LAMBDA FUNCTIONS, TUPLES and LISTS

(download slides and .py files to follow along)

6.100L Lecture 9

Ana Bell

FROM LAST TIME

```
def apply(criteria,n):
    """
    * criteria: function that takes in a number and returns a bool
    * n: an int
    Returns how many ints from 0 to n (inclusive) match the
    criteria (i.e. return True when run with criteria) """
    count = 0
    for i in range(n+1):
        if criteria(i):
            count += 1
    return count

def is_even(x):
    return x%2==0

print(apply(is_even,10))
```

ANONYMOUS FUNCTIONS

- Sometimes don't want to name functions, especially simple ones. This function is a good example:

```
def is_even(x):
    return x%2==0
```

- Can use an **anonymous** procedure by using `lambda`

```
lambda x: x%2 == 0
```

parameter

Body of lambda
Note no return keyword

- `lambda` creates a procedure/function object, but simply does not bind a name to it

ANONYMOUS FUNCTIONS

- Function call with a named function:

```
apply( is_even , 10 )
```

- Function call with an anonymous function as parameter:

```
apply( lambda x: x%2 == 0 , 10 )
```

- `lambda` function is **one-time use**. It can't be reused because it has no name!

YOU TRY IT!

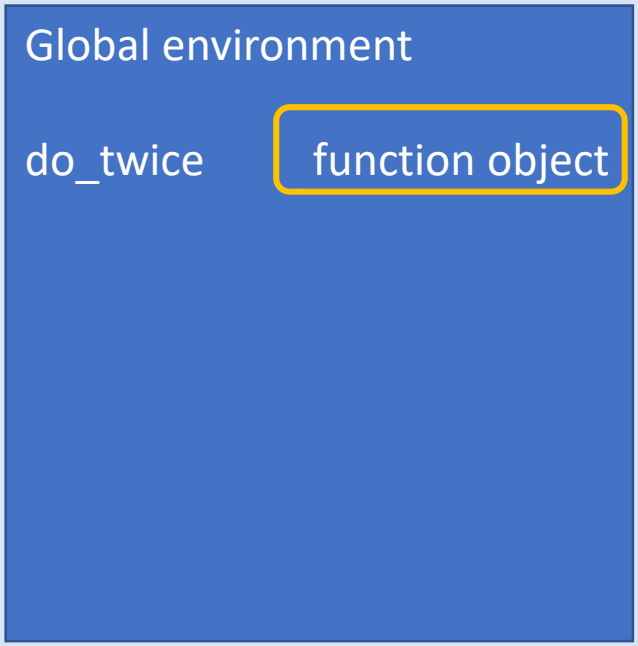
- What does this print?

```
def do_twice(n, fn):
    return fn(fn(n))

print(do_twice(3, lambda x: x**2))
```

YOU TRY IT!

- What does this print?

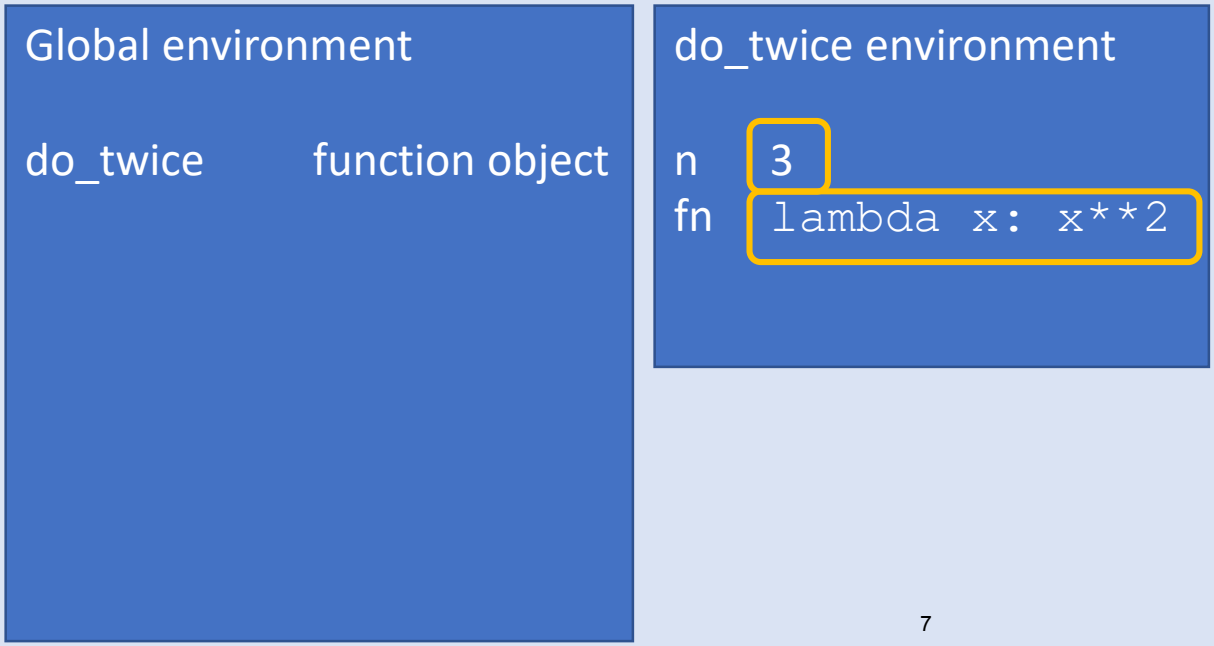
```
def do_twice(n, fn):
    return fn(fn(n))

print(do_twice(3, lambda x: x**2))
```



YOU TRY IT!

- What does this print?

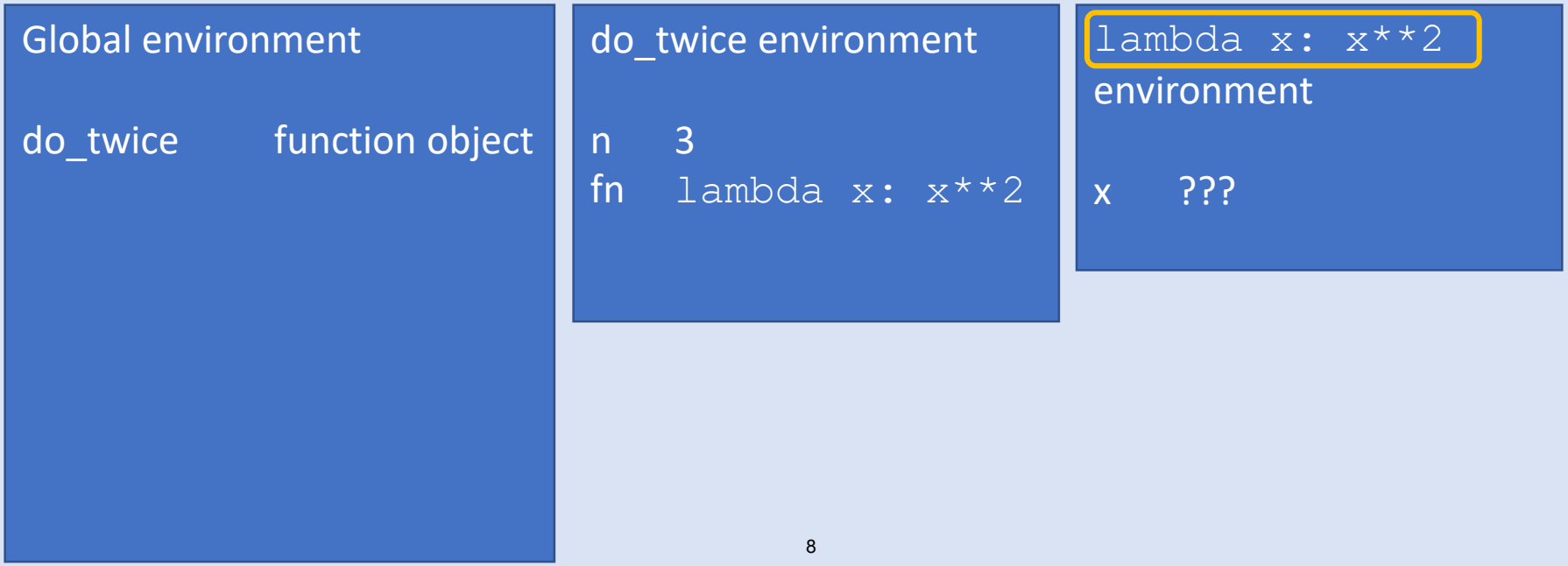
```
def do_twice(n, fn):
    return fn(fn(n))

print(do_twice(3, lambda x: x**2))
```



YOU TRY IT!

- What does this print?

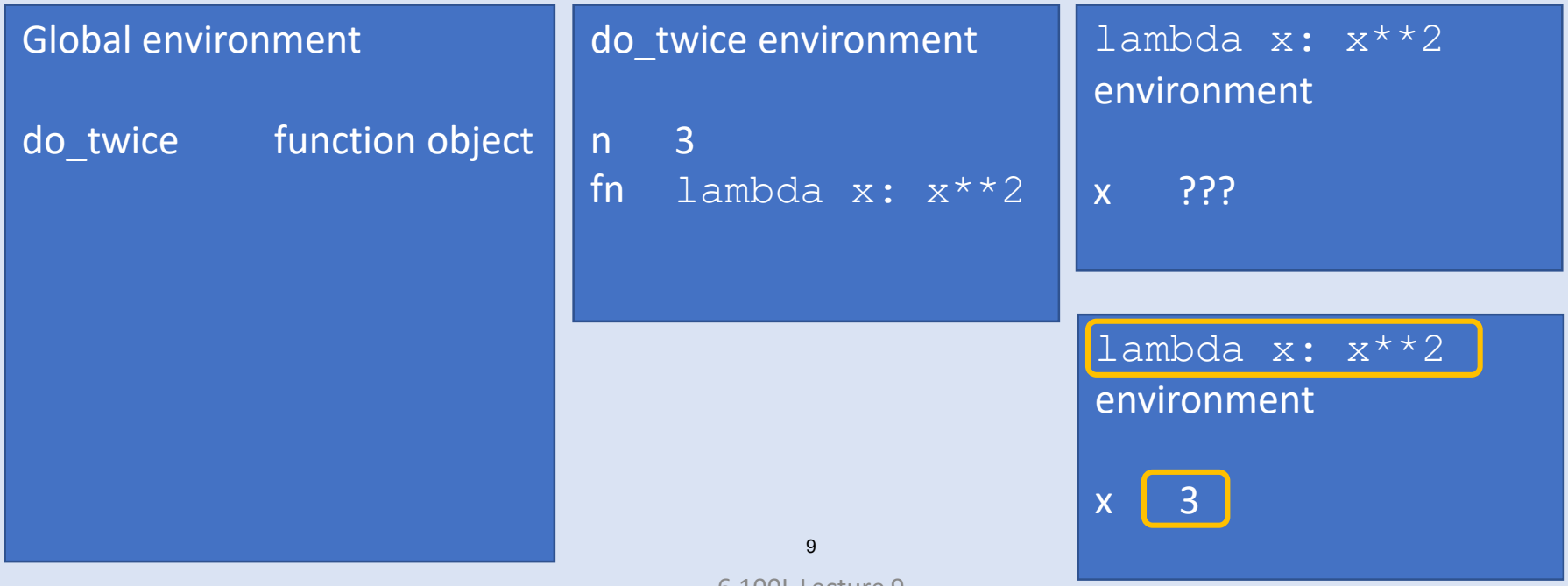
```
def do_twice(n, fn):
    return fn(fn(n))

print(do_twice(3, lambda x: x**2))
```



YOU TRY IT!

- What does this print?

```
def do_twice(n, fn):
    return fn(fn(n))

print(do_twice(3, lambda x: x**2))
```

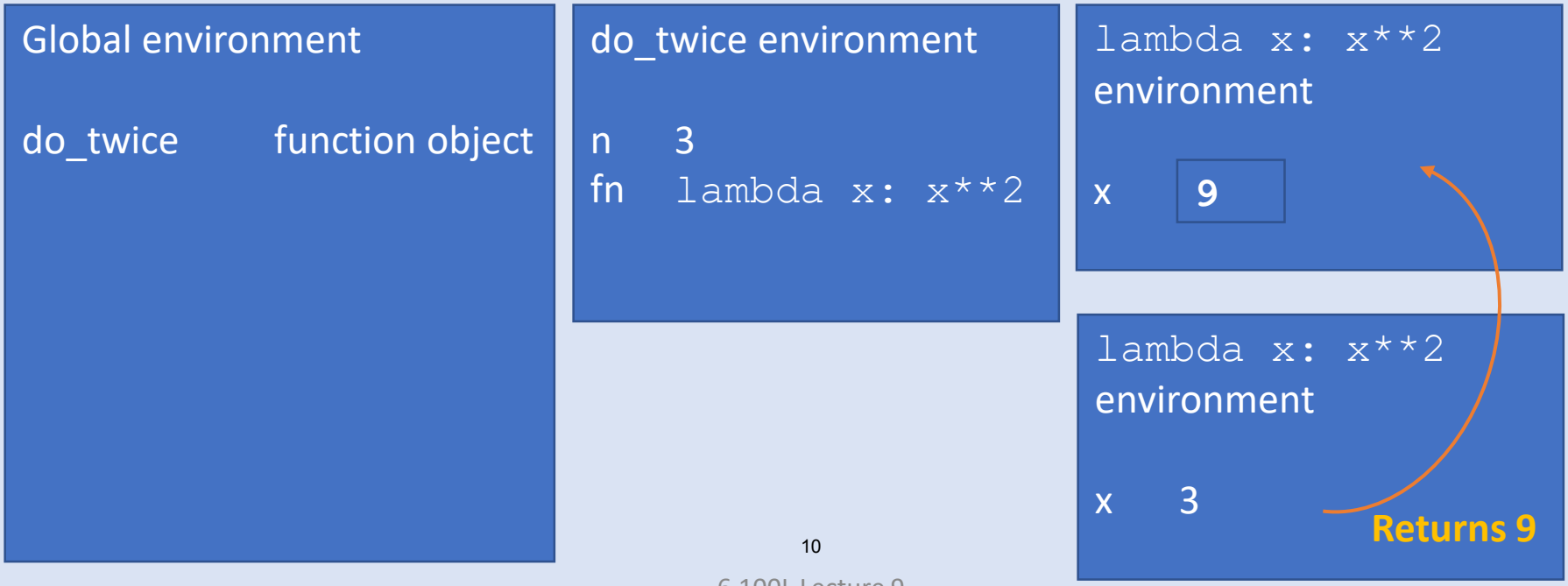


YOU TRY IT!

- What does this print?

```
def do_twice(n, fn):
    return fn(fn(n))

print(do_twice(3, lambda x: x**2))
```

9

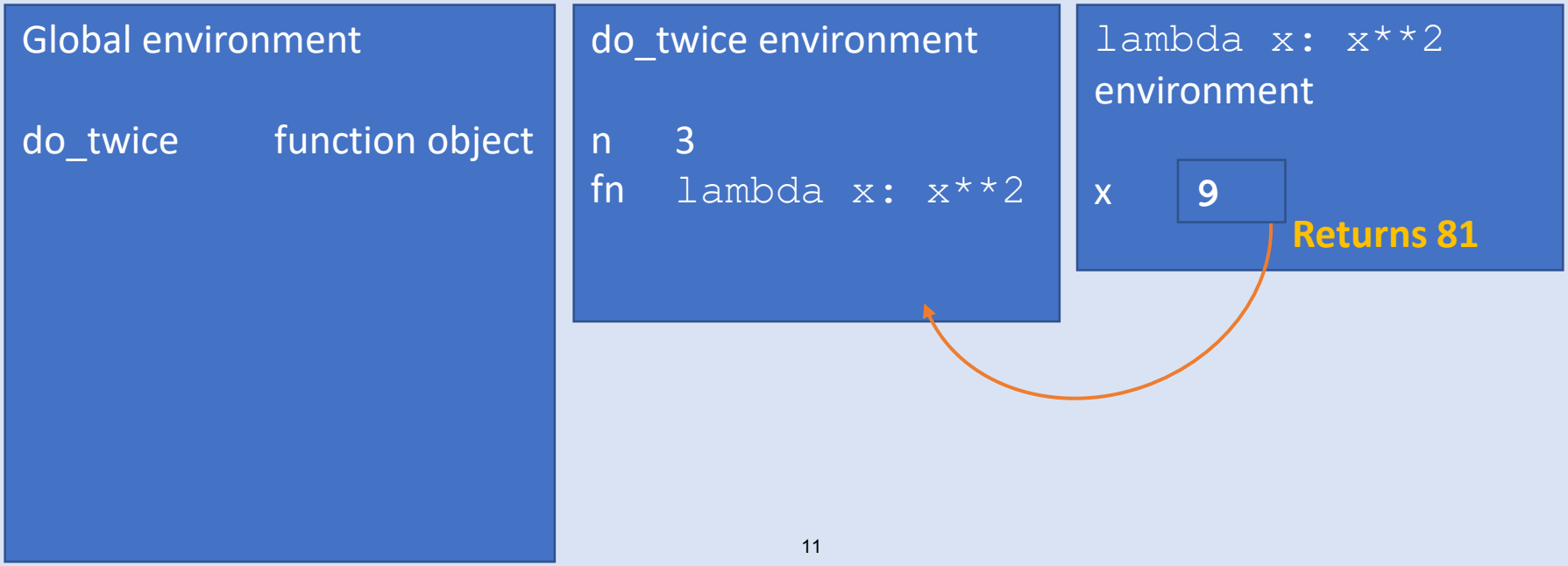


YOU TRY IT!

- What does this print?

```
def do_twice(n, fn):
    return fn(fn(n))

print(do_twice(3, lambda x: x**2))
```

81

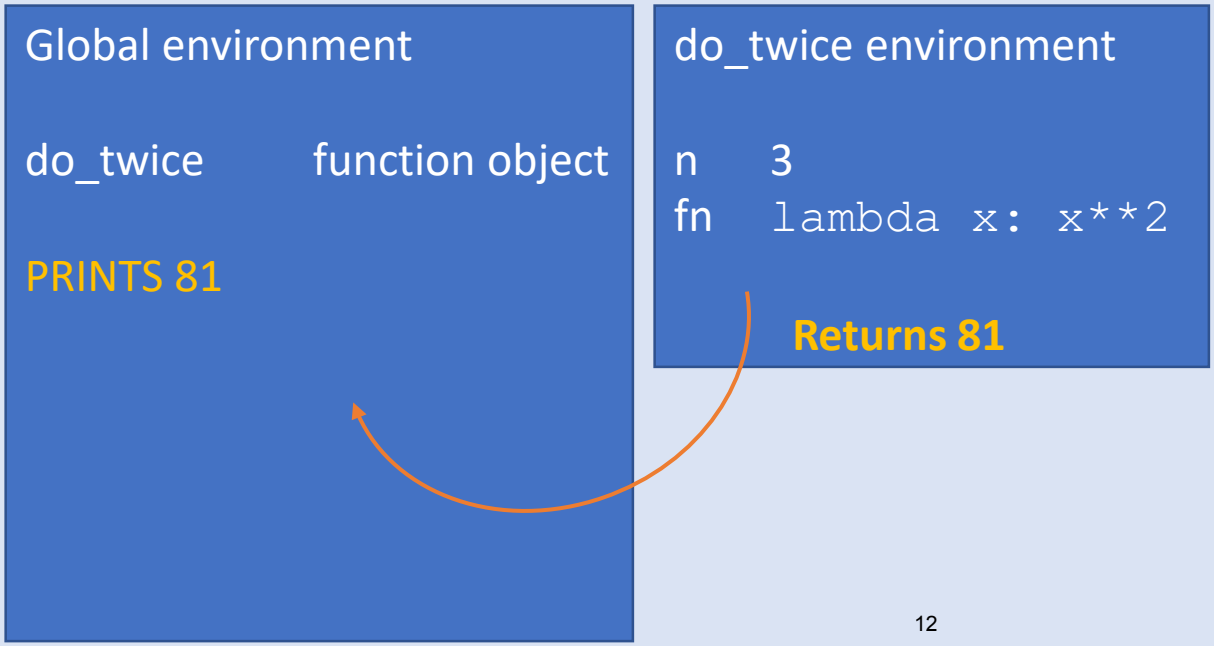


YOU TRY IT!

- What does this print?

```
def do_twice(n, fn):
    return fn(fn(n))

print(do_twice(3, lambda x: x**2))
```

81



TUPLES

A NEW DATA TYPE

- Have seen scalar types: `int`, `float`, `bool`
- Have seen one compound type: `string`
- Want to introduce more general **compound data types**
  - Indexed sequences of elements, which could themselves be compound structures
  - **Tuples** – immutable
  - **Lists** – mutable

- Next lecture, will explore ideas of
  - Mutability
  - Aliasing
  - Cloning

TUPLES

Remember strings?

- **Indexable ordered sequence** of objects
  - Objects can be any type – int, string, tuple, tuple of tuples, …
- Cannot change element values, **immutable**

```
te = ()
ts = (2,)
```

Empty tuple

Extra comma means tuple with one element

Compare with `ts = (2)`

```
t = (2, "mit", 3)
```

Multiple elements in tuple separated by commas

`t[0]` → evaluates to 2 Indexing starts at 0

`(2,"mit",3) + (5,6)` → evaluates to a new tuple `(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

`max((3,5,0))` → evaluates 5 Other functions also work, e.g, sum

`t[1] = 4` → gives error, **can't modify object**

INDICES AND SLICING

Remember strings?

```
seq = (2,'a',4,(1,2))
  index:  0   1   2    3
print(len(seq))          → 4
print(seq[3])            → (1,2)
print(seq[-1])           → (1,2)
print(seq[3][0])         → 1
print(seq[4])            → error
```

An element of a sequence is at an **index**, indices start at 0

```
print(seq[1])            → 'a'
print(seq[-2:]           → (4,(1,2))
print(seq[1:4:2]         → ('a',(1,2))
print(seq[:-1])          → (2,'a',4)
print(seq[1:3])          → ('a',4)
```

Slices extract subsequences.
Indices evaluated from left to right

```
for e in seq:            → 2
    print(e)               a
                           4
                         (1,2)
```

Iterating over sequences

TUPLES

- Conveniently used to **swap** variable values

```
x = 1
y = 2
x = y
y = x
```

```
x = 1
y = 2
temp = x
x = y
y = temp
```

```
x = 1
y = 2
(x, y) = (y, x)
```

First, create tuple
y = 2
x = 1

Then, bind values in new tuple

TUPLES

- 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)
```

One object!
(with 2 elements/values)

(3,1)

(3, 1)

```
both = quotient_and_remainder(10,3)
```

1

2

(2, 1)

```
(quot, rem) = quotient_and_remainder(5,2)
```

BIG IDEA

Returning
one object (a tuple)
allows you to return
multiple values (tuple elements)

YOU TRY IT!

- Write a function that meets these specs:
- Hint: remember how to check if a character is in a string?

```
def char_counts(s):
    """ s is a string of lowercase chars
    Return a tuple where the first element is the
    number of vowels in s and the second element
    is the number of consonants in s """
```

VARIABLE NUMBER of ARGUMENTS

- Python has some built-in functions that take variable number of arguments, e.g, `min`
- Python allows a programmer to have same capability, using *** notation**

```
def mean(*args):
    tot = 0
    for a in args:
        tot += a
    return tot/len(args)
```

- `numbers` is bound to a **tuple of the supplied values**
- Example:
  - `mean(1,2,3,4,5,6)`

args → (1,2,3,4,5,6)

LISTS

LISTS

- **Indexable ordered sequence** of objects
  - Usually homogeneous (i.e., all integers, all strings, all lists)
  - But can contain mixed types (not common)
- Denoted by **square brackets**, [ ]
- **Mutable**, this means you can change values of specific elements of list

Tuples were ()

Remember tuples are immutable – you **cannot** change element values.
Lists are mutable, you can change them directly.

INDICES and ORDERING

Remember strings and tuples?

```
a_list = []
```
empty list

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

`[1,2]+[3,4]` → evaluates to `[1,2,3,4]`

`len(L)` → evaluates to 4 — Gives length of top level of tuple

`L[0]` → evaluates to 2 — Indexing starts at 0

`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'`

`max([3,5,0])` → evaluates 5

ITERATING OVER a LIST

- Compute the **sum of elements** of a list
- Common pattern

```
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 elements of list **directly**

This version is more "pythonic"!

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

ITERATING OVER a LIST

- Natural to capture iteration over a list inside a function

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

```
def list_sum(L):
    total = 0
    for i in L:
        # i is 8 then 3 then 5
        total += i
    return total
```

- Function call `list_sum([8,3,5])`
  - **Loop variable `i` takes on values in the list in order!** 8 then 3 then 5
  - To help you write code and debug, comment on what the loop var values are so you don't get confused!

LISTS SUPPORT ITERATION

- Because lists are ordered sequences of elements, they naturally interface with iterative functions

Add the ***elements*** of a list

```
def list_sum(L):
    total = 0
    for e in L:
        total += e
    return(total)
```

e is: 1 then 3 then 5

```
list_sum([1,3,5]) → 9
```

Add the ***length of elements*** of a list

```
def len_sum(L):
    total = 0
    for s in L:
        total += len(s)
    return(total)
```

s is: 'ab' then 'def' then 'g'

2 then 3 then 1

```
len_sum(['ab', 'def', 'g']) → 6
```

YOU TRY IT!

- Write a function that meets these specs:

```
def sum_and_prod(L):
    """ L is a list of numbers
    Return a tuple where the first value is the
    sum of all elements in L and the second value
    is the product of all elements in L """
```

SUMMARY

- **Lambda functions** are useful when you need a simple function once, and whose body can be written in one line
- **Tuples** are indexable sequences of objects
  - Can't change its elements, for ex. can't add more objects to a tuple
  - Syntax is to use ()
- **Lists** are indexable sequences of objects
  - Can change its elements. Will see this next time!
  - Syntax is to use []
- Lists and tuples are very **similar to strings** in terms of
  - Indexing,
  - Slicing,
  - Looping over elements

6.100L Introduction to Computer Science and Programming Using Python
Fall 2022