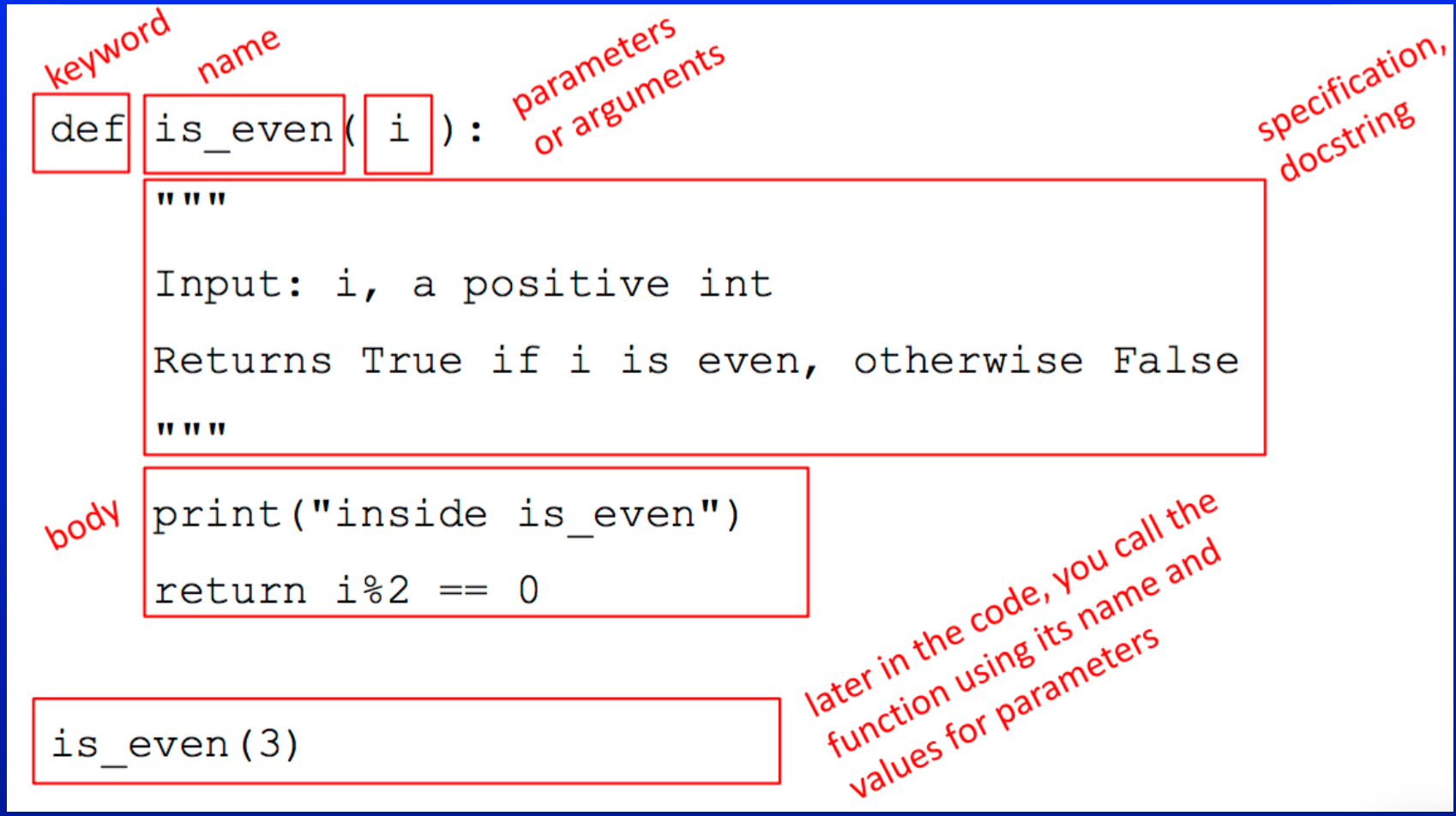
Python: Functions

Functions

- write reusable pieces/chunks of code, called **functions**
- functions are not run in a program until they are "**called**" or "**invoked**" in a program
- function characteristics:
  - has a **name**
  - has **parameters** (0 or more)
  - has a **docstring** (optional but recommended)
  - has a **body**
  - **returns** something

Source:https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-0001-introduction-to-computer-science-and-programming-in-python-fall-2016/lecture-slides-code/

Defining and invoking a function



Source:https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-0001-introduction-to-computer-science-and-programming-in-python-fall-2016/lecture-slides-code/

Defining and invoking 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
```

run some commands

keyword

expression to evaluate and return

Source:https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-0001-introduction-to-computer-science-and-programming-in-python-fall-2016/lecture-slides-code/

Defining and invoking a function

Consider $f(x) = x^2$

```
def square(x):          #defining function
  return   x*x


square(4)          #invoking function

16                 # output
```

Defining and invoking a function

Example: Functions may not have arguments, and return statement

```
def myprint():          #defining function
  print ("Hello world")

myprint()          #invoking function

Hello world        # output
```

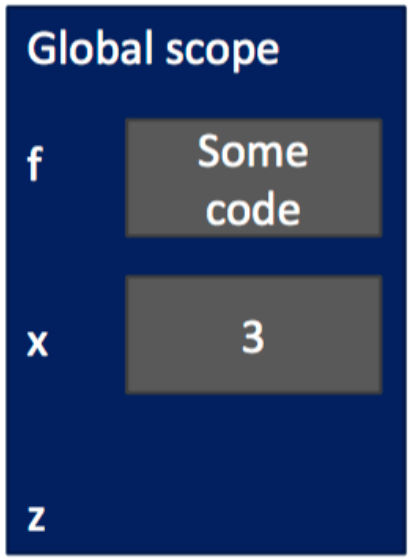
Defining and invoking a function

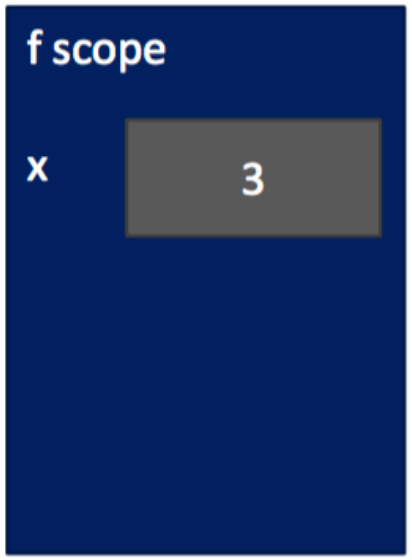
Example: Function calling another function

```
def repeatmyprint():
     myprint()
     myprint()

repeatmyprint()      #invoking function

Hello world     # output
Hello world
```

Scope of a Variable

```
def f( x ):
    x = x + 1
    print('in f(x): x =', x)
    return x

x = 3
z = f( x )
```

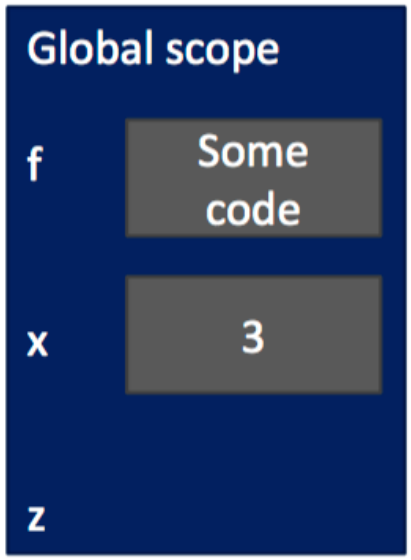


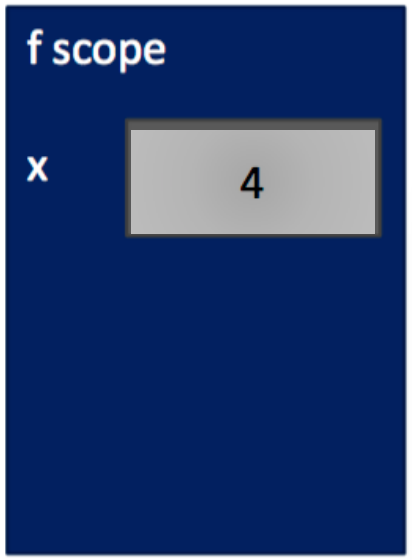


Source:https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-0001-introduction-to-computer-science-and-programming-in-python-fall-2016/lecture-slides-code/

Scope of a Variable

```
def f( x ):
    x = x + 1
    print('in f(x): x =', x)
    return x

x = 3
z = f( x )
```

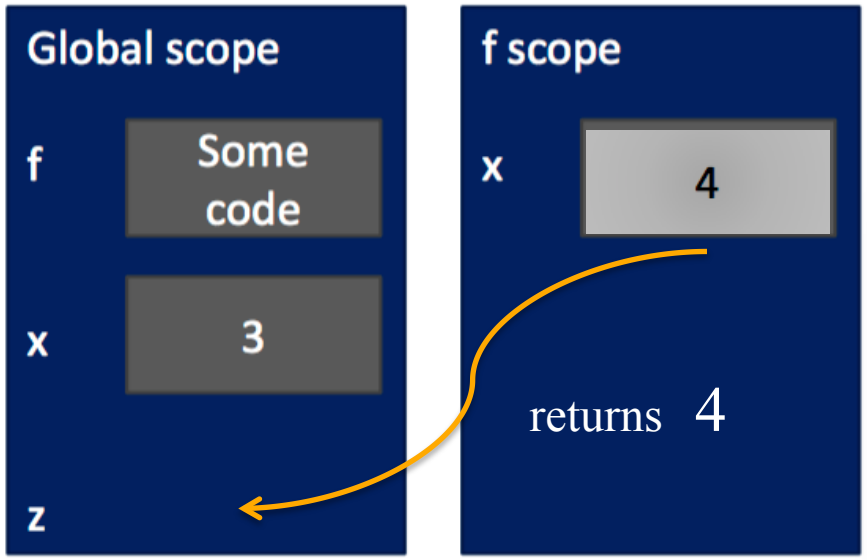




Source:https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-0001-introduction-to-computer-science-and-programming-in-python-fall-2016/lecture-slides-code/

Scope of a Variable

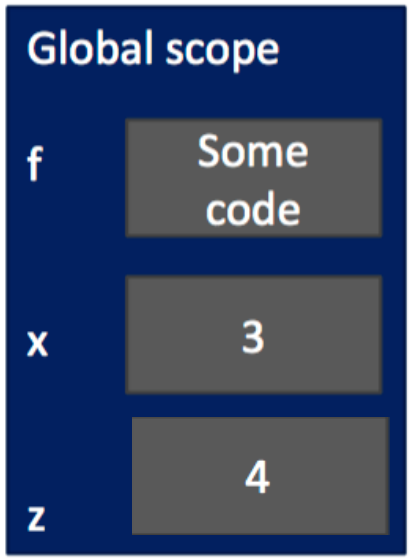
```
def f( x ):
    x = x + 1
    print('in f(x): x =', x)
    return x

x = 3
z = f( x )
```



Source:https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-0001-introduction-to-computer-science-and-programming-in-python-fall-2016/lecture-slides-code/

Scope of a Variable

```
def f( x ):
    x = x + 1
    print('in f(x): x =', x)
    return x

x = 3
z = f( x )
```



Source:https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-0001-introduction-to-computer-science-and-programming-in-python-fall-2016/lecture-slides-code/

Function: Scope

```
def f(y):
    x=1
    x+=1
    print(x)
```

```
x=5
f(x)
print(x)
```

x is redefined locally

Output

2

5

Source:https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-0001-introduction-to-computer-science-and-programming-in-python-fall-2016/lecture-slides-code/

Function: Scope (Example)

```
def g(x):
    def h():
        x = 'abc'
    x = x + 1
    print('g: x =', x)
    h()
    return x


x = 3
z = g(x)
print (z)
```

Output

```
g:x=4
4
```

Source:https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-0001-introduction-to-computer-science-and-programming-in-python-fall-2016/lecture-slides-code/

Python: Recursive Functions

Recursive Functions

Recall factorial function:

```
def factorial(n):
    i=0
    fact=1
    while (i<n):
        i=i+1
        fact=fact*i
    return fact
```

Iterative Algorithm

Loop construct (while)

can capture computation in a set of **state variables** that update on each iteration through loop

Recursive Functions

Alternatively:

Consider

5! = 5x4x3x2x1
can be re-written as 5!=5x4!

In general n! = nx(n-1)!

factorial(n) = n * factorial(n-1)

Recursive Functions

Alternatively:

Consider

5! = 5x4x3x2x1
can be re-written as 5!=5x4!

In general n! = nx(n-1)!

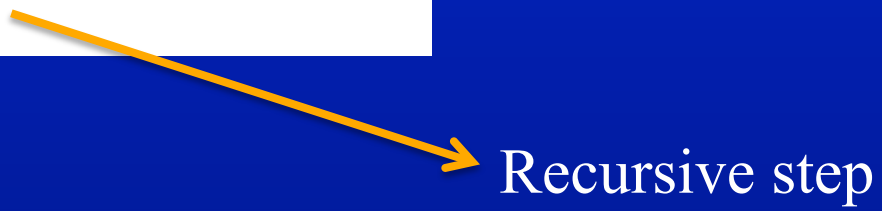
Recursive Algorithm

function calling itself

factorial(n) = n * factorial(n-1)

Recursive Functions

```
def factorial( n):
    if (n == 0):
        return 1
    else:
        return n * factorial(n – 1)
```



GCD Algorithm

$$\gcd(a, b) = \begin{cases} b & \text{if } a \bmod b = 0 \\ \gcd(b, a \bmod b) & \text{otherwise} \end{cases}$$

```
def gcd(a,b):
 r=a%b
 while (r !=0 ):
   a=b
   b=r
   r=a%b
 return b
```

Iterative Algorithm

```
def gcd(a,b):
   if (a%b == 0):
     return b
   else:
     return gcd(b,a%b)
```

Recursive Algorithm

GCD Algorithm

$$\gcd(a, b) = \begin{cases} b & \text{if } a \bmod b = 0 \\ \gcd(b, a \bmod b) & \text{otherwise} \end{cases}$$

```
def gcd(a,b):
    if (a%b == 0):
        return b
    else:
        return gcd(b,a%b)
```

```
gcd (6, 10)
  gcd (10, 6)
    gcd (6, 4)
      gcd (4, 2)
       2
     2
   2
2
```

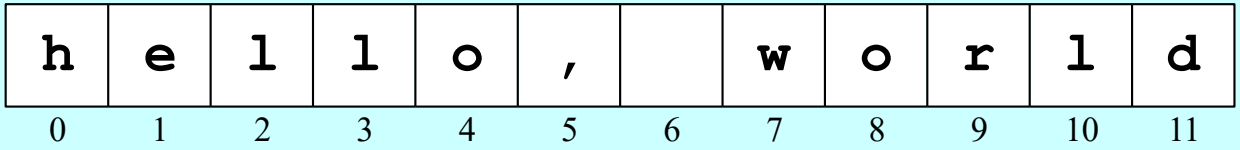
Courtesy Prof PR Panda CSE Department IIT Dehi

Selecting Characters from a String

- A string is (still) an ordered collection of characters. The character positions in a Python string are, as in most computer languages, identified by an ***index*** beginning at 0.
- For example, if `s` is initialized as

```
s = "hello, world"
```

  the characters in `s` are arranged like this:



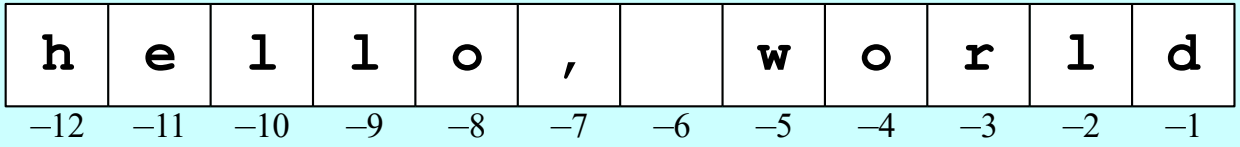
- You can select an individual character using the syntax *str*`[`*k*`]`, where *k* is the index of the desired character. The expression

```
s[7]
```

  returns the one-character string `"w"` that appears at index 7.

Negative Indexing

- Unlike JavaScript, Python allows you to specify a character position in a string by using negative index numbers, which count backwards from the end of the string. The characters in the `"hello, world"` string on the previous slide can therefore be numbered using the following indices:



- You can select the `"w"` toward the end of this string using the expression

```
s[-5]
```

  which is shorthand for the positive indexing expression

```
s[len(s) - 5]
```

Concatenation

- One of the more familiar operations available to Python strings is ***concatenation***, which consists of combining two strings end to end with no intervening characters.
- Concatenation is built into Python in the form of the **+** operator. This is consistent with how JavaScript and most other languages support concatenation.
- Noteworthy difference between Python and JavaScript: Python interprets the **+** operator as concatenation only if **both** operands are strings. If one of the operands is something other than a string, then string concatenation isn't applied. Restated, Python doesn't automatically convert numbers to strings as JavaScript does.

Repetition

- In much the same way that Python redefines the `+` operator to indicate string concatenation, it also redefines the `*` operator for strings to indicate repetition, so that the expression `s * n` indicates `n` copies of the string `s` concatenated together.
- The expression `"la" * 3` therefore returns `"lalala"`, which is three copies of the string `"la"` concatenated together.
- Note that this interpretation is consistent with the idea that multiplication is repeated addition:

```
"la" * 3  →  "la" + "la" + "la"
```

- You can use this feature, for example, to print a line of 80 hyphens like this:

```
print("-" * 80)
```

Slicing

- Python allows you to extract a substring by specifying a range of index positions inside the square brackets. This operation is known as ***slicing***.
- The simplest specification of a slice is `[`*start*`:`*stop*`]`, where *start* is the index at which the slice begins, and *stop* is the past-the-end index where the slice ends.
- The *start* and *stop* components of a slice are optional, but the colon must be present. If *start* is missing, it defaults to 0, and if *stop* is missing, it defaults to the length of the string.
- A slice specification may also contain a third component called a ***stride***, as with `[`*start*`:`*stop*`:`*stride*`]`. Strides indicate how many positions are omitted between selected characters.
- The *stride* component can be negative, in which case the selection occurs backwards from the end of the string.

Exercise: Slicing

- Suppose that you have initialized **`ALPHABET`** as

  `ALPHABET = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"`

  so that the index numbers (in both directions) run like this:

A	**B**	**C**	**D**	**E**	**F**	**G**	**H**	**I**	**J**	**K**	**L**	**M**	**N**	**O**	**P**	**Q**	**R**	**S**	**T**	**U**	**V**	**W**	**X**	**Y**	**Z**
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
–26	–25	–24	–23	–22	–21	–20	–19	–18	–17	–16	–15	–14	–13	–12	–11	–10	–9	–8	–7	–6	–5	–4	–3	–2	–1

- What are the values of the following slice expressions?

(a) **`ALPHABET[7:9]`**

(b) **`ALPHABET[-3:-1]`**

(c) **`ALPHABET[:3]`**

(d) **`ALPHABET[-1:]`**

(e) **`ALPHABET[14:-12]`**

(f) **`ALPHABET[1:-1]`**

(g) **`ALPHABET[0:5:2]`**

(h) **`ALPHABET[::-1]`**

(i) **`ALPHABET[5:2:-1]`**

(j) **`ALPHABET[14:2:-3]`**

Methods for Finding Patterns

str`.find(`*pattern*`)` Returns the first index of *pattern* in *str,* or –1 if it does not appear.
str`.find(`*pattern*`, `*k*`)` Same as the one-argument version but starts searching from index *k*.
str`.rfind(`*pattern*`)` Returns the last index of *pattern* in *str,* or –1 if it does not appear.
str`.rfind(`*pattern*`, `*k*`)` Same as the one-argument version but searches backward from index *k*.
str`.startswith(`*prefix*`)` Returns `True` if this string starts with *prefix*.
str`.endswith(`*suffix*`)` Returns `True` if this string ends with *suffix*.

Methods for Transforming Strings

str`.lower()` Returns a copy of *str* with all letters converted to lowercase.
str`.upper()` Returns a copy of *str* with all letters converted to uppercase.
str`.capitalize()` Capitalizes the first character in *str* and converts the rest to lowercase.
str`.strip()` Removes whitespace characters from both ends of *str*.
str`.replace(`*old*`, `*new*`)` Returns a copy of *str* with all instances of *old* replaced by *new*.

Methods for Classifying Characters

ch`.isalpha()` Returns `True` if *ch* is a letter.
ch`.isdigit()` Returns `True` if *ch* is a digit.
ch`.isalnum()` Returns `True` if *ch* is a letter or a digit.
ch`.islower()` Returns `True` if *ch* is a lowercase letter.
ch`.isupper()` Returns `True` if *ch* is an uppercase letter.
ch`.isspace()` Returns `True` if *ch* is a ***whitespace character*** (space, tab, or newline).
str`.isidentifier()` Returns `True` if this string is a legal Python identifier.

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**

Source:https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-0001-introduction-to-computer-science-and-programming-in-python-fall-2016/lecture-slides-code/

Lists

```
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'`

Source:https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-0001-introduction-to-computer-science-and-programming-in-python-fall-2016/lecture-slides-code/

Lists

- 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

Source:https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-0001-introduction-to-computer-science-and-programming-in-python-fall-2016/lecture-slides-code/

Lists

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

like strings, can iterate over list elements directly

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

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

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

Source:https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-0001-introduction-to-computer-science-and-programming-in-python-fall-2016/lecture-slides-code/

Lists

- **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

Source:https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-0001-introduction-to-computer-science-and-programming-in-python-fall-2016/lecture-slides-code/

Lists

- 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]`

Source:https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-0001-introduction-to-computer-science-and-programming-in-python-fall-2016/lecture-slides-code/

Lists

- 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

all these operations mutate the list

```
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]
```

Source:https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-0001-introduction-to-computer-science-and-programming-in-python-fall-2016/lecture-slides-code/

Lists to String

- 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"
```

Source:https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-0001-introduction-to-computer-science-and-programming-in-python-fall-2016/lecture-slides-code/

Lists: Other Operations

- `sort()` and `sorted()`
- `reverse()`

```
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]`

https://docs.python.org/3/tutorial/datastructures.html

Source:https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-0001-introduction-to-computer-science-and-programming-in-python-fall-2016/lecture-slides-code/

Lists

- lists are **mutable**
- behave differently than immutable types
- is an object in memory
- variable name points to object
- any variable pointing to that object is affected
- key phrase to keep in mind when working with lists is **side effects**

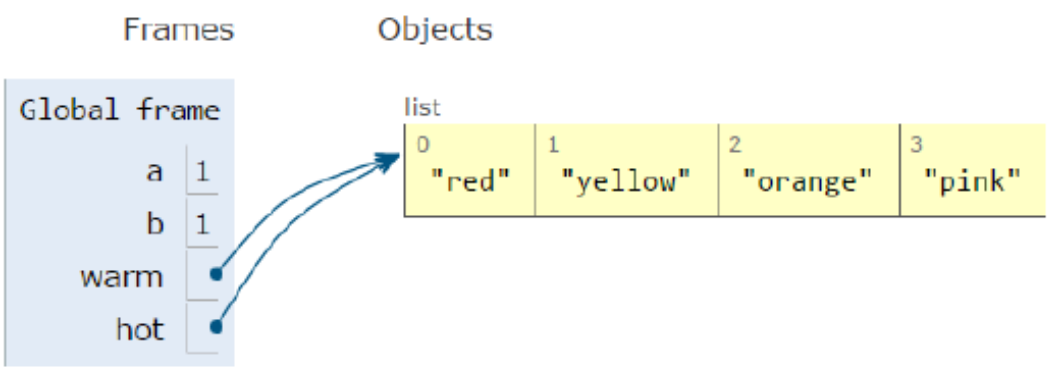
Source:https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-0001-introduction-to-computer-science-and-programming-in-python-fall-2016/lecture-slides-code/

Lists: Aliasing

- `hot` is an **alias** for `warm` – changing one changes the other!
- `append()` has a side effect

```
1  a = 1
2  b = a
3  print(a)
4  print(b)
5
6  warm = ['red', 'yellow', 'orange']
7  hot = warm
8  hot.append('pink')
9  print(hot)
10 print(warm)
```

```
1
1
['red', 'yellow', 'orange', 'pink']
['red', 'yellow', 'orange', 'pink']
```



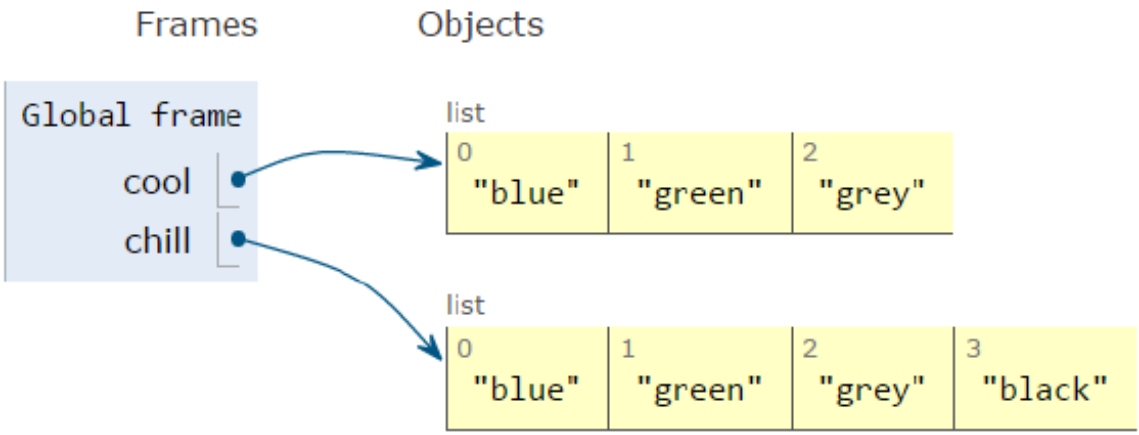
Source:https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-0001-introduction-to-computer-science-and-programming-in-python-fall-2016/lecture-slides-code/

Lists: Cloning

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

```
1 cool = ['blue', 'green', 'grey']
2 chill = cool[:]
3 chill.append('black')
4 print(chill)
5 print(cool)
```

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



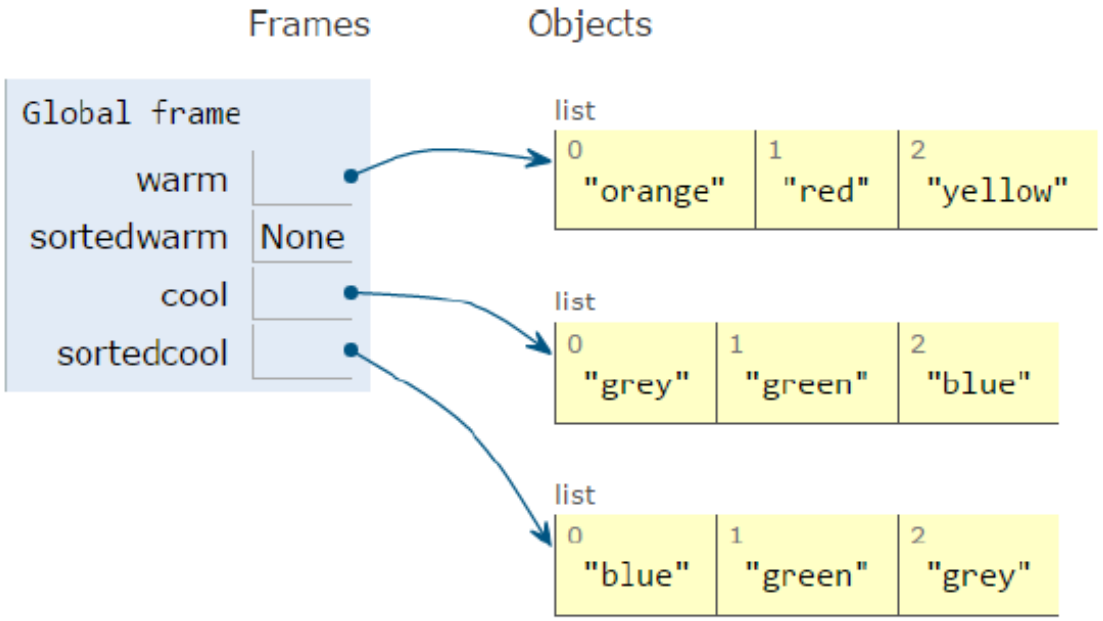
Source:https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-0001-introduction-to-computer-science-and-programming-in-python-fall-2016/lecture-slides-code/

Lists: Sorting

- 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']
```

```
1 warm = ['red', 'yellow', 'orange']
2 sortedwarm = warm.sort()
3 print(warm)
4 print(sortedwarm)
5
6 cool = ['grey', 'green', 'blue']
7 sortedcool = sorted(cool)
8 print(cool)
9 print(sortedcool)
```



Source:https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-0001-introduction-to-computer-science-and-programming-in-python-fall-2016/lecture-slides-code/