SCALAR OBJECTS

- `int` – represent **integers**, ex. `5`
- `float` – represent **real numbers**, ex. `3.27`
- `bool` – represent **Boolean** values `True` and `False`
- `NoneType` – **special** and has one value, `None`
- can use `type()` to see the type of an object

```
>>> type(5)
int
>>> type(3.0)
float
```

what you write into the Python shell

what shows after hitting enter

TYPE CONVERSIONS (CAST)

- can **convert object of one type to another**
- `float(3)` converts integer `3` to float `3.0`
- `int(3.9)` truncates float `3.9` to integer `3`

OPERATORS ON ints and floats

- `i+j` → the **sum**
- `i-j` → the **difference**
- `i*j` → the **product**

if both are ints, result is int
if either or both are floats, result is float

- `i/j` → **division**

result is float

- `i%j` → the **remainder** when `i` is divided by `j`
- `i**j` → `i` to the **power** of `j`

STRINGS

- letters, special characters, spaces, digits
- enclose in **quotation marks or single quotes**

```
hi = "hello there"
```

- **concatenate** strings

```
name = "ana"

greet = hi + name

greeting = hi + " " + name
```

- do some **operations** on a string as defined in Python docs

```
silly = hi + " " + name * 3
```

INPUT/OUTPUT: `print`

- used to **output** stuff to console
- keyword is `print`

```
x = 1
print(x)
x_str = str(x)
print("my fav num is", x, ".", "x =", x)
print("my fav num is " + x_str + ". " + "x = " + x_str)
```

INPUT/OUTPUT: `input("")`

- prints whatever is in the quotes
- user types in something and hits enter
- binds that value to a variable

```
text = input("Type anything... ")
print(5*text)
```

- `input` **gives you a string** so must cast if working with numbers

```
num = int(input("Type a number... "))
print(5*num)
```

COMPARISON OPERATORS ON `int, float, string`

- `i` and `j` are variable names
- comparisons below evaluate to a Boolean

`i > j`

`i >= j`

`i < j`

`i <= j`

`i == j` → **equality** test, `True` if `i` is the same as `j`

`i != j` → **inequality** test, `True` if `i` not the same as `j`

LOGIC OPERATORS ON bools

- `a` and `b` are variable names (with Boolean values)

`not a` → `True` if `a` is `False`
`False` if `a` is `True`

`a and b` → `True` if both are `True`

`a or b` → `True` if either or both are `True`

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

COMPARISON EXAMPLE

```
pset_time = 15
sleep_time = 8
print(sleep_time > pset_time)
derive = True
drink = False
both = drink and derive
print(both)
```

CONTROL FLOW - BRANCHING

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

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

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

- `<condition>` has a value `True` or `False`
- evaluate expressions in that block if `<condition>` is `True`

INDENTATION

- matters in Python
- how you denote blocks of code

```
x = float(input("Enter a number for x: "))
y = float(input("Enter a number for y: "))
if x == y:
    print("x and y are equal")
    if y != 0:
        print("therefore, x / y is", x/y)
elif x < y:
    print("x is smaller")
else:
    print("y is smaller")
print("thanks!")
```

CONTROL FLOW: `while` LOOPS

```
while <condition>:
    <expression>
    <expression>
    ...
```

- `<condition>` evaluates to a Boolean
- if `<condition>` is `True`, do all the steps inside the while code block
- check `<condition>` again
- repeat until `<condition>` is `False`

`while` LOOP EXAMPLE

```
You are in the Lost Forest.
************
************
  ☺
************
************
Go left or right?
```

PROGRAM:

```
n = input("You're in the Lost Forest. Go left or right? ")
while n == "right":
    n = input("You're in the Lost Forest. Go left or right? ")
print("You got out of the Lost Forest!")
```

CONTROL FLOW: `while` and `for` LOOPS

- iterate through numbers in a sequence

```
# more complicated with while loop
n = 0
while n < 5:
    print(n)
    n = n+1


# shortcut with for loop
for n in range(5):
    print(n)
```

CONTROL FLOW: `for` LOOPS

```
for <variable> in range(<some_num>):
    <expression>
    <expression>
    ...
```

- each time through the loop, `<variable>` takes a value
- first time, `<variable>` starts at the smallest value
- next time, `<variable>` gets the prev value + 1
- etc.

`range(start,stop,step)`

- default values are `start = 0` and `step = 1` and optional
- loop until value is `stop - 1`

```
mysum = 0
for i in range(7, 10):
    mysum += i
print(mysum)
```

```
mysum = 0
for i in range(5, 11, 2):
    mysum += i
print(mysum)
```

`break` STATEMENT

- immediately exits whatever loop it is in
- skips remaining expressions in code block
- exits only innermost loop!

```
while <condition_1>:
    while <condition_2>:
        <expression_a>
        break
        <expression_b>
    <expression_c>
```

`break` STATEMENT

```
mysum = 0
for i in range(5, 11, 2):
    mysum += i
    if mysum == 5:
        break
        mysum += 1
print(mysum)
```

- what happens in this program?

`for` VS `while` LOOPS

`for` loops

- **know** number of iterations
- can **end early** via `break`
- uses a **counter**
- **can rewrite** a `for` loop using a `while` loop

`while` loops

- **unbounded** number of iterations
- can **end early** via `break`
- can use a **counter but must initialize** before loop and increment it inside loop
- **may not be able to rewrite** a `while` loop using a `for` loop

STRINGS

- think of as a **sequence** of case sensitive characters
- can compare strings with ==, >, < etc.
- `len()` is a function used to retrieve the **length** of the string in the parentheses

```
s = "abc"
```

`len(s)` → evaluates to 3

STRINGS

- square brackets used to perform **indexing** into a string to get the value at a certain index/position

```
s = "abc"
```

index: 0 1 2 ← indexing always starts at 0

index: -3 -2 -1 ← last element always at index -1

```
s[0]   → evaluates to "a"
s[1]   → evaluates to "b"
s[2]   → evaluates to "c"
s[3]   → trying to index out of bounds, error
s[-1]  → evaluates to "c"
s[-2]  → evaluates to "b"
s[-3]  → evaluates to "a"
```

STRINGS

- can **slice** strings using `[start:stop:step]`
- if give two numbers, `[start:stop]`, `step=1` by default
- you can also omit numbers and leave just colons

If unsure what some command does, try it out in your console!

`s = "abcdefgh"`

`s[3:6]` → evaluates to `"def"`, same as `s[3:6:1]`

`s[3:6:2]` → evaluates to `"df"`

`s[::]` → evaluates to `"abcdefgh"`, same as `s[0:len(s):1]`

`s[::-1]` → evaluates to `"hgfedbca"`, same as `s[-1:-(len(s)+1):-1]`

`s[4:1:-2]` → evaluates to `"ec"`

STRINGS

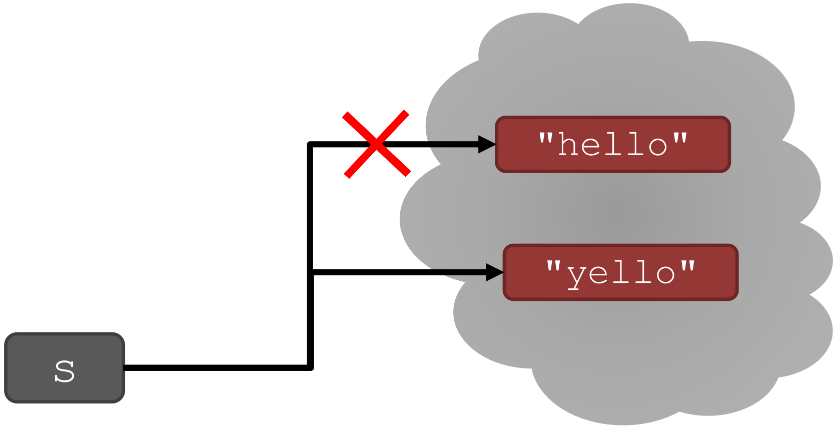
- strings are "**immutable**" – cannot be modified

```
s = "hello"
```

```
s[0] = 'y'
s = 'y'+s[1:len(s)]
```

→ gives an error

→ is allowed, s bound to new object



for LOOPS RECAP

- `for` loops have a **loop variable** that iterates over a set of values

```
for var in range(4):
    <expressions>
```

→ `var` iterates over values 0,1,2,3

→ expressions inside loop executed with each value for `var`

```
for var in range(4,6):
    <expressions>
```

→ `var` iterates over values 4,5

- `range` is a way to iterate over numbers, but a for loop variable can **iterate over any set of values**, not just numbers!

STRINGS AND LOOPS

- these two code snippets do the same thing
- bottom one is more “pythonic”

```
s = "abcdefgh"
for index in range(len(s)):
    if s[index] == 'i' or s[index] == 'u':
        print("There is an i or u")

for char in s:
    if char == 'i' or char == 'u':
        print("There is an i or u")
```

CODE EXAMPLE:
ROBOT CHEERLEADERS

```
an_letters = "aefhilmnorsxAEFHILMNORSX"

word = input("I will cheer for you! Enter a word: ")
times = int(input("Enthusiasm level (1-10): "))
```

```
i = 0
while i < len(word):
    char = word[i]
```

✗

```
for char in word:
```

✓

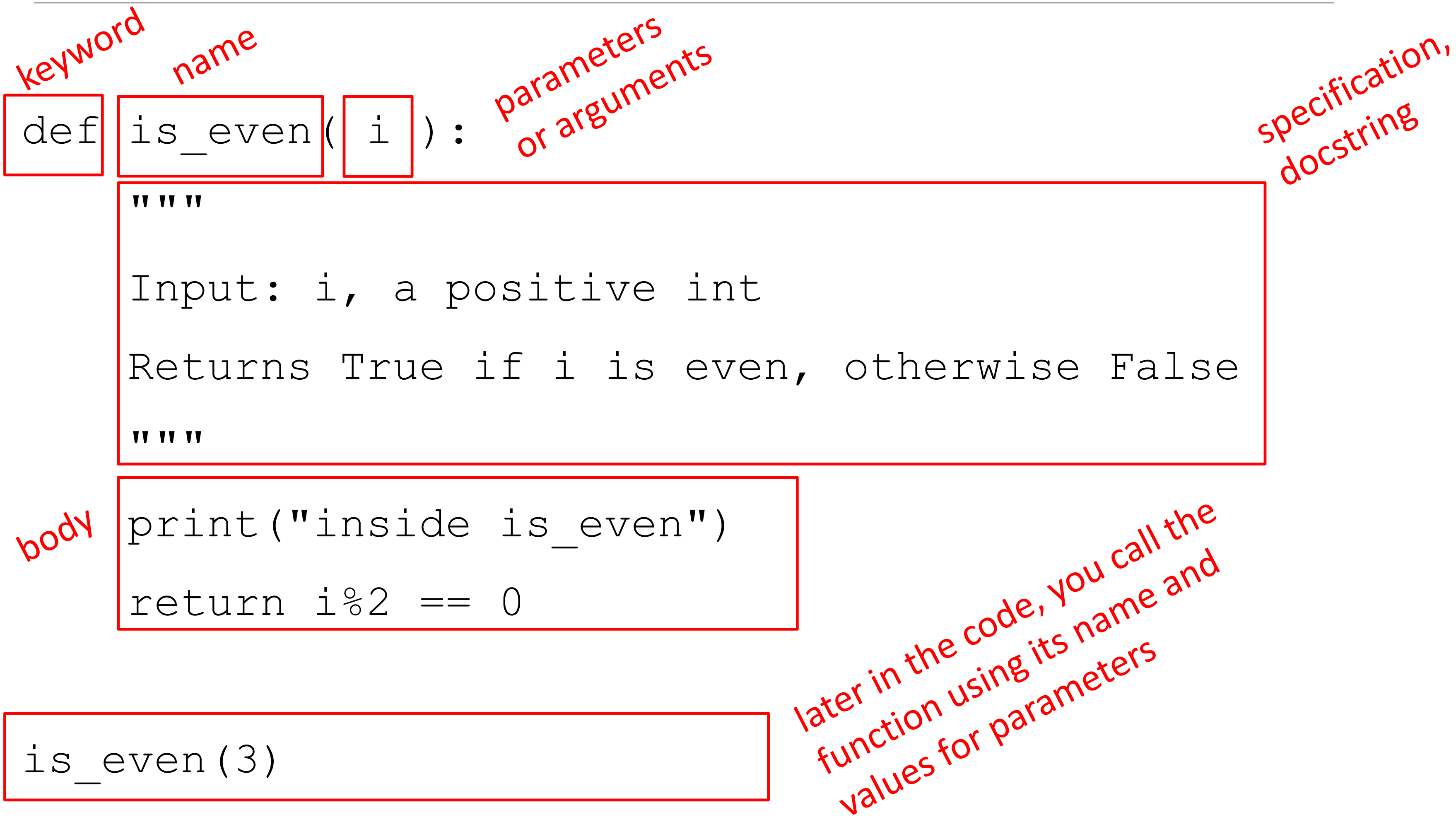
```
    if char in an_letters:
        print("Give me an " + char + "! " + char)
    else:
        print("Give me a  " + char + "! " + char)
```

✗

```
    i += 1
```

```
print("What does that spell?")
for i in range(times):
    print(word, "!!!")
```

HOW TO WRITE and CALL/INVOKE A FUNCTION



```
def is_even( i ):
    """
    Input: i, a positive int
    Returns True if i is even, otherwise False
    """
    print("inside is_even")
    return i%2 == 0

is_even(3)
```

keyword

name

parameters or arguments

specification, docstring

body

later in the code, you call the function using its name and values for parameters

IN THE FUNCTION BODY

```
def is_even( i ):
    """
    Input: i, a positive int
    Returns True if i is even, otherwise False
    """
    print("inside is_even")
    return i%2 == 0
```

run some commands

keyword

expression to evaluate and return

ONE WARNING IF NO `return` STATEMENT

```
def is_even( i ):
    """
    Input: i, a positive int
    Does not return anything
    """
    i%2 == 0
```

without a return statement

- Python returns the value **None, if no `return` given**
- represents the absence of a value

TUPLES

- an ordered sequence of elements, can mix element types
- cannot change element values, **immutable**
- represented with parentheses

remember strings?

```
te = ()
```

empty tuple

```
t = (2,"mit",3)
t[0]                    → evaluates to 2
(2,"mit",3) + (5,6)     → evaluates to (2,"mit",3,5,6)
t[1:2]      → slice tuple, evaluates to ("mit",)
t[1:3]      → slice tuple, evaluates to ("mit",3)
len(t)      → evaluates to 3
t[1] = 4    → gives error, can't modify object
```

extra comma means a tuple with one element

TUPLES

- conveniently used to **swap** variable values

```
x = y
y = x
```

✗

```
temp = x
x = y
y = temp
```

✓

```
(x, y) = (y, x)
```

✓

- used to **return more than one value** from a function

```
def quotient_and_remainder(x, y):
    q = x // y
    r = x % y
    return (q, r)
(quot, rem) = quotient_and_remainder(4,5)
```

integer division

LISTS

- **ordered sequence** of information, accessible by index
- a list is denoted by **square brackets**, `[]`
- a list contains **elements**
  - usually homogeneous (ie, all integers)
  - can contain mixed types (not common)
- list elements can be changed so a list is **mutable**

INDICES AND ORDERING

`a_list = []` empty list

`L = [2, 'a', 4, [1,2]]`

`len(L)` → evaluates to 4

`L[0]` → evaluates to 2

`L[2]+1` → evaluates to 5

`L[3]` → evaluates to `[1,2]`, another list!

`L[4]` → gives an error

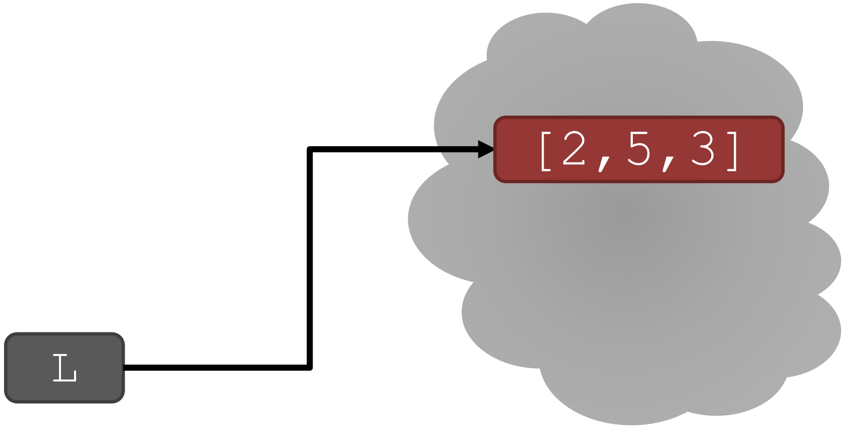
`i = 2`
`L[i-1]` → evaluates to ‘a’ since `L[1]='a'` above

CHANGING ELEMENTS

- lists are **mutable**!
- assigning to an element at an index changes the value

```
L = [2, 1, 3]
L[1] = 5
```

- `L` is now `[2, 5, 3]`, note this is the **same object** `L`



ITERATING OVER A LIST

- compute the **sum of elements** of a list
- common pattern, iterate over list elements

```
total = 0
  for i in range(len(L)):
      total += L[i]
  print total
```

```
total = 0
  for i in L:
      total += i
  print total
```

like strings, can iterate over list elements directly

- notice
  - list elements are indexed `0` to `len(L)-1`
  - `range(n)` goes from `0` to `n-1`

OPERATIONS ON LISTS - ADD

- **add** elements to end of list with `L.append(element)`
- **mutates** the list!

  `L = [2,1,3]`

  `L.append(5)` → L is now `[2,1,3,5]`

- what is the dot?
  - lists are Python objects, everything in Python is an object
  - objects have data
  - objects have methods and functions
  - access this information by `object_name.do_something()`
  - will learn more about these later

OPERATIONS ON LISTS - ADD

- to combine lists together use **concatenation**, + operator, to give you a new list
- **mutate** list with `L.extend(some_list)`

`L1 = [2,1,3]`

`L2 = [4,5,6]`

`L3 = L1 + L2` → `L3` is `[2,1,3,4,5,6]`
`L1, L2` unchanged

`L1.extend([0,6])` → mutated `L1` to `[2,1,3,0,6]`

OPERATIONS ON LISTS - REMOVE

- delete element at a **specific index** with `del(L[index])`
- remove element at **end of list** with `L.pop()`, returns the removed element
- remove a **specific element** with `L.remove(element)`
  - looks for the element and removes it
  - if element occurs multiple times, removes first occurrence
  - if element not in list, gives an error

```
L = [2,1,3,6,3,7,0] # do below in order
L.remove(2) → mutates L = [1,3,6,3,7,0]
L.remove(3) → mutates L = [1,6,3,7,0]
del(L[1])   → mutates L = [1,3,7,0]
L.pop()     → returns 0 and mutates L = [1,3,7]
```

all these operations mutate the list

CONVERT LISTS TO STRINGS AND BACK

- convert **string to list** with `list(s)`, returns a list with every character from `s` an element in `L`
- can use `s.split()`, to **split a string on a character** parameter, splits on spaces if called without a parameter
- use `''.join(L)` to turn a **list of characters into a string**, can give a character in quotes to add char between every element

`s = "I<3 cs"` → `s` is a string

`list(s)` → returns `['I','<','3',' ','c','s']`

`s.split('<')` → returns `['I', '3 cs']`

`L = ['a','b','c']` → `L` is a list

`''.join(L)` → returns `"abc"`

`'_'.join(L)` → returns `"a_b_c"`

OTHER LIST OPERATIONS

- `sort()` and `sorted()`
- `reverse()`
- and many more!
https://docs.python.org/3/tutorial/datastructures.html

`L=[9,6,0,3]`

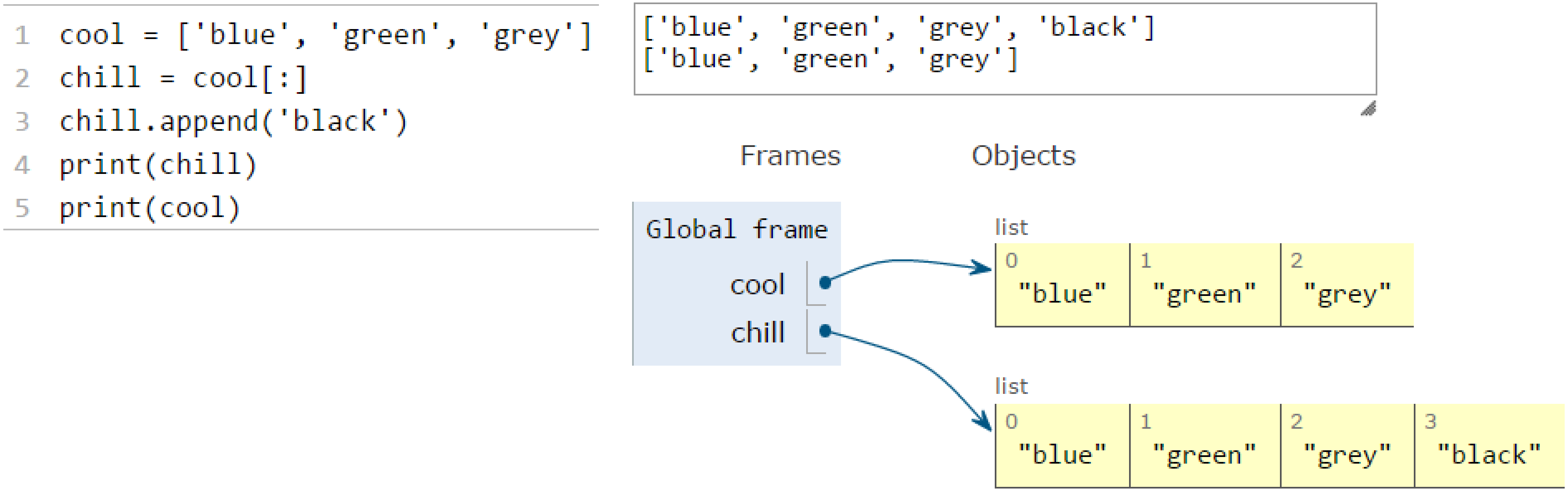
`sorted(L)` → returns sorted list, does **not mutate** `L`

`L.sort()` → **mutates** `L=[0,3,6,9]`

`L.reverse()` → **mutates** `L=[9,6,3,0]`

CLONING A LIST

- create a new list and **copy every element** using `chill = cool[:]`

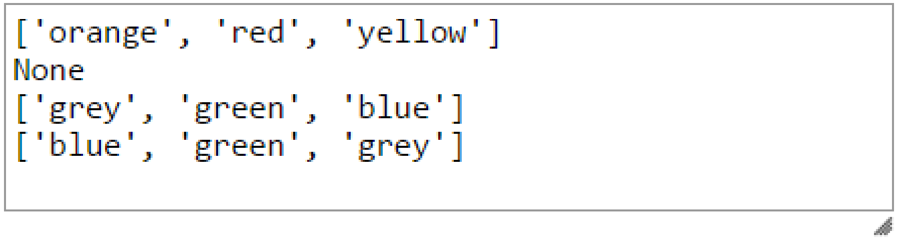


```
cool = ['blue', 'green', 'grey']
chill = cool[:]
chill.append('black')
print(chill)
print(cool)
```

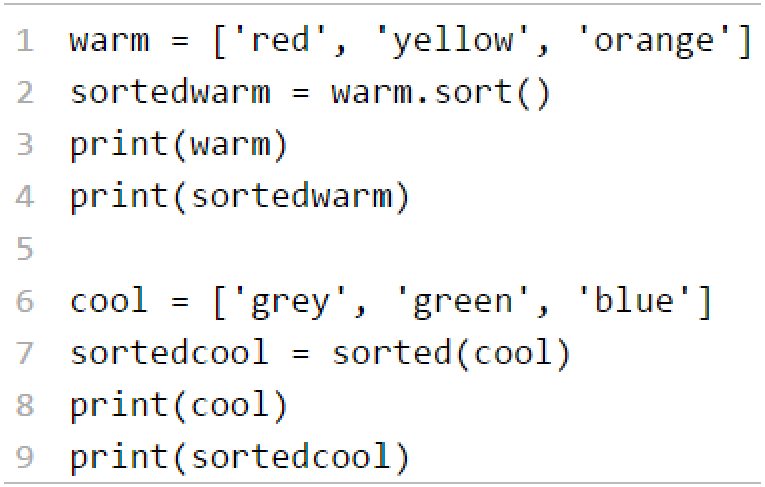
```
['blue', 'green', 'grey', 'black']
['blue', 'green', 'grey']
```

SORTING LISTS

- calling `sort()` **mutates** the list, returns nothing
- calling `sorted()` **does not mutate** list, must assign result to a variable

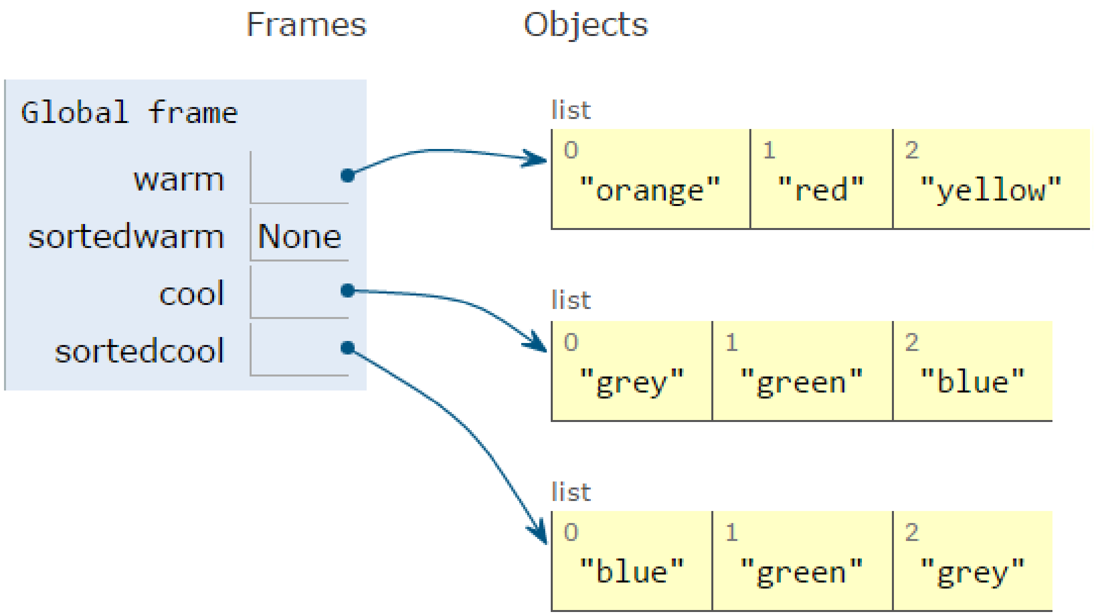


```
['orange', 'red', 'yellow']
None
['grey', 'green', 'blue']
['blue', 'green', 'grey']
```



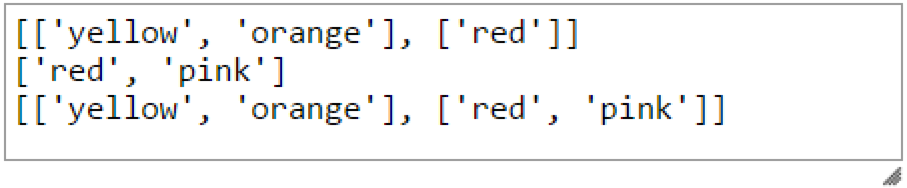
```
warm = ['red', 'yellow', 'orange']
sortedwarm = warm.sort()
print(warm)
print(sortedwarm)

cool = ['grey', 'green', 'blue']
sortedcool = sorted(cool)
print(cool)
print(sortedcool)
```

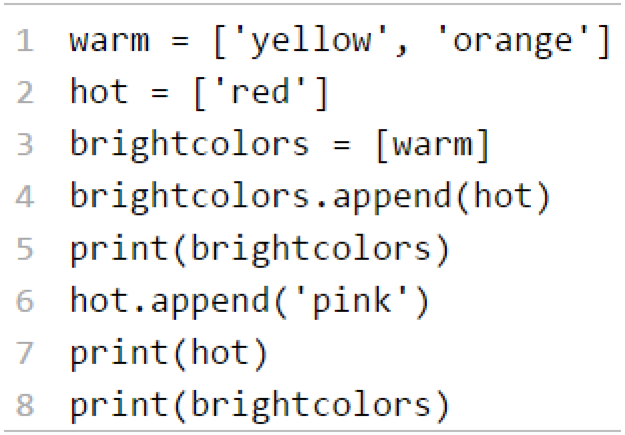


LISTS OF LISTS OF LISTS OF….

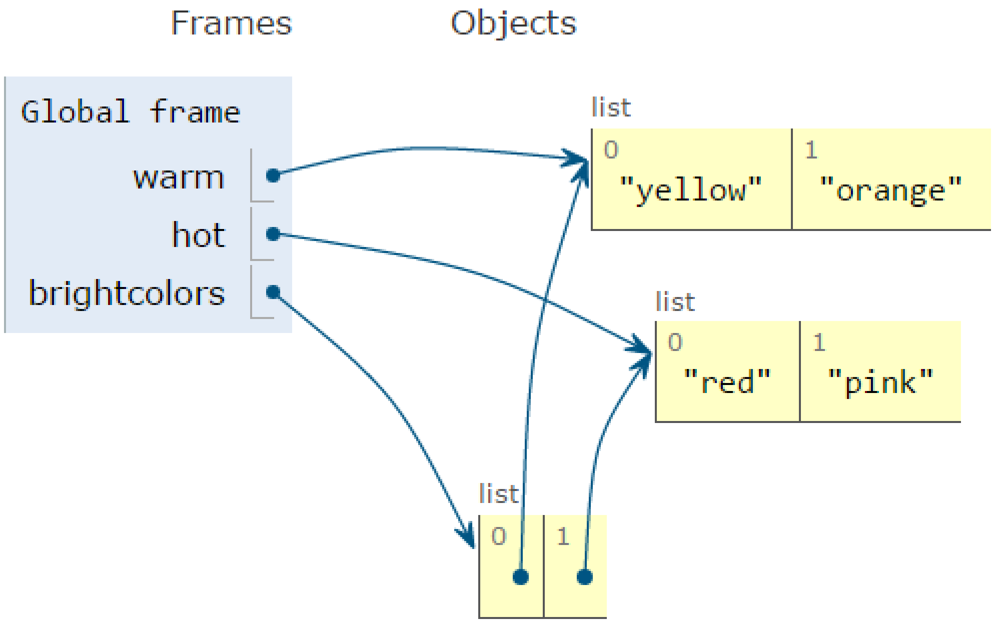
- can have **nested** lists
- side effects still possible after mutation



```
[['yellow', 'orange'], ['red']]
['red', 'pink']
[['yellow', 'orange'], ['red', 'pink']]
```



```
warm = ['yellow', 'orange']
hot = ['red']
brightcolors = [warm]
brightcolors.append(hot)
print(brightcolors)
hot.append('pink')
print(hot)
print(brightcolors)
```



MUTATION AND ITERATION
Try this in Python Tutor!

- **avoid** mutating a list as you are iterating over it

✗

```
def remove_dups(L1, L2):
    for e in L1:
        if e in L2:
            L1.remove(e)
```

✓

```
def remove_dups(L1, L2):
    L1_copy = L1[:]
    for e in L1_copy:
        if e in L2:
            L1.remove(e)
```

clone list first, note that `L1_copy = L1` does NOT clone

```
L1 = [1, 2, 3, 4]
L2 = [1, 2, 5, 6]
remove_dups(L1, L2)
```

- `L1` is `[2,3,4]` not `[3,4]` Why?
  - Python uses an internal counter to keep track of index it is in the loop
  - mutating changes the list length but Python doesn't update the counter
  - loop never sees element 2