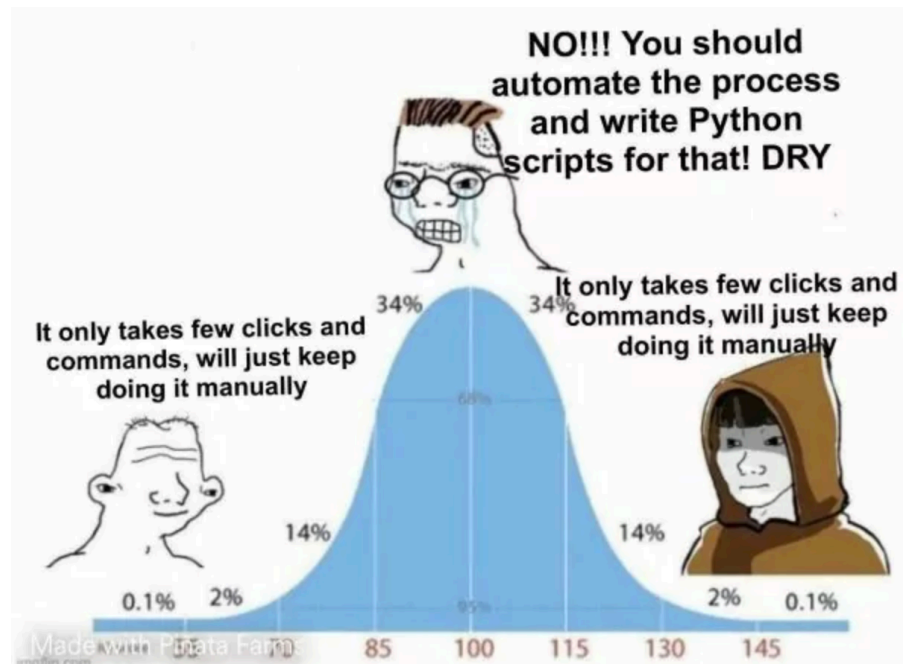


Discussion 10

DSC 20, Winter 2024

Final Exam Review👉

Meme of the week



Logistics

- Final Exam takes place Saturday, March 16th
- 3pm - 6pm
- Location: refer to EDstem (CSB 01, CSB 02, SOLIS 104, SOLIS 107)
- Seating chart will be released soon
- cheatsheet allowed (normal A4 double-sided)
- make sure to bring ID and pencils

Midterm 1 Content

- Basics
- Conditional Statements
- Doctests
- Loops
- List, tuple, set, Dictionaries
- List/Dict comp
- indexing/slicing
- files/text processing
- asserts

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

Basic Data Types

String - Data type for text

Int - Data type for whole numbers

Float - Data type for non-integers

Bool - True or False

note: **None** is technically its own type

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 overridden by parentheses (just like PEMDAS)

Checkpoint

What is the output of the following code?

```
In [ ]: bool(-100)
```

Checkpoint Solution

```
In [2]: bool(-100)
```

```
Out[2]: True
```

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

```
//Do other other stuff
```

Short-circuits, 'return', 'print'

- In python, expressions are evaluated from left to right
- With certain operations, a "short-circuit" can happen if conditions are met
 - ex: True or 1/0 -> "or" only requires 1 True, so the expression is done evaluating before 1/0 can trigger an error.
- **'return'** is a very important and special keyword in python
 - all statements with return will short-circuit the function afterwards
 - It's python's functional way to pass a value out of a function
 - not every function needs a return (but most do)
- **'print'** displays some object (int, str, bool, etc.) 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)

Checkpoint

Which terminal command(s) runs all doctests?

1. `python doctest code.py`
2. `python -i doctest code.py`
3. `python -m doctest code.py`
4. `python -m -v doctest code.py`

Checkpoint Solution

3 - `python -m doctest code.py`

(according to past final exam performances, I did in fact need to include this.)

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

to run doctests: terminal -> file location -> **python -m doctest [filename].py**

note: if you want to see the doctests you passed when you run them, add a '-v' at the end.

Assert Statements

- Used to evaluate written code
- **asserts** -> input validation (are the arguments the correct types?)
- Often combined with boolean functions (any(), all(), etc.) and list comp
- result should be a boolean (True = pass, False = error)
- check naive cases first, then more specific cases

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.

```
In [3]: while x is True:
        # do something
        for value in x:
            # do something
```

```
Cell In[3], line 3
    for value in x:
    ^
```

```
IndentationError: expected an indented block after 'while' state
ment on line 1
```

Mutability

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

Lists, Tuples, Sets

List

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

Tuple

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

Set

- Mutable vector of values
- Only stores unique elements (removes duplicates)
- Order is not preserved

List Comprehension

- Shorthand method of writing for loops
- Syntax changes depending on use case
- can be nested, just like lists
- Can contain multiple for loops in one list comp

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]

```
In [2]: temp = [[1], [2,3], [4,5,6]]  
        [x for sub in temp for x in sub]
```

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

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 reassigned)!

- `iterable[start:stop:skip]` (start:inclusive, stop: NOT inclusive)
- not every section needs to be specified (can just use start or stop or skip)
- Trying to access an index that doesn't exist in the list will result in an error
- Slicing beyond the range will yield an empty output

```
In [4]: lst = list(range(2,7))
print("original list: " + str(lst))
print("reversed list: " + str(lst[::-1]))
print("the 2nd to 4th element: " + str(lst[2:4]))
print("every third element from the 1st to 10th element: " + str(lst[1:]))
print("slicing outside of range: " + str(lst[11:]))
```

original list: [2, 3, 4, 5, 6]

reversed list: [6, 5, 4, 3, 2]

the 2nd to 4th element: [4, 5]

every third element from the 1st to 10th element: [3, 6]

slicing outside of range: []

Dictionaries

- Mutable storage of key, value pairs
- Can store any data type, multiple at a time
- Elements are accessed via keys (dct[key])
- 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: hashability 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!

```
In [3]: data = ['a', 'b', 'c', 'd', 'e']  
        {i:data[i] for i in range(len(data)) if i%2==0}
```

```
Out[3]: {0: 'a', 2: 'c', 4: 'e'}
```

Files

- storage for data (csv, txt, json, parquet, 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.

Checkpoint

What happens when I try to open a file in my current folder that doesn't exist yet in write mode?

Checkpoint Solution

Python will automatically write an empty file with the provided filename.

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).

```
In [3]: with open('files/sample.txt', 'r') as f:
        print(f.read())
        with open('files/sample.txt', 'r') as f:
            print(f.readlines())
```

Final Exam is coming up!

Good luck!

```
['Final Exam is coming up!\n', 'Good luck!']
```

Checkpoint

What is the result of this expression?

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

- A. None
- B. None None
- C. 10 10
- D. 10 None

Checkpoint Solution

```
In [1]: def fizz(x):  
        return print(x)  
        print(fizz(10))
```

10

None

Midterm 2 Content

- lambda/map/filter
- HOF
- advanced argument passing
- time complexity
- recursion

Lambda Functions

- known as anonymous functions (their functions are so simple, they don't need a name)
- syntax: lambda (input): (some operation)
- Lambdas cannot take in `for` or `return`

```
In [8]: lambda x: return x
```

```
Cell In[8], line 1
    lambda x: return x
              ^
SyntaxError: invalid syntax
```

```
In [14]: lambda x: for i in range(len(x))
```

```
Cell In[14], line 1
    lambda x: for i in range(len(x))
              ^
SyntaxError: invalid syntax
```

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 an 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 an iterator through the iterable object that only yields values that pass the function.

I literally cannot emphasize this more, Filter's lambda has to return a boolean

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.

*args

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

```
In [9]: def summation(*nums):  
        return sum(nums)  
print(summation())  
print(summation(1,2,3,4,5))
```

```
0  
15
```

****kwargs**

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

```
In [15]: def create_dct(**entry):  
          return dict(entry)  
          print(create_dct())  
          print(create_dct(marina=1, langlois=2))
```

```
{}  
{'marina': 1, 'langlois': 2}
```

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

Complexity Fundamentals

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

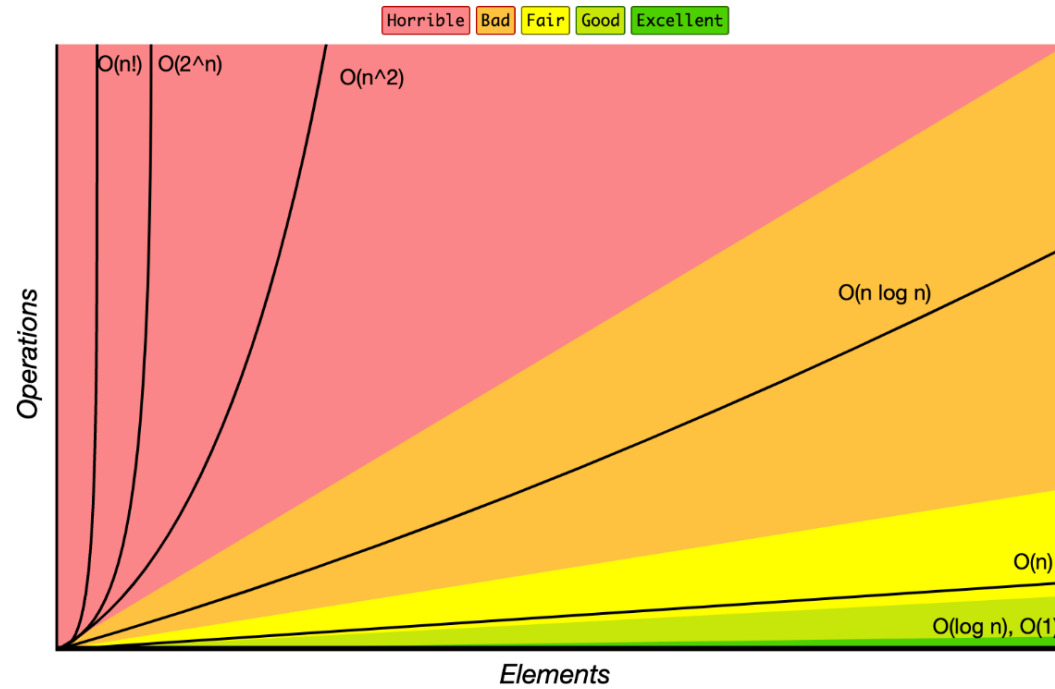
Time Complexity

Utilize mathematical principles to classify code as runtime families to quantify their relative efficiency

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 AND the inside operations (don't assume n runtime)
3. Order of Growth follows mathematical principles. Only consider the largest term (without constants)

Big-O Complexity Chart



[source](#)

Checkpoint

What is the runtime of the following code?

```
for _ in range(n, n^2, 1):  
    print('hi')
```

Checkpoint Solution

Does become or ? [desmos](#)

Recursion - 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.

Recursion - 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.

New Content

- classes
- inheritance
- special methods
- ASCII

Classes

- aggregate code into function bodies
- custom defined objects for specialized use

```
In [11]: class phone:
          """
          Class representation of a regular phone.
          """
          pass
```

Constructors

- denoted by `__init__` (special function)
- should ingest self as the first parameter (ALWAYS)
- populates instance with attributes
- if no constructor is explicitly defined, a default one is used

Working with self

- Think of classes as a blueprint and everytime you create an instance of one, you've produced a "physical" manifestation.
- We use the self keyword to interact with THIS specific instance of the class.
- Try to think of 'self' as referencing a **specific, singular** instance of the class. Every method that works with a specific instance's values must be passed in self as an argument.

```
In [18]: class phone:
        """
        Class representation of a regular phone.
        """
        id_num = 1 # class variable
        def __init__(self, maker, version, owner):
            self.maker = maker
            self.version = version
            self.owner = owner
            self.id = phone.id_num
            phone.id_num+=1
            # instance variables
            self.apps = []
            self.free_memory = 800

        def install_app(self, app, memory):
            if self.free_memory - memory >= 0:
                self.apps.append(app)
                self.free_memory -= memory
                return True
            return False # class method
```


@classmethod

The `@classmethod` decorator modifies a method to become a class method. Class methods affect the class itself and are called on the class rather than its instances. This means that they receive the class as their first argument (`cls`) instead of the instance (`self`).

- **Class-Level Operation:** `@classmethod` allows a method to modify class state that applies across all instances or to call other class methods.
- **Access to Instance Variables:** While class methods can access class attributes, they do not have direct access to instance variables unless explicitly passed an instance. This is because they are not bound to a specific instance but to the class itself.

@staticmethod

The `@staticmethod` decorator transforms a method into a static method. Static methods, unlike class methods or instance methods, do not require a reference to the instance (`self`) or class (`cls`). This makes them behave more like plain functions that happen to reside within the class.

- **No Automatic Arguments:** `@staticmethod` functions do not automatically receive the `cls` or `self` arguments. They behave like regular functions but belong to the class's scope.
- **Use Cases:** Static methods are used when some processing is related to the class but does not require the class or its instances to perform its task. They can be called on the class itself or on instances of the class and are usually utility or helper functions.

Inheritance

- classes can have a "parent-child" relationship
- child class(es) "inherit" methods from their parent class
- use "is-a" paradigm to distinguish; given a parent class phone and a child class pearphone:
 - pearphones are phones (child is a parent)
 - but not all phones are pearphones (parents are not children)
- Though not required, child classes can **overwrite** methods of their parent class
- if certain things are to be retained, `super()` is a useful function to use

note: `super` refers to the direct parent of a class

In [20]:

```
class pearphone(phone):
    def __init__(self, maker, version, owner, age):
        # self is implicitly passed with super()
        super().__init__(maker, version, owner)
        self.age = age

        self.apps = []
        self.free_memory = 800

    def install_app(self, app, memory):
        '''install_app but better (whoo pear phones!)'''
        if self.free_memory - memory//2 >= 0:
            self.apps.append(app)
            self.free_memory -= memory//2
            return True
        return False
```

Special Methods

- Unique class methods that impose specific functionality
- Denoted by `__` prepending and appending method names
- The one you're most familiar with is `__init__`
- Also called Dunder Methods!

`__str__`

- human readable representation of a class
- invoked when the string representation of a class is necessary (i.e. in a print or casted to a string)
- does not invoke `__repr__`

`__repr__`

- computer/technical representation of a class
- commonly written so that the returned value can be used to recreate the object
- if no `__str__` is implemented, `__repr__` will be used in lieu

__(lt, gt, eq, etc.)__

- encode operation between class instances

```
In [7]: class phone:
    def __init__(self, name, owner, memory):
        self.name = name
        self.owner = owner
        self.memory = memory

    def __repr__(self):
        return f'phone("{self.name}", "{self.owner}", {self.memory})'

    def __lt__(self, other_phone):
        return self.memory < other_phone.memory

    def __eq__(self, other_phone):
        return self.name == other_phone.name
```

```
In [8]: iphone = phone('iPhoneIX', 'nikki', 800)
samsung = phone('SamsungEarth', 'nikki', 1600)
print(f'is the iphone less than the samsung (memory): {iphone < samsung}')
print(f'is the iphone less than the samsung (memory): {samsung < iphone}')
print(f'is the iphone less than the samsung (memory): {iphone > samsung}')
print(f'are the two phones the same (name): {iphone == samsung}')
```

```
is the iphone less than the samsung (memory): True
is the iphone less than the samsung (memory): False
```


is the iphone less than the samsung (memory): False
are the two phones the same (name): False

Exceptions

- Python class that represents Errors
- Triggered by any event that disrupts normal program flow
- utilizes `raise` keyword

Common Exception Types and Cases

- **KeyError** - related to dictionaries; attempted access using a key not present in the object
- **IndexError** - related to lists/strings; attempted access of an index that's out of range
- **TypeError** - attempt to unify non - matching data types (ex. str + int) or attempt to access unknown attribute of datatype
- **FileNotFoundError** - related to files; attempted to open a file name that can't be found

raise

python keyword to throw an error.

syntax: raise [exception class](error message // optional)

```
In [2]: def bar_entry(age):  
        if age < 21:  
            raise ValueError("too young to drink...")  
        bar_entry(18)
```

```
-----  
-----  
ValueError                                Traceback (most recent  
call last)  
Cell In[2], line 4  
      2     if age < 21:  
      3         raise ValueError("too young to drink...")  
----> 4 bar_entry(18)  
  
Cell In[2], line 3, in bar_entry(age)  
      1 def bar_entry(age):  
      2     if age < 21:  
----> 3         raise ValueError("too young to drink...")  
  
ValueError: too young to drink...
```

try except else finally

- Exceptions can be handled in `try-except` blocks, preventing code from terminating the moment an error happens.
- `Except` generally handles specific error classes (ex. `except zerodivisionerror`)
- just like with files, it's common to specify a handle for easier processing (`as e`)
- `else` is code that's ran in the case of no errors occurring
- `finally` is code that will always run

```
In [3]: def attempt_bar_entry(age):  
        try:  
            bar_entry(age)  
        except ValueError as e:  
            print(e)  
        else:  
            print('welcome in!')  
        finally:  
            print('-----')  
  
        attempt_bar_entry(18)  
        attempt_bar_entry(21)
```

too young to drink...

welcome in!

try-except vs assert

Assert statements are used to validate inputs and prevent logical errors. Try-except is used to catch error generating inputs / code.

Ex. Asserts can't ascertain that a filepath exists, while try-except can prevent errors like this from happening

is VS ==

- The `==` operator is used to check the equality of the values of two objects
- The `is` operator is used to check if two variables refer to the same object in memory

tl;dr: `==` checks for equality of values, while `is` checks for identity

In-place vs Not In-place Algorithms

- **In-place algorithms** transform data without needing extra space, or by using a minimal amount of additional space. The operation happens on the original data structure rather than a copy.
- **Not in-place algorithms** require additional space proportional to the input data size and create a copy of the data.

```
In [5]: lst = [1,3,2]
        print(sorted(lst))
        print(lst)
```

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

```
In [6]: lst.sort()
        print(lst)
```

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