Computer Programming Midterm Review

2024/04/09 Tim Chen

Contents

Recursion

Recursion

Recursion a.k.a. **Recurs**ive Funct**ion**

- Tips:
  1. Base case: e.g. $F_0 = 1, F_1 = 1$
  2. Recursive case: e.g. $F_n = F_{n-1} + F_{n-2}$ for $n > 1$
  3. Call stack
- Fibonacci Numbers

$$\begin{cases} F_0 = 1, F_1 = 1 \\ F_n = F_{n-1} + F_{n-2} \text{ for } n > 1 \end{cases}$$

Recursion (cont.)

```
def fib(n: int) -> int:
  """Assume n >= 0, returns fibonacci of n"""
  if n == 0 or n == 1:
    return 1
  else:
    return fib(n - 1) + fib(n - 2)

print(fib(5)) # 8
```

Scoping

Scoping

The `global` keyword

```
def fib(n: int) -> int:
  """Assume n >= 0, returns fibonacci of n"""
  global numFibCalls
  numFibCalls += 1
  if n == 0 or n == 1:
    return 1
  else:
    return fib(n - 1) + fib(n - 2)
```

Tuples, List & Dictionary

Tuple

- **Immutable ordered sequences** of elements.
- Elements can be of **any type**.
- Declare & Define `tuple_test1 = (1, 'two')`
- Repetition: `2 * tuple_test1 # Equals (1, 'two', 1, 'two')`
- Iteration:

```
for t in tuple_test1:
  print(t)
# 1 two 1 two
```

List

- Append: Add the given variable to the next element.
- Extend: Add the element of given iterable object to the end of the list.
- List is mutable, the above operation will modify the list directly.

```
l1 = [1, 2, 3]
li.append(4)       # l1 = [1, 2, 3, 4]
li.extend([5, 6]) # l1 = [1, 2, 3, 4, 5, 6]
```

List (cont.)

```
l1 = [1, 2, 3, 4, 5]
print(l1[1:3])          # [2, 3]
print(l1[-1])           # 5
print(l1[-3], l1[2])    # 3 3
```

Notes: Avoid mutating a list when it is iterating.

Dictionary

- A set of key/value pairs.
- Element is written as a key followed by a colon followed by a value.

```
monthNumbers = {'Jan': 1, 'Feb': 2, 'Mar': 3, 'Apr': 4}
print(monthNumbers['Jan']) # 1
```

- Dictionaries are mutable. e.g. `monthNumbers['May'] = 5`

Dictionary (cont.)

- To iterate through a dictionary:

```
for m in monthNumbers:
  print(m) # Jan, Feb, Mar, Apr, May

for m in monthNumbers:
  print(monthNumbers[m]) # 1, 2, 3, 4
```

List Comprehension

List Comprehension

- A concise way to apply operations to the values in a sequence.
- Create a new list in which each element is the result of applying a given operation.

```
listOfNum = [x for x in range(0, 7)]
print(listOfNum) # [0, 1, 2, 3, 4, 5, 6]
```

List Comprehension (cont.)

If statement

```
randomNum = [1, 2, 3, 4, 5]
odd = [x for x in randomNum if x % 2 != 0]
even = [x for x in randomNum if x % 2 == 0]
```

If-else statement

```
doubleOrNot = [2*x if x%2==0 else x for x in randomNum]
# [1, 4, 3, 8, 5]
```

Higher-order function

Higher-order function

`map` - the built-in higher-order function. ⟶ `map(func, iter)`

- Designed to be used with a `for` loop

```
def double(n):
  return 2 * n

for i in map(double, [1, 2, 3]):
  print(i) # 2, 4, 6
```

Higher-order function

lambda expression: `lambda <seq of variable>: <expression>`

```
for i in map(lambda x: x**2, [1, 2, 3, 4]):
  print(i) # 1, 4, 9, 16
```

Other Tips for Midterm

Other Tips for Midterm

1. Nested functions are allowed. e.g.

```
def outer_func():
  def inner_func():
    pass
  pass
```

Other Tips for Midterm (cont.)

2. Lamdba function can accept more than one value. e.g.

```
for i in map(lambda x, y: x*y, [1, 2, 3], [4, 5, 6]):
  print(i) # 4, 10, 18
```

Other Tips for Midterm (cont.)

3. Use table to easily trace the variables' values. e.g.

```
def fib(n):
  if n == 0 or n == 1:
    return 1
  else:
    return fib(n - 1) + fib(n - 2)

fib(4)
```

Other Tips for Midterm (cont.)

Note: target = `fib(4)`

step	target	return value	backtrace
1	`fib(4)`	`fib(3) + fib(2)`	5
2	`fib(3)`	`fib(2) + fib(1)`	3
3	`fib(2)`	`fib(1) + fib(0)`	2
4	`fib(1)`	1	N/A
5	`fib(0)`	1	N/A

Thank you