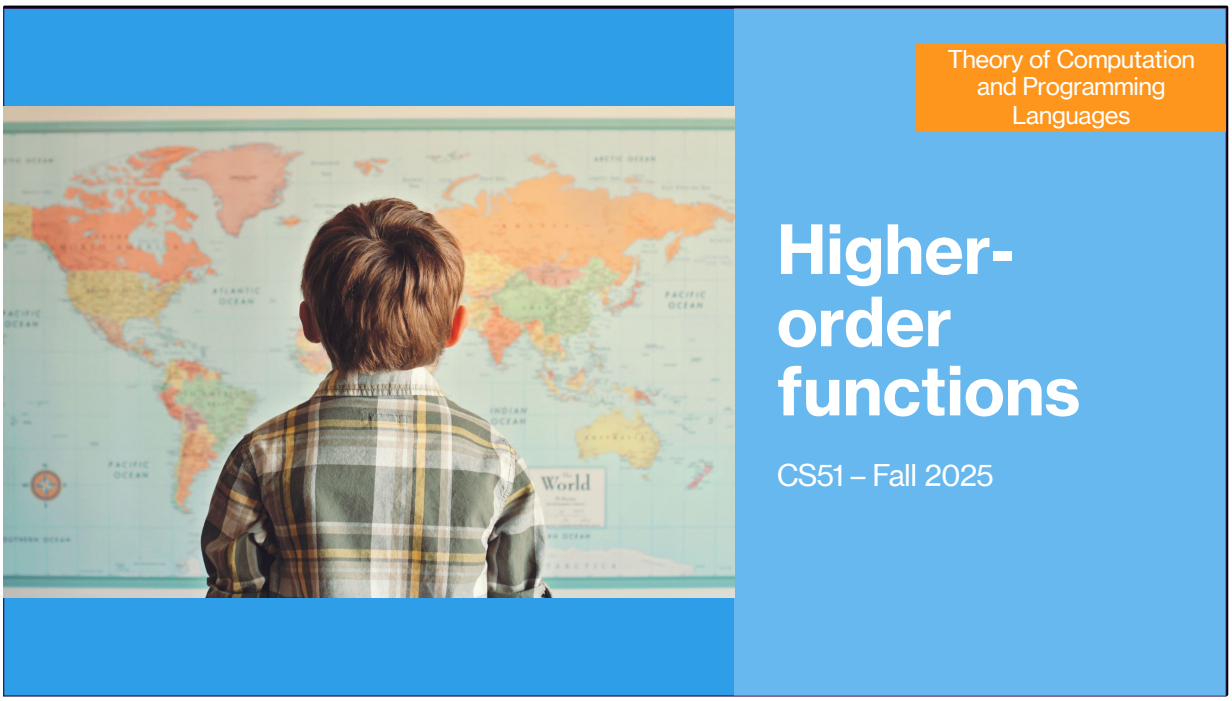


A new unit is here! We are ready to explore concepts from theory of computation and programming languages, areas of CS that are covered in CS54 and CS101. If you have taken CS50 some of the concepts we will see might sound familiar.

Higher order functions

- Have you ever called a function but forgot the parentheses? For example,

```
def add_one(x):
    return x+1
>> add_one
<function add_one at 0x108e962f0>
```

- Notice that it does not give an error.

```
>>> type(add_one)
<class 'function'>
```

2

Have you ever defined a function and when trying to call it you forgot the parentheses? What would happen in this example? You would get something like this is a function and that's its name rather than an error. In fact, if we pass add_one to type, we get that its type is function.

Referencing a function

- Functions in Python are objects, just like everything else!

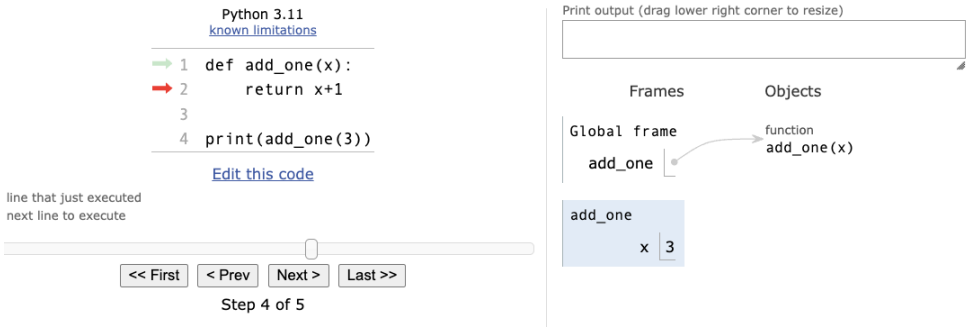
```
>>> y = add_one
>>> y
<function add_one at 0x108e962f0>
>>> y(2)
3
>>> my_abs = abs
>>> my_abs(-10)
10
```

- What the keyword `def some_function` actually does is create a new function object and reference it with `some_function`.

3

Functions in Python are objects, like everything else. That means we can create references to them as always, e.g., we can create a reference y to add_one. So if next we try to see what y is, it will give us the same message as before. We can even call a function using the reference y and pass the argument it expects. We can not only reference our own functions but built in functions like abs (returns the absolute value of a number). So all this time whenever we wrote def some_function we meant create a new function and reference it by that name.

Referencing a function



https://pythontutor.com/render.html#mode=display

4

Remember you can always use pythontutor and see that indeed add_one is just a name that references a function object.

Higher-order functions

- We can pass functions as parameters, return them from functions, even create them on the fly!
- Such functions are known as **higher-order functions**.
- We can define our own or work with built-in higher-order functions.

5

In general, when working with functions, we can pass them as parameters, return them from functions, even create them on the fly. A higher-order function is a function that either takes another function as an argument or returns a function as its result. Today we will see both how we can define our own higher order functions and how to work with built-in ones that Python provides.

Take a look at `higher_order_functions.py`

- What do the first four functions do?
  - Take two parameter and do standard mathematical calculations.
- What does `add2` do in `higher_order_functions.py` ?
  - Takes one parameter, a tuple of two items.
  - Unpacks the tuple, adds and returns its items.
- What does `double` do in `higher_order_functions.py`?
  - Takes one parameter.
  - Multiplies by 2 and returns it.
- What does `is_even` do in `higher_order_functions.py` ?
  - Takes one parameter, a number.
  - Returns whether this number is even.

6

Please open the higher_order_functions file linked at the end of the slide. The first four (subtract, add, multiply, and exponent) take two parameters and perform standard mathematical calculations. What about add2? It only takes one parameter, and it appears this is a tuple because it unpacks the tuple adds, and returns its items. How about double? It takes one parameter, multiplies it by 2 and returns it. Finally, what does is_even do? It returns whether the provided number is even.

Take a look at `higher_order_functions.py`

- What does `apply_function` do?
  - Takes three parameters
  - The first is a function! Then passes second and third parameter to that function and returns the result.

```
>>> apply_function(add, 2, 3)
5
>>> apply_function(subtract, 2, 3)
-1
```

7

Let's continue in the same fashion. What do you think apply_function do? It takes three parameters and the first one is a function. It then passes the second and third parameters to that function and returns the result. What do you think apply_function(add, 2, 3) would return? It's equivalent to writing return add(2,3) so it will give us back 5. Similarly, apply_function(subtract, 2, 3) will be equivalent to saying return subtract(2,3) so it will give us back -1.

Take a look at higher_order_functions.py

- What does the apply_function_to_list function?
  - Takes a function f and a list some_list as parameters
  - Creates an empty list result
  - Iterates through each value in some_list
  - Applies the function f to each value and appends it to the result.
- High-level: applies the function f to each element in some_list and returns a new list containing the result from each of those applications.
- For example:

```
>>> apply_function_to_list(double, [1, 2, 3, 4])
[2, 4, 6, 8]
>>> apply_function_to_list(add2, [(1, 2), (3, 4)])
[3, 7]
```

8

What do you think the apply_function_to_list function do in higher_order_functions.py?

Takes a function f and a list some_list as parameters
Creates an empty list result
Iterates through each value in some_list
Applies the function f to each value and appends it to the result.

All that to say that it applies the function f to each element in some_list and returns a new list containing the result from each of those applications.

Take a look at `higher_order_functions.py`

- What does the `apply_function_to_tuple` function do?
  - Takes a function `f` and a list `some_list` of 2-tuples as parameters
  - Iterates through each 2-tuple in `some_list` and unpacks it
  - Applies the function `f` on the two unpacked items
  - Appends the result of this application to a list that is returned at the end.
- For example:

```
>>> apply_function_to_tuple(add, [(1, 2), (3, 4)])
[3, 7]
```

9

Let's look next at apply_function_to_tuple. We see that it takes a function `f` and a list `some_list` of 2-tuples as parameters. The function f has to take two parameter. It then creates a new list, iterates through each 2-tuple in `some_list` , unpacks it, applies the function f on the two unpacked items and appends the result of this application to result.

map

- `apply_function_to_list` is actually built-in to Python and is called `map`.
- `map` takes as input a function and something that is iterable
  - only difference from apply_function_to_list is that it returns a map object (not a list), which is also iterable. For example:

```
>>> map(double, [1, 2, 3, 4])
<map object at 0x7f7ff809b128>
>>> for val in map(double, [1, 2, 3, 4]):
        print(val)
2
4
6
8
```

10

What we just saw with apply_function_to_list is a higher-order function that comes built-in Python and is called map. map takes as input a function and an iterable object and returns back a map object which is also iterable.

map

- By itself, this may not seem useful, but we can do more complicated things. What would this print?

```
for val in map(double, map(double, [1, 2, 3, 4])):
    print(val)
```

4

8

12

16

The first call to map doubles it and then we iterate on this result and double it again!

11

What do you think this for loop would do? The first call to map will double each list element and then the second will double it again

Take a look at `higher_order_functions.py`

- What does the `filter_list` function do?
- What does `some_function` return when passed a `some_list` element?
  - It should return a `bool`
- Similarly to *map*, Python has a built-in function for this behavior called `filter`.
- The `filter` function returns a list of all elements of `some_list` that would return `True` when passed to `some_function`. Note how it differs from *map*. For example,

```
>>> list(map(is_even, [1, 2, 3, 4]))
[False, True, False, True]
>>> list(filter(is_even, [1, 2, 3, 4]))
[2, 4]
```

12

Let's look at filter_list next. Are there any expectations on some_function should return when given an element from some_list? Given that it's passed to an if statement it should return a bool. This higher-order function is already implemented for us in Python as filter. The `filter` function returns a list of all elements of `some_list` that would return `True` when passed to `some_function`. Note that it filters out elements rather than applying a function to them.

Anonymous functions

- It can be a bit annoying having to define all of these simple functions to simply pass them as an argument to another function.
- Python allows us to create **anonymous functions**, i.e., functions that don't have an explicit name, but are simply code.
- The syntax is:
  `lambda <input>: <expression>`
- <input> is the parameter(s) to the anonymous function.
  - If you need to pass multiple inputs, just pass them as a tuple.
- <expression> is the body of the function that is executed and returned. It can only be a single expression (i.e., something that represents a value).

13

So far we have been defining functions and then passing them as arguments to another function. Python allows us to create anonymous functions, that is functions that don't have a name. The syntax to create an anonymous function is lambda input: expression. input is the parameter(s) you want to pass to the anonymous function. Expression is the body of the function that is executed and returned. It can only be a single expression so not multiple lines.

Anonymous functions

- An example:
```
>>> lambda x: x+1
<function <lambda> at 0x7f7ff80981e0>
```
- Notice that it gives the same function type back, but it doesn't have a name!
```
>>> (lambda x: x+1)(2)
3
```
- We can also associate it with a variable and call it, e.g.,
```
f = lambda x: x+1
>>> f(2)
3
```

14

For example, lambda x : x+1 will define an anonymous function that takes one input, x, and then increases t by 1 and returns it. Kinda defeating the purpose of an anonymous function, we could reference it, for example here by f, and call it.

Anonymous functions

- An example:

```
>>> filter_list(lambda num: num % 2 == 0, [1, 2, 3, 4])
[2, 4]
```

15

What do you think the following call will do? Remember, filter_list expects a function and a list. Instead of passing it the name of a function we have already defined, we can create an anonymous function on the spot. That function will take one parameter, num, and will return a bool based on whether the number was even. Since filter_list applies the function on each element, it will return back a list only of the even numbers.

Practice time

- Write anonymous functions for the following procedures:
- Multiply argument `num1` with argument `num2` and return the result:
- Add arguments `num1`, `num2`, `num3` and return the result

16

Your turn to write two simple anonymous functions.

Answer

- Write anonymous functions for the following procedures:
- Multiply argument num1 with argument num2 and return the result:
  - `lambda num1, num2 : num1 * num2`
- Add arguments num1, num2, num3 and return the result
- `lambda num1, num2, num3 : num1 + num2 + num3`

17

Remember, anonymous functions start with the keyword lambda, followed by the input(s) to the function (if more than one, separate by comma, as a tuple) followed by a colon, and then the body of the function of what we want to return (we don't write return!)

`sorted` function

- Python supports a built-in function that takes an iterable object, such as a list, string, tuple, dictionary, etc), and returns a **new** sorted list from it without modifying the original.
- ```
  numbers = [3, 2, 5, 1]
  sorted_numbers = sorted(numbers)
  print(numbers) # remains [3, 2, 5, 1]
  print(sorted_numbers) # is now [1, 2, 3, 5]
  ```
- It can also sort reversely, e.g.,

```
sorted_numbers = sorted(numbers, reverse = True)
print(sorted_numbers) # is now [5, 3, 2, 1]
```

18

The last built-in Python higher-order function we will see is sorted. sorted takes an iterable object (something like a sequence or dictionary) and returns a new sorted list from it without modifying the original. Behind the scenes it uses a sorting algorithm called timsort but the neat thing is that you can call it without needing to worry about its correctness. If you give it a list of numbers it will order it non-decreasing order, and a list of strings in lexicographic order. More flexibly than our own implementations, it can optionally take an argument that can specify that we want the reverse order, e.g., from largest to smallest.
```

`sorted` function

- Sometimes, we want to sort our data in a non-standard way, e.g., instead of lexicographic order for a list of strings sort by length.
- sorted can take a key that is a function to be called on each list element prior to making comparisons.
- ```
  fruit = ['apple', 'banana', 'plum', 'kiwi']
  sorted_fruit = sorted(fruit, key = len)
  ```

```
  # secretely converts it into [5, 6, 4, 4] which uses to sort
print(sorted_fruit) # ['plum', 'kiwi', 'apple', 'banana']
```

19

Sometimes, we might want to do even more unusual things, like instead of sorting a list of string sin lexicographic order we want to sort it by their length. sorted has another optional parameter called key which takes a function (that's why it's a higher order function). For example, we can pass len. Behind the scenes, the function passed is applied on each key and its result is used to determine the ordering.
```

Anonymous functions and `sorted`

- We can combine what we learned about anonymous functions and the `sorted` function.
- ```
  professors = [('Papoutsaki', 222), ('Kauchak', 224), ('Osborn', 103)]
  sorted_professors_by_office = sorted(professors, key = lambda x:x[1])
  # [('Osborn', 103), ('Papoutsaki', 222), ('Kauchak', 224)]
  sorted_professors_by_name = sorted(professors, key = lambda x:x[0])
  # [[('Kauchak', 224), ('Osborn', 103), ('Papoutsaki', 222)]
  ```

20

Since key is a function we can combine what we learned about anonymous functions and sorted. for example, if I have a list of 2-item tuples, I can sort my list based on either of these items.
  ```

Practice time

- What would the following code do?

```
my_list = [3, 6, 3, 2, 4, 8, 23]
sorted(mylist, key=lambda x: x % 2 == 0)
```

https://stackoverflow.com/questions/8966538/syntax-behind-sortedkey-lambda

21

Doo you think this code will do?

Answer

```
my_list = [3, 6, 3, 2, 4, 8, 23]
my_sorted_list = sorted(mylist, key=lambda x: x % 2 == 0)
# secretely considers it as [False, True, False, True, True, True,
False]
# Remember False==0, True == 1, thus False<True
# my_sorted_list == [3, 3, 23, 6, 2, 4, 8]
```

https://stackoverflow.com/questions/8966538/syntax-behind-sortedkey-lambda

22

It will turn `my_sorted_list == [3, 3, 23, 6, 2, 4, 8]`. This is because the lambda expression is applied and since it returns a bool and False is smaller than True, the odd items come before the even ones. Note that we don't sort two individual sublists, the original ordering is retained.

Last thing on nonymous functions

- Let's look at this unusual function that inside it defines a function and returns it??

```
def kinda_crazy(num):
    def multiplier(x):
        return num * x
    return multiplier
>>>type(kinda_crazy(3))
<class 'function'>
>>>kinda_crazy(3)(2)
6
```

- We could use an anonymous function to be even more concise!

```
def crazy(num):
    return lambda x: num * x
>>> crazy(3)(2)
6
```

23

Last thing that we will see is the function kinda_crazy which inside it defines a function returns a … function. How do we call a function that returns a function? if we just pass one argument, we get back a function, the function multiplier. So we would need to pass a second argument (in another set of parentheses. We could have avoided defining a named function within a function if we used anonymous functions.

Code:

- higher_order_functions.py