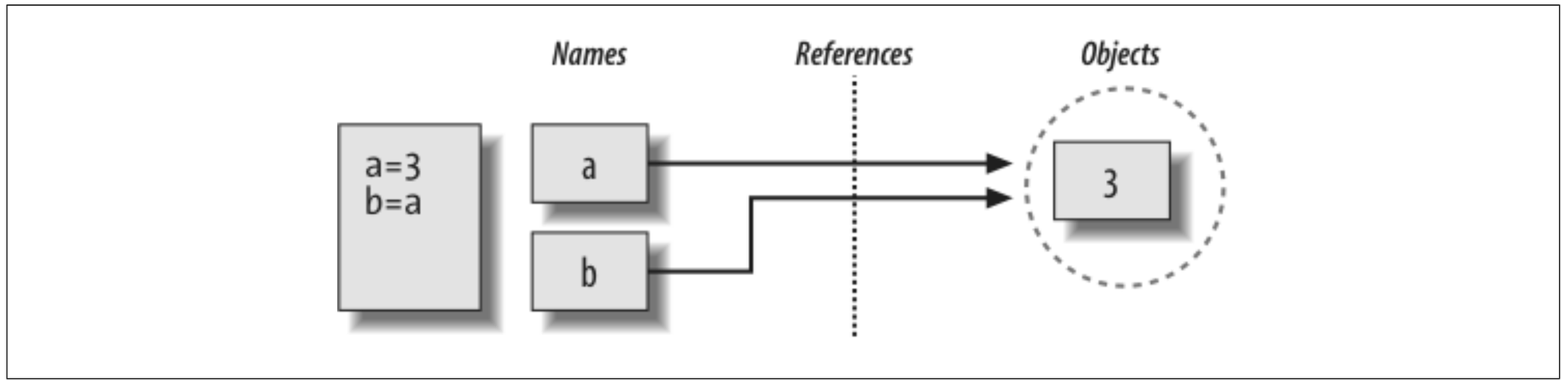
Re-binding variables (2)

```
>>>> area = pi * (radius**2)
>>>> radius = radius * 2
```



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"
```

Strings (text)

```
>>> year = 2021
```

Integer (number)

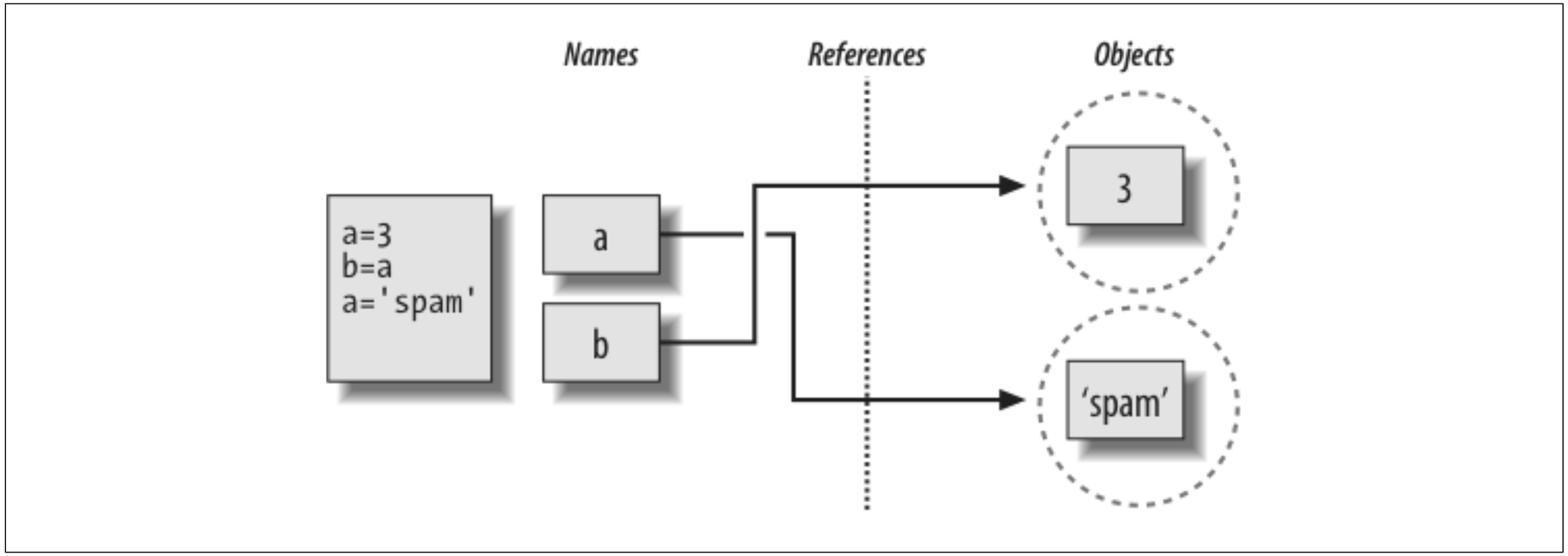
Types and operations

- 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`

Integers in Python

- Signed whole numbers (no decimal)
- Can be as large as needed (no size limit)

```
>>> x = 1

>>> y = 2021

>>> z = -123

>>> type(x)
int
```

Floats in Python

- Signed floating point (decimal) numbers
- Usually 64-bit precision (more on this later)

```
>>> x = 1.

>>> y = 3.1415

>>> z = -123.456

>>> type(x)
float
```

Operations on floats and ints

Code	Operation
`x + y`	Addition
`x - y`	Subtraction
`x * y`	Multiplication
`x / y`	Division
`x // y`	Floor division / Integer division
`x % y`	Modulo (remainder of devision)
`x ** y`	Exponentiation

Operations on floats and ints (2)

Code	Value
`3 + 2`	5
`3 - 2`	1
`3 * 2`	6
`3 / 2`	1.5
`3 // 2`	1
`3 % 2`	1
`3 ** 2`	9

Strings in Python

- Sequence of text characters
- Support operations such as **concatenation**

```
>>> string1 = "Hello"

>>> string2 = "world"

>>> string1 + " " + string2
Hello world

>>> string1 * 3
HelloHelloHello
```

String concatenation

String multiplication

Booleans in Python

- True or False
- Results of comparisons and logical expressions

```
>>>> 1 == 1
True

>>>> 1.11 < 2.22
True

>>>> "a" > "b"
False
```

Comparisons

Code	Operation
`x > y`	Greater than
`x >= y`	Greater than or equal to
`x < y`	Less than
`x <= y`	Less than or equal to
`x == y`	Equal
`x != y`	Not equal

Logical operators

- `A and B`
  - `True` if both `A` and `B` are `True`
- `A or B`
  - `True` if at least one of `A` or `B` is `True`
- `not A`
  - `True` if `A` is `False`
  - `False` if `A` is `True`

Truth table

A	B	A and B	A or B
True	True	True	True
True	False	False	True
False	True	False	True
False	False	False	False

Using boolean values

- Programming logic relies on booleans
- Use for control flow (more on this later)

```
>>> emission = get_CO_level()

>>> if emission > dangerous_limit:
...     shut_down()
... else:
...     keep_going()
```

None in Python

- Special value used to define null value
- Has its own data type (NoneType)
- NOT the same as:
  - 0 or 0.
  - False
  - Empty string
- Represents lack of a value

COLLECTION
DATA STRUCTURES

Collection data structures

- What are Python's collection data structures?
- Indexing and slicing
- Mutable vs. immutable objects

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

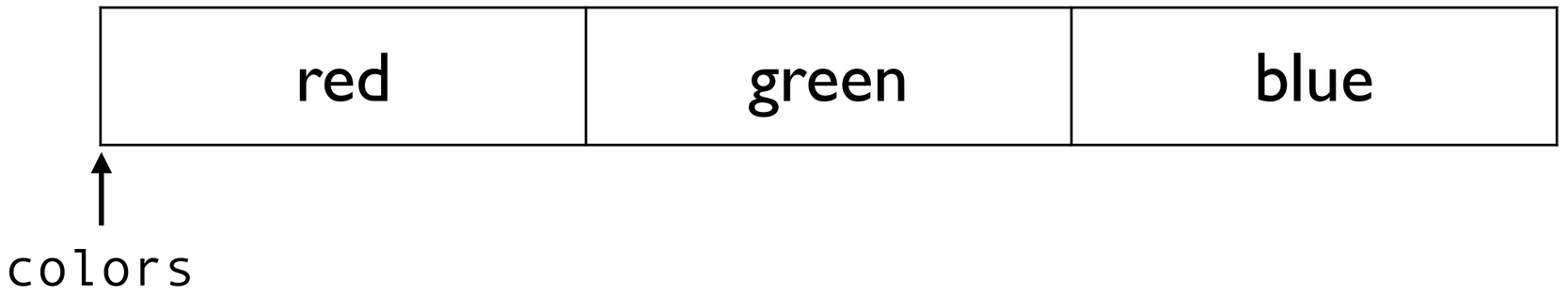
- Ordered collection of arbitrary objects
- Can be modified after creation
- Access elements by *offset*

`[]`	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

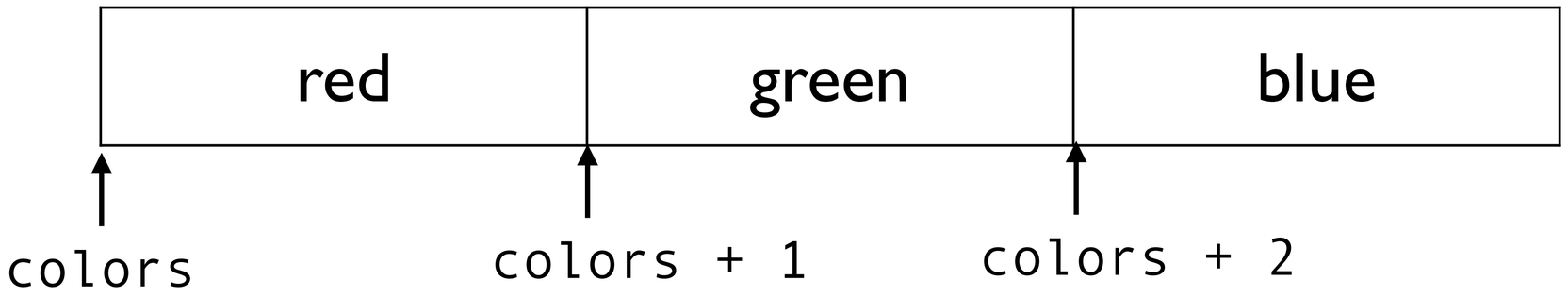
```
colors = ["red", "green", "blue"]
```



Accessing elements of a list

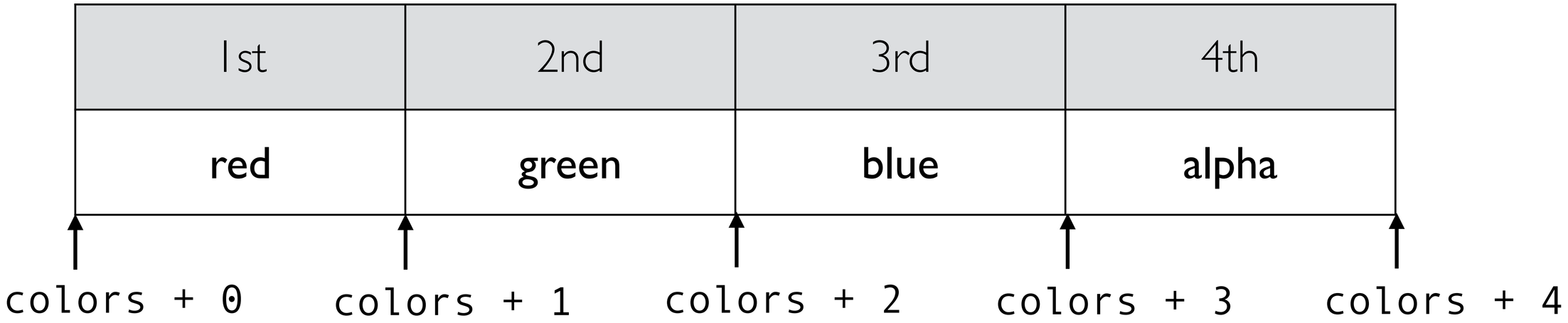
- Different languages access elements:
  - By ***offset*** from the beginning of the list (from 0)
  - By ***ordinal index*** of the element (from 1)
- Python accesses elements by **offset**

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



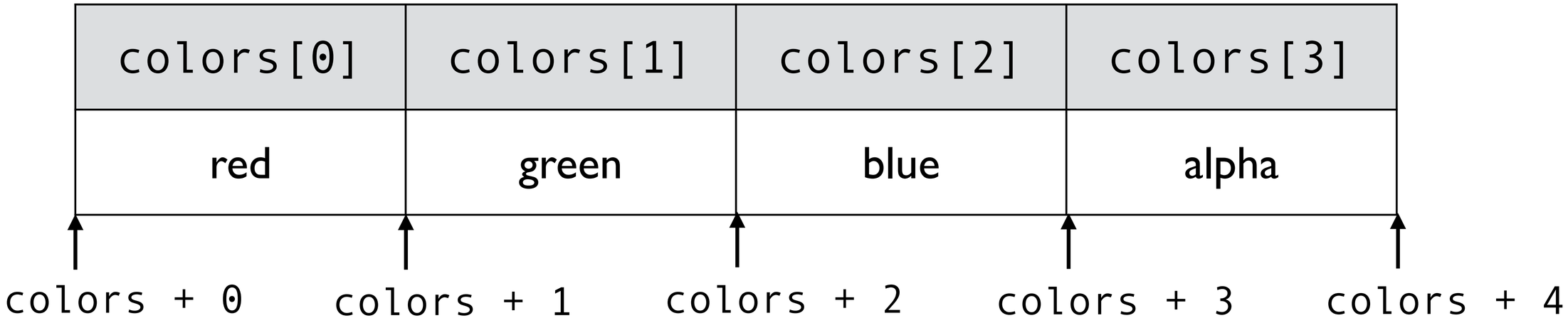
Offset vs ordinal indexing

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



Indexing in Python

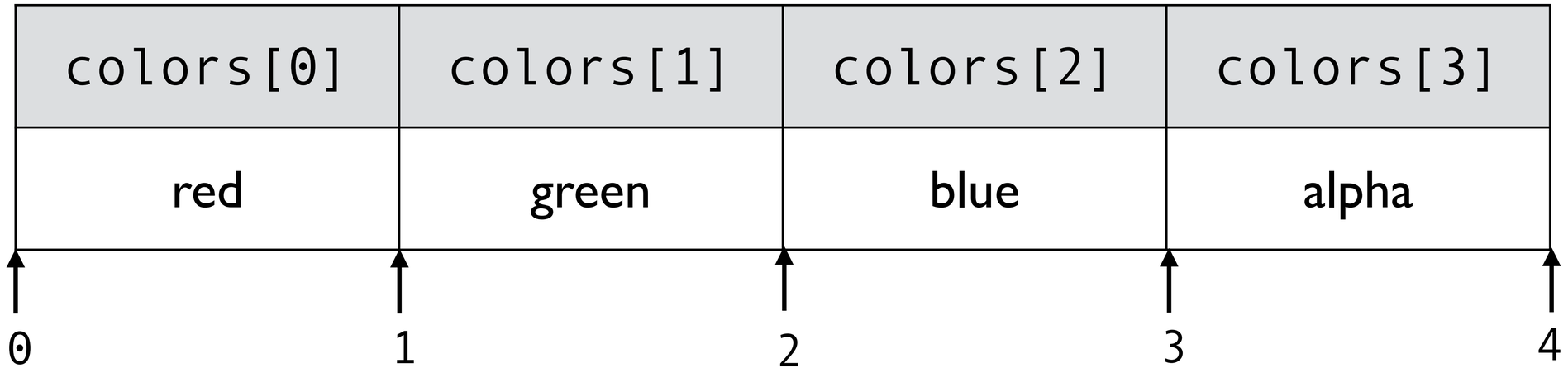
```
colors = ["red", "green", "blue", "alpha"]
```



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

Indexing in Python

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



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

Indexing (2)

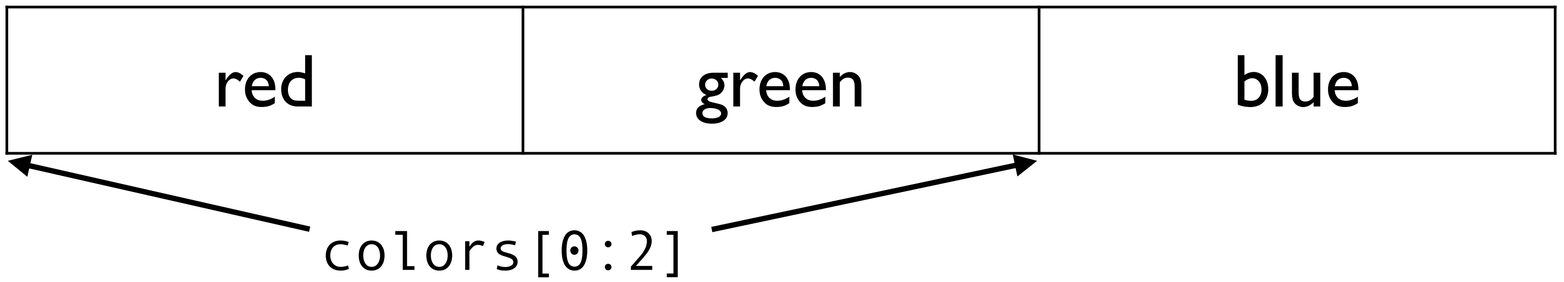
```
s = "Hello, world!"
```

Expression	Value
`s[0]`	"H"
`s[4]`	"o"
`s[-1]`	"!"
`s[-2]`	"d"
`len(s)`	13

Slicing in Python

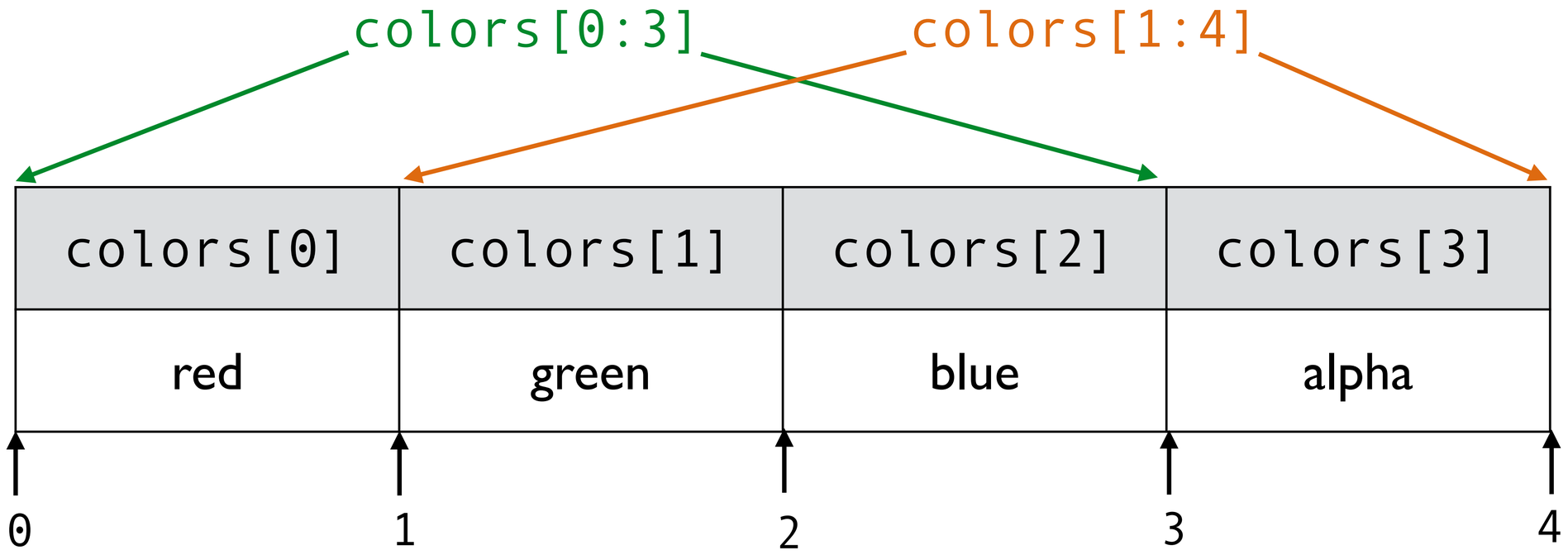
- **Slicing** is a powerful method of subsetting
- Access a subsequence of an ordered collection
- Slice a sequence using `start:end`

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



Slicing a list

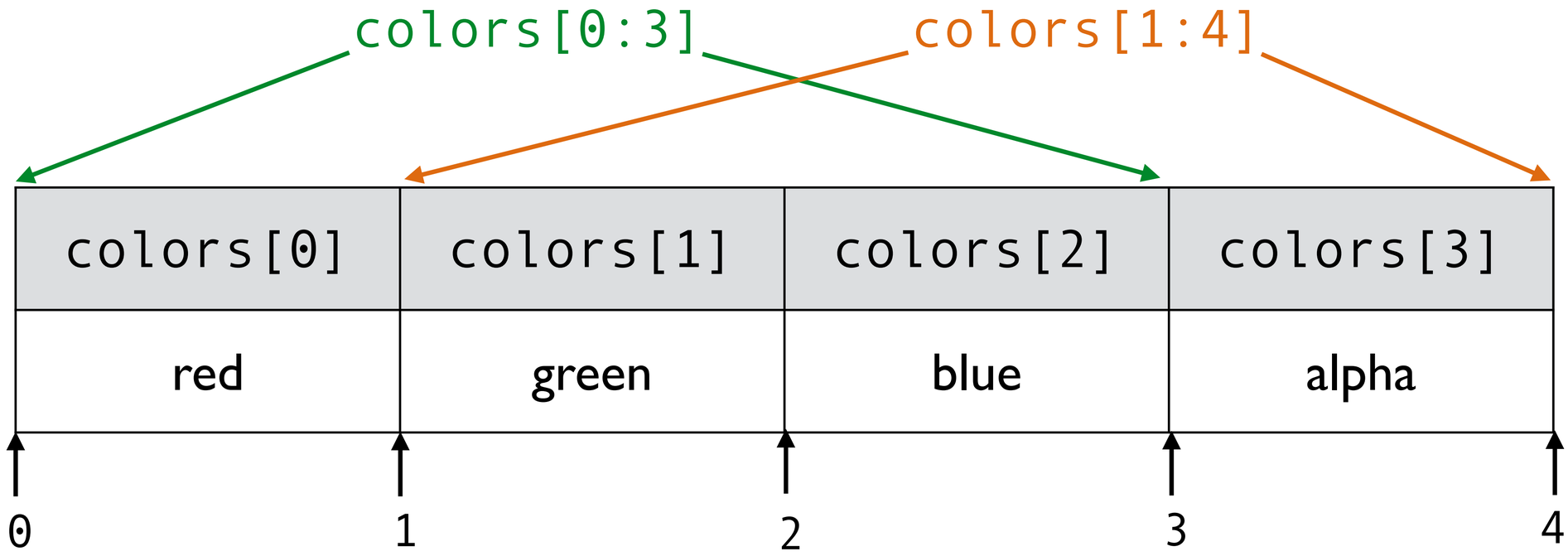
```
colors = ["red", "green", "blue", "alpha"]
```



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]

Slicing (2)

```
s = "Hello, world!"
```

Expression	Value
`s[0]`	"H"
`s[-1]`	"!"
`s[0:3]`	"Hel"
`s[6:]`	" world!"
`s[:-3]`	"Hello, wor"

Indexing and slicing

- Index and slice any ordered collection
  - Lists, tuples, strings
- Extracting subsets is a common operation
- Be careful of ***off-by-1*** errors!

Operations on lists

- Lists support some arithmetic operators
  - Concatenation
  - Multiplication
- Lists support *functions* and *methods*

```
>>> a = [1, 2, 3]
>>> b = [4, 5, 6]

>>> a * 2
[1, 2, 3, 1, 2, 3]

>>> a + b
[1, 2, 3, 4, 5, 6]
```

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!

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

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*

Lists vs. Tuples

- Both are ordered collections
- List
  - `["red", "blue", 1, 2]`
  - *Mutable* — can be modified after creation
- Tuple
  - `("red", "blue", 1, 2)`
  - *Immutable* — cannot be modified

Mutable vs. immutable

- Mutable object
  - Can be modified after creation
  - More memory-efficient
  - Use for data that changes
- Immutable object
  - 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[1] = 100
>>> a
[1, 100, 3]

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

Both references see changes

Methods and mutability

- Methods can modify the original object
- Sorting a list
  - `L.sort()` modifies the original list
  - `sorted(L)` returns a new list

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

CONTROL FLOW

Conditionals

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

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

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

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

Conditional example

- Control the flow of program logic
- Branch between different choices

```
emission = get_CO_level()

if emission > dangerous_limit:
    shut_down()
else:
    keep_going()
```

Indentation denotes blocks of statements!

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 <condition>:  # 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 <condition>:  # 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 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

FILES AND I/O

Reading a file

- Open a file for reading
  - `f = open("mydata.txt")`
- Read entire contents as a string
  - `content = f.read()`
- Read one line at a time
  - `line = f.readline()`
- Read all lines as a list of strings
  - `lines = f.readlines()`
- Always close the file when done
  - `f.close()`

Using the `with` pattern

- Always close files at end of operations
- But it's easy to forget to close a file
- Use `with` to close files automatically

```
with open("mydata.txt") as f:
    for line in f
        <statements>
```

File `f` is closed automatically at end of `with` block