Discussion 7

DSC 20, Winter 2024

Midterm Review

No worksheet today, go over past questions, look over practice exams

Meme of the Week

Agenda

Topics:

- Basics
- Loops
- Data Types
- return vs print
- Files
- List Comp
- Asserts
- Lambda
- Iterator/map/filter
- HOF
- Complexity
- Advanced argument passing
- Recursion

Basic Operations

+ - Numerical addition / concatenation operator

- - Numerical Subtraction operator

/ - Classic Division operator

// - Floor Division operator

***** - Numerical Multiplication / repetition operator

****** - Numerical Exponential operator

% - Numeric remainder operator

Checkpoint

What is the result of this expression?

In [ ]:
```
2 ** ((int(3.7) // 2) % 2)
```

A. 0

B. 1

C. 2

D. 4

Checkpoint Solution

In [8]:
```
2 ** ((int(3.7) // 2) % 2)
```

Out[8]:
```
2
```

Boolean Operators

Logical Operators (in order priority):

- **not** - reverses the outcome of the following expression
- **and** - all expressions compared with “and” must be True to evaluate True
- **or** - at least one expression compared with “or” must be True to evaluate True

Comparison Operators (generates booleans):

- **==** - equality check
- **!=** - inequality check
- **>, >=, <, <=** directional check

note: order of evaluation is overriden by parentheses (just like PEMDAS)

Short-circuits

- In python, expressions are evaluated from left to right
- With certain operations, a "short-ciruit" can happen if conditions are met
- Certain functions within python also act as short-circuits -> return is a short circuit

Conditional Statements

if (boolean expression):

    //Do stuff

elif (other boolean expression):

note: elif is optional, can have as many elifs as necessary

    //Do other stuff

else:

note: else is optional, will execute only if the conditions in the “if” and “elif” statements are not true. "else" is the final part of a chained conditional statement.

    //Do other other stuff

Checkpoint

What is the result of this expression?

Checkpoint

What is the result of this expression?

In [ ]:
```
def foo(x):
    if x%2==0 and x//0:
        return x
    else:
        return 'oops'
foo(3)
```

A. 'oops'

B. 3

C. Error

Checkpoint Solution

In [10]:
```
def foo(x):
    if x%2==0 and x//0:
        return x
    else:
        return 'oops'
foo(3)
```

Out[10]:
```
'oops'
```

return vs print

'return' is a very important and special keyword in python

- It's python's functional way to pass a value out of a function
- not every function needs a return (but many have one)
- functions with no explicitly defined return will return None by default

'print' displays some output onto a console

- commonly used for debugging, sometimes integral to the purpose of a function.
- print's result is None (it doesn't "return" a meaningful value)

Doctests

- Tests to check that your function works as intended
- denoted by the '>>> ' symbol (space included)!
- the line right after the '>>> ' represents the intended output
- well written doctests make sure your code is logically sound

Checkpoint

What is the result of this expression?

In [ ]:
```
def fizz(x):
    return print(x)
fizz(10)
```

A. None

B. None None

C. 10 10

D. 10 None

Loops

Loops are used to **repeat computations** many times.

- Two types of loops:
    - While loop: Uses logical conditions, useful for when the number of iterations is unknown (as long as a condition is true, code will run).
    - For loop : Uses an iterable object (ex. list), usually for when the number repetition is known.

Mutability

- Object is mutable if it can be directly changed after it is created
- If it can't, it is immutable
- Lists are **mutable**
- strings, tuples, and numbers are **immutable**

Checkpoint

What is the result of this expression?

In [ ]:
```
def buzz(x):
    output = ''
    for i in x:
        if i.isnumeric():
            output +=i
        else:
            i = 'Z'
            output +=i
    return output

buzz('abc123')
```

A. 'ZZZ123'

B. 'ZZZZZZ'

C. 'abcZZZ'

D. Error

Checkpoint Solution

In [15]:
```
def buzz(x):
    output = ''
    for i in x:
        if i.isnumeric():
            output +=i
        else:
            i = 'Z'
            output +=i
    return output

buzz('abc123')
```

Out[15]:
```
'ZZZ123'
```

Lists, Tuples, Sets

List

- Mutable vector of values
- Can store any data type, multiple types at a time
- Elements are acccessed via indexing

Tuple

- **Immutable** vector of values
- all else equal to list

Set

- Mutable vector of values
- Only stores unique elements (removes duplicates)
- otherwise, list like behavior

Indexing/Slicing

Indexing/slicing refers to accessing specific element(s) from an iterable object. Two of the most common cases for this are lists and strings. Indexing results in a copy (unless reasssigned)!

- iterable[start:stop:skip] (start:inclusive, stop: NOT inclusive)
- not every section needs to be specified (can just use start or stop or skip)
- sub indexes can be applied (ex. lst[0][0] -> takes the first element of the first element)
- Trying to access an index that doesn't exist in the list will result in an error

Checkpoint

What is the result of this expression?

In [ ]:
```
lst = list(range(0,11))
lst[3:8:2][::-1]
```

A. [7, 5, 3]

B. [3, 5, 7]

C. [2, 4, 6]

D. [6, 4, 2]

Checkpoint Solution

In [22]:
```
lst = list(range(0,11))
lst[3:8:2][::-1]
```

Out[22]:
```
[7, 5, 3]
```

List Comprehension

- Fancy, shorthand method of writing for loops
- Syntax changes depending on use case
- can be nested in each other, just like lists
- Can contain multiple for loops in one list comp
- Can also be a nested loop

  Syntax

  - [x for x in iterable]
  - [x for x in iterable if (condition)]
  - [x if (condition) else y for x in iterable]
  - [x if (condition) else y if (condition) else z for x in iterable]

Dictionaries

- Mutable storage of key, value pairs
- Can store any data type, multiple at a time
- Elements are acccessed via keys
- keys must be **hashable** and **unique**

methods

- accessing keys (as a list) -> dict.keys()
- accessing values (as a list) -> dict.values()
- accessing key,value pairs (as a list of tuples) -> dict.items()

note: hashablility correlates to the stability of the data - essentially, **data that can't change is hashable** (int, str, tuple, etc.) while **data that can change is not hashable (list, dictionary). Basically, it's all about mutability!**

Dictionary Comprehension

- Fancy, shorthand method of populating dictionaries
- Syntax changes depending on use case

Syntax

- basically the same as list comp, but now it expects key:value
- can include a list comp!

Checkpoint

Write an expression to yield the value 'marina'

In [23]:
```
dct = {
    "nikki":{"nikki":1},
    ('a','b','c'):[1, {'nikki':'marina'}],
    'max':'nikki'
}
```

A. list(dct[('a','b','c')][0].items())[0][1]

B. dct['nikki']['nikki']

C. dct[('a','b','c')][1][0]

D. list(dct[('a','b','c')][1].values())[0]

Checkpoint Solution

In [1]:
```
dct = {
    "nikki":{"nikki":1},
    ('a','b','c'):[1, {'nikki':'marina'}],
    'max':'nikki'
}
list(dct[('a','b','c')][1].values())[0]
```

Out[1]:
```
'marina'
```

Checkpoint

What is the result of this expression? (Bonus: What is the function calculating?)

In [ ]:
```
def boo(x):
    return {a:[b for b in range(1,a+1) if a%b==0] for a in range(1,x+1)}
boo(6)
```

A. {1: [1], 2: [1, 2], 3: [1, 3], 4: [1, 2, 3, 4], 5: [1, 5], 6: [1, 3, 6]}

B. {1: [1], 2: [1, 2], 3: [1, 2, 3], 4: [1, 2, 4], 5: [1, 2, 5], 6: [1, 2, 3, 4, 6]}

C. {1: [1], 2: [1, 2], 3: [1, 2, 3], 4: [1, 2, 3, 4], 5: [1, 2, 3, 4, 5], 6: [1, 2, 3, 4, 5, 6]}

D. {1: [1], 2: [1, 2], 3: [1, 3], 4: [1, 2, 4], 5: [1, 5], 6: [1, 2, 3, 6]}

Checkpoint Solution

In [35]:
```
def factor_decomposer(x):
    return {a:[b for b in range(1,a+1) if a%b==0] for a in range(1,x+1)}
factor_decomposer(6)
```

Out[35]:
```
{1: [1], 2: [1, 2], 3: [1, 3], 4: [1, 2, 4], 5: [1, 5], 6: [1,
2, 3, 6]}
```

Files

- storage for data (think csv's from DSC10, txt's from assignments, etc.)
- unique methods to access within code

Access Modes

Write : 'w' -> every time the file is opened in write mode, the file is wiped. Calling file.write() will add in your data.

Append : 'a' -> file.write() will append your data to what existed in the file beforehand.

Read : 'r' -> no writing privilege, can only pull the data from the file with relevant methods.

note: If you try to open a file in write mode that doesn't exist, python will create it.

Text Processing

reading data:

- file.read() -> reads in all the data as a single string
- file.readline() -> reads in data line by line (has to be recalled)
- file.readlines() -> reads in all the data as a list where each line is another element of the list

After reading in the data, you can transform it however you'd like, and then rewrite it back into the file using .write() (if this is relevant).

Assert Statements

- Used to evaluate written code
- **asserts** -> input validation (are the arguments the correct types?)
- Often combined with boolean functions (any(), all(), etc.)

Lambda Functions

- known as anonymous functions (their functions are so simple, they don't need a name)
- syntax: lambda (input): (some operation)
- within the scope of this course, lambda is used in conjunction with map and filter

Iterator

Iterator - **Syntax: iter(iterable)**, **next(iterator)**

- An iterator in Python is an object that can be iterated upon, meaning that you can traverse through all the values.
- Typically, an iterator is created from an iterable using the `iter()` function and the elements are accessed via the `next()` function.
- Iterators remember the state as you traverse through them. The next call to `next()` starts off where the previous one stopped.

Map

Map - **Syntax: map(function, iterable)**

- Map allows you to apply a function to all elements to an iterable input
- very common to use a lambda function as the function to apply
- returns a lazy iterator through the iterable object, applying the function as it traverses

Filter

Filter - **Syntax: filter(function, iterable)**

- Filter takes in a function that returns a boolean and only keeps elements that satisfy the function (i.e. return True).
- Very common to use a lambda function as the function to apply, but keep in mind the function **must return a boolean**.
- Returns a lazy iterator through the iterable object that only yields values that pass the function.

Checkpoint

Given a file with the following content, use map/filter/lambda to create a dictionary consisting of entries from people from 'California' who are above 18.

In [ ]:
```
marina,Russia,30
ben,Taiwan,21
bryce,California,20
Anish,California,18
```

In [ ]:
```
def people_filter(file_path):
    """
    >>> people_filter('files/people.txt')
    {'bryce':['California', '20']}
    """
    # Either try writing it
    # or think about what steps you would do
```

Checkpoint Solution

In [46]:
```
def people_filter(file_path):
    with open(file_path, 'r') as f:
        data = f.readlines()
    filtered_cali = filter(lambda x: x.split(',')[1]=='California', dat
    filtered_age = filter(lambda x: int(x.split(',')[2]) > 18, filtered
    stripped = map(lambda x: x.strip(), filtered_age)
    return dict(map(lambda x: (x.split(',')[0], x.split(',')[1:]),strip
```

In [47]:
```
people_filter('files/people.txt')
```

Out[47]:
```
{'bryce': ['California', '20']}
```

HOF

- Algorithm design framework to create generalized code
- Functions that either return functions or use other functions
- Helper functions!
- Many uses, including abstraction, scope protection, etc.

Complexity Fundamentals

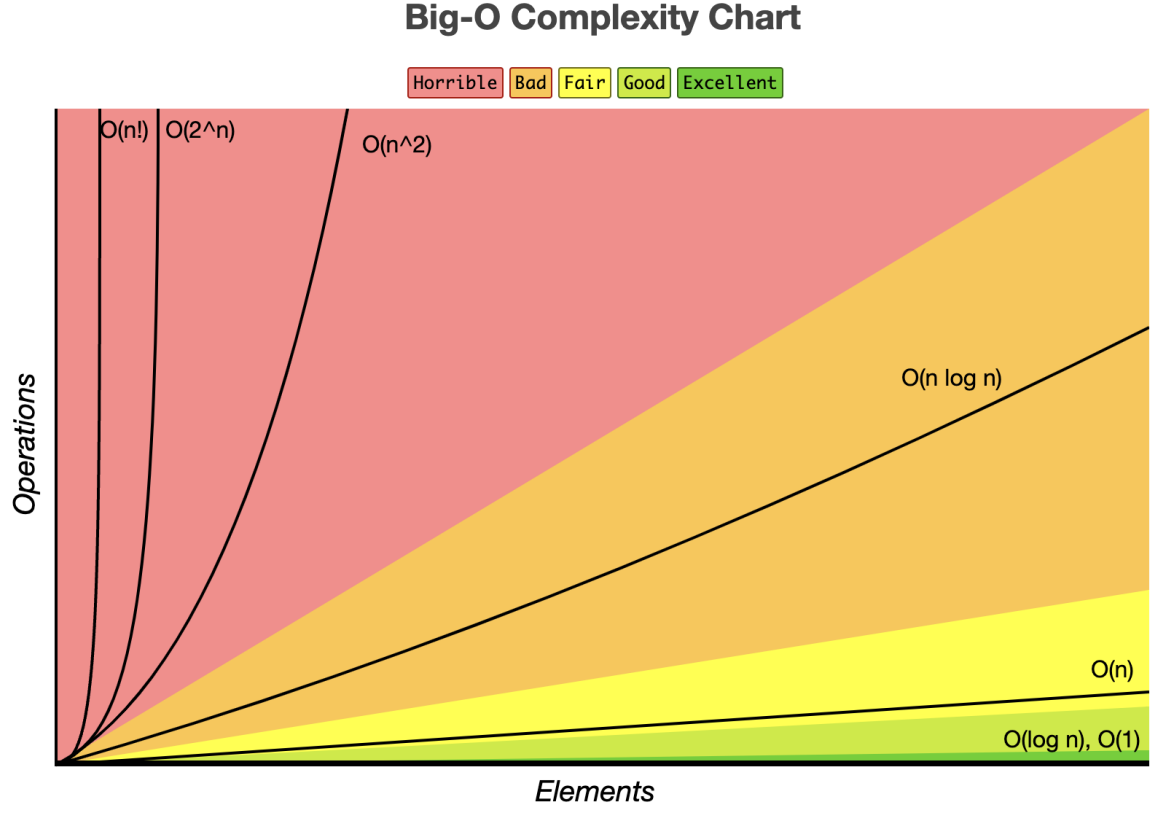
Time complexity is an empirical method to measure the efficiency of code. Since everyone's computer is different, we can't compare code with actual numbers. Instead, we classify them into different runtime categories that allow us to infer its runtime relative to our input.

1. Code should be quantified into big O values
2. Nested code will have compounded big O values
3. The largest term dictates the growth rate (i.e. only largest term matters)
4. Constants are irrelevant
5. big O analysis uses similar ideas to calculus and limits - reference your math knowledge

Complexity - Calculation

tips:

1. Always look for "hallmark features" (ex. if I see i // 2, there's probably log involved)
2. If there are loops, check the iteration count (don't assume n runtime)
3. Order of Growth follows mathematical principles. Only consider the largest term



Checkpoint

What is the runtime of the following function?

In [9]:
```
def permute_list(lst):
    n = len(lst)
    c = [0] * n
    print(lst)

    i = 0
    while i < n:
        if c[i] < i:
            if i % 2 == 0:
                lst[0], lst[i] = lst[i], lst[0]
            else:
                lst[c[i]], lst[i] = lst[i], lst[c[i]]

            print(lst)
            c[i] += 1
            i = 0
        else:
            c[i] = 0
            i += 1

permute_list([1, 2, 3])
```

```
[1, 2, 3]
[2, 1, 3]
[3, 1, 2]
```

```
[1, 3, 2]
[2, 3, 1]
[3, 2, 1]
```

hint: Instead of reading the code, think about what it's doing - there's a natural answer that arises.

Checkpoint Solution

The time complexity of this function is O(n!), where n is the length of the list.

This is because the function generates all possible orderings of the list elements. There are n! (factorial of n) such orderings.

For each recursive call, the function iterates over the elements of the list, making n recursive calls at the first level, n-1 at the next level, and so on, down to 1. This results in n! calls in total.

*args

- Used when an unknown number of arguments will be passed into a function
- Denoted by * in the method header (IMPORTANT)
- processed in a similar manner to a list

**kwargs

- Used when an unknown number of **keyworded** arguments will be passed into a function
- Denoted by ** in the method header (IMPORTANT)
- processed in a similar manner to a dictionary

default_arguments

- Basically normal arguments, but with a default value
- if no value is passed, default value is set
- if a value is passed, default value is overwritten

note

complex argument ordering gets really messy

In [2]:
```
def func(norm, *args, darg=2, **kwargs):
    return [norm, list(args), darg, dict(kwargs)]
func(42,1,1,1,1,1,1,3,darg=4, test=1)
```

Out[2]:
```
[42, [1, 1, 1, 1, 1, 1, 3], 4, {'test': 1}]
```

In [3]:
```
def func(norm, darg=2, *args, **kwargs):
    return [norm, list(args), darg, dict(kwargs)]
func(42,4,1,1,1,1,1,1,3,test=1)
```

Out[3]:
```
[42, [1, 1, 1, 1, 1, 1, 3], 4, {'test': 1}]
```

Checkpoint

What is the result of the following expression?

In [13]:
```
def permute_division(*n):
    return [a//b for a in n for b in range(max(n))]
permute_division(5)
```

A. [1, 1, 1, 2]

B. [1, 1, 2, 5]

C. [5, 2, 1, 1]

D. Error

Checkpoint Solution

In [18]:
```
def permute_division(*n):
    return [a//b for a in n for b in range(max(n))]
permute_division(5)
```

```
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent
call last)
Cell In[18], line 3
      1 def permute_division(*n):
      2     return [a//b for a in n for b in range(max(n))]
----> 3 permute_division(5)

Cell In[18], line 2, in permute_division(*n)
      1 def permute_division(*n):
----> 2     return [a//b for a in n for b in range(max(n))]

Cell In[18], line 2, in <listcomp>(.0)
      1 def permute_division(*n):
----> 2     return [a//b for a in n for b in range(max(n))]

ZeroDivisionError: integer division or modulo by zero
```

Recursion

Recursion is a design method for code - it refers to a class of functions that call on itself repeatedly

Base Case:

Base case(s) are regarded as the most important part of a recursive function. They determine the stop point for recursion and begin the argument passing up the "stack" of recursive calls. Without a well written base case, recursion will either never end or end incorrectly. When writing recursion questions, always start with determining the base case.

Recursive Calls:

The crux of a recursive function working is the recursive calls. These calls will repeat until the base case is reached, creating the "stack" of recursive calls that will begin resolving at the base case. Keep in mind when writing recursive calls that every call needs to trend towards the base case.

Checkpoint

What is the time complexity of the following expression?

In [ ]:

```
def foo(x):
    if x%10==0:
        return x
    else:
        return foo(x-1)
foo(8)
```

A.

B.

C.

D.

E. Other

Checkpoint Solution

In [16]:
```
def foo(x):
    if x%10==0:
        return x
    else:
        return foo(x-1)
print([foo(x) for x in range(20)])
```

```
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 10, 10, 10, 10, 10, 10, 10, 10, 1
0, 10]
```

Solution :

Regardless of the input to this function, the maximum number of times it can occur recursively is 9 (i.e. if I pass in any number that ends in 9, that is the "worst case" for this function). Since the runtime is **deterministic**, the time complexity of this function is .

Checkpoint

Write a recursive function to check if a given string is a palindrome

In [19]:

```
def check_palindrome(x):
    """
    >>> check_palindrome('chatgpt')
    False
    >>> check_palindrome('racecar')
    True
    """
    # Try it or think about what you would write
```

Checkpoint Solution

In [23]:
```python
def check_palindrome(x):
    """
    >>> check_palindrome('chatgpt')
    False
    >>> check_palindrome('racecar')
    True
    """
    if len(x)<=1:
        return True

    return all([x[0]==x[-1], check_palindrome(x[1:-1])])
```

In [24]:
```python
check_palindrome('chatgpt')
```

Out[24]:
```
False
```

In [25]:
```python
check_palindrome('racecar')
```

Out[25]:
```
True
```

Thanks for coming!