CSC 461 Quiz #2

Study Guide

Mary Moore

Basic Python Code

- overview
    - scripting language
    - partially interpreted like java
    - has a REPL
- read-eval-print loop (REPL)
    - partially compiles as it goes
    - reuses the compiled stuff when possible
    - slow the first run, but no wait for compilation
    - faster the second time due to partial compilation
- generation:
    - higher level
- paradigm:
    - procedural
    - object-oriented
    - functional
- location:
    - everywhere
- application:
    - web
    - stats
    - scripting
        - text manipulation
        - gluing programs together
        - prototyping
- philosophy
    - “small & simple is beautiful”
    - free, open source, cross platform
    - highly extensible & modular
    - multiparadigm programming language with support for procedural, object-oriented, and even functional
    - originally meant to encourage good programming practices → but irl python programmers are very sloppy
        - ex. forces consistent indentation
    - does not crash easily
        - geared towards dynamic typing
        - great for odd input common to any user??
- general purpose programming language
    - originally intended as a teaching language
- green flags 🟢
    - high level of abstraction
        - can be an oop language if it tries

    - expressive & powerful
    - simple syntax ⇒ ez to learn & use
        - consistent operators
    - small, portable, cross platform, free
    - support for many domains
        - database, text processing, scientific & numeric apps, graphics, guis, web apps
- red flags 🚩
    - less efficient than c++
    - higher level language
        - greater abstraction from the underlying hardware
    - not fully compiled
        - however, high performance modules can be written in other languages like c
- beige flags 🟫
    - dynamic typing
        - interpreter cannot automatically detect all type errors
        - however, dynamic typing makes it much easier to write general purpose routines
        - moves type check responsibility to the programmer
    - garbage collection
        - handles clean dynamic memory location for you 🥳
        - tends to hit at the worst time 😭

Python Language Concepts

- dynamic type binding
    - c++

```
int x; // x is always an int
```

    - python

```
x = 1 # x is an in
x = "test" # x is now a string!
```

- use a backslash to continue a statement over multiple lines or a semicolon to separate multiple statements on the same line
    - but both are bad and forbidden from use in this class 🙁
- match
    - like switch but better, can match any data type (very slow)

Python Specialities

- yield
    - optional return → returns a generator
    - code will pick back up where it left off when the function is next called
- closure
    - can save a block and when it is called, it has access to all the variables in whatever block called it (not made it)

Questions:

1. **What does yield do in Python?**
   it pauses the function and optionally returns a value. when the function is called again, it restarts right after the yield

2. **Ask the user for an integer. Then, write a for loop in Python that asks for that many doubles. Lastly, output the array's values divided by 2, using integer division, on the same line separated with a space.**

```
x = [0] * 5
n = int(input("count: "))
for i in range(n):
    y = float(input("number: "))
    x[i] = y//2

for i in range(n):
    print(x[i], end=" ")
```

Regex

to test: regex101

- pattern matching language for text
- if there are no special characters it matches exactly
- most common special characters
  - . → matches any character except a newline
    - h.t ⇒ hat, hot, hit
  - ^ → matches the start of the string
    - ^h.t ("hat hot hit") ⇒ hat
  - $ → matches the end of the string
  - * → causes the resulting regex to match 0+ repetitions of the preceding regex
    - a* ⇒ a, aa, aaaa, aaa, null
  - + → causes the resulting regex to match 1+ repetitions of the preceding regex
  - ? → causes the resulting regex to match 0 or 1 repetitions of the preceding regex
    - dogs? ⇒ dog, dogs
  - {m} → specifies that exactly m copies of the previous regex should be matched
    - a{2} ⇒ aa
  - {m, n} → specifies that m to n copies of the previous regex should be matched (append ? to get fewest repetitions)
    - a{2, 4} ⇒ aa, aaa, aaaa
    - stype=".*?" ⇒ style="a", but NOT style="a"a"
  - \ → either escapes special characters (ex. *, ?, $) or signals a special character class (ex. \d ⇒ any digit)
    - * ⇒ *

- | → match EITHER pattern
    - hat|hit ⇒ hat, hit
- (...) → matches the pattern inside the parentheses and makes it a ‘group’. the contents of a group can be retrieved later with the “\number” in sequence
    - (h|g)it ⇒ hit, git
- [] → used to indicate a set of characters, but special characters loose their meanings (ex. [+*] matches these characters: + *), character classes such as \d are accepted inside the set as normal, use ^ to negate the set (ex. [^5] will match any character except ‘5’), and - between two regex means range
- \s → any whitespace character (\S negates the meaning)
- \w → any word character (\W negates the meaning) (ex. [a-zA-Z0-9])
- \b → word boundary (\B negates the meaning)
- \d → digit
- r’...’ → raw string, one we don’t have to add \s to all special characters

Questions:

1. **Write the regular expression for 0 or more a’s followed by 1 or 2 b’s.**
   a*b{1,2}

Syntax Choices

- syntax concepts
    - portability
        - write once, run everywhere
        - Java & C are both portable, but for very different reasons
        - compile v.s. interpret is the BIGGEST factor for portability
        - interpretive languages “died” in the middle years b/c they were too “slow” ⇒ now, compiled is too slow for us (the programmers)
        - compiler:
            - translates ALL to lower level languages
            - src program → compiler → assembly language → assembler → machine code
        - interpreter:
            - translates line by line
            - read, eval, print loop ⇒ REPL
            - slow, but much easier to test out code snippets
            - src program → interpreter → output
              input →
              OR
              src program → translator → intermediate program → vm→ output
              input →
        - C is portable b/c it’s SMALL
            - only 20 keywords
    - simplicity

        - simple & ez to learn
        - too simple & things can be hard to write
    - orthogonality
        - the syntax means the same thing (consistent) for ALL fundamental constants
    - expressivity
        - how many ways to do the same thing?
        - orthogonality & expressivity *normally* affect each other
    - reliability
        - less reliability = more flexibility, but more stuff for programmer to check
        - how does the languages handle something going wrong?
            - exception handling
- syntax
    - needs to be context free
- semantic
    - adds meaning, may or may not be grammatically correct
- legal syntax v.s. nonsense semantics
- language
    - set of strings, operators, and numbers that are legal
    - always have an alphabet (usually made up of letters, numbers, and some math chars/punctuation)
- token
    - smallest meaningful unit/category
- lexem
    - an instance of a token
- lexical analysis
    - process of breaking up code into the parts the compiler actually uses
        1. strip whitespace/comments ⇒ figures out what the tokens are
        2. puts tokens into a parse tree ⇒ checks for correctness & fails if there is a lexical error
        3. checks against context-free grammar (syntax)
- context-free grammar, a.k.a. syntax!
    - operation order
        - infix → 1 + 2
        - postfix → 1 2 +
        - prefix → + 1 2
    - assignment location
        - x = 1+2
        - 1+2 = x
    - some means of specifying function/methods
    - some means of designating blocks
        - if{} → C
        - if   → Ruby
          …
          end

        - if → Python
          …
    - some naming scheme
        - be all letters
        - start with a letter
        - start with a $
    - set of recursive rules
        - concatenation
        - selection
        - repetition
        - recursion ⇒ w/o recursion it's still regex, but recursion is required for context-free grammar…
- multiple forms to describe languages
    - BNF: Backus Naur Form
    - EBNF: extended BNF
    - syntax diagram (flowchart)

Questions:

1. **The grammar or context-free part of a language is called what?**
   Syntax

2. **The meaning of a language is called what?**
   Semantics

Binding

- lifetime
    - whether an item is in memory
- scope
    - "access"
    - local or global
- static
    - happens before runtime
- dynamic
    - happens after runtime
    - a.k.a. heap allocation (new keyword in c++)
- 7 bindings:
    - language time
        - for
    - implementation time
        - int → 32 bit
    - program time
        - class ABC
    - compile time

        - const
        - static
    - link time
        - #include <>
    - load time
        - choose memory location
        - static
    - run time
        - dynamic
        - new
        - stack

Scoping

- scope
    - the state where the identifier can be referenced
- why scoping matters
    - modularity → multiple modules with the same name…
    - access levels → group into modules/packages/namespaces
- reference environment
    - visible names at a given environment
    - like all the *current* variables from the outermost nesting to the current block in c++/java
    - closures, shallow binding, and deep binding wreak havoc on this assumption
- shallow binding
    - similar to lazy initialization
    - reference environment determined @ line (call time)
    - more typical in dynamic scoping
- deep binding
    - reference environment @ creation (when the function is used as an argument)
    - common in static scoping b/c permanent access
- closures
    - deep binding, but more 🙂
    - saved reference environment that may or may not be the current one, along with the function that uses it
    - only occurs when you can return a function within a function
    - VERY important w/nested functions

```
func A(x):
    func B():
        print(x)
    return B
```

`c = A(2)` → `x = 2`, creates a R.E. of what was accessible by B when it was made

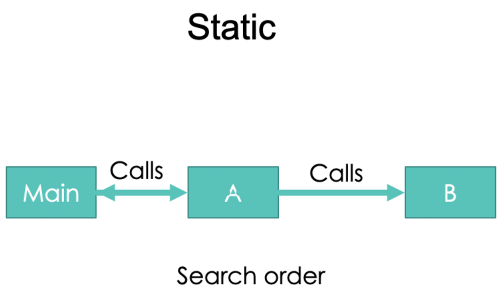
    - remember's what was enclosing it @ the time of creation
- static/dynamic scoping & deep/shallow binding can be in any combo
- first class object → for closures, fns need to be first class
    - can be assigned a variable
    - can be passed into a function
    - can be returned from a function
    - regularly occur in dynamic languages
- second class object
    - can be assigned a variable
    - can be passed into a function
- third class object
    - it exists
    - can not even be passed in 😢
    - syntax level → like for loops, keywords, etc.
- closure & nesting
    - as soon as we have a nested function, the closure tells us what it can access
    - MOST languages also support passing functions in some form
    - who has access to what in a nested function?
        - just the function
            - static scope + shallow bind
        - all variables in an outer block (python)
            - static scope + deep bind
        - all variables from the caller when the function was made
            - dynamic scope + deep bind
        - everything made afterwards? (everything in current call chain)
            - dynamic scoping + shallow binding
    - closure issues are less common in languages that forbid nested functions, but… can still have the same effect with interfaces, operator overloading, and lambda expressions
- shadowing
    - hiding the outer variable on redeclaration
    - definition v.s. declaration of variables
        - declaration (no value) → int x
        - definition → x = 2
    - nested scope (shadowing)

```
{
    int x = 2
    {
        int x = 3 # completely legal syntax & this x overshadows the outer
x
        print(x) # print's 3
    }
}
```

    - in general is very bad form, hay some situations where it is necessary tho
        - access of the outer block's x varies by language

Static

- "normal code"
- determined by placement of variables at or before compile time
- modern systems:
    - access is granted as long as they're in the same block
- ye olden days:
    - *had* to declare & define everything before it was used
- path the compiler checks:

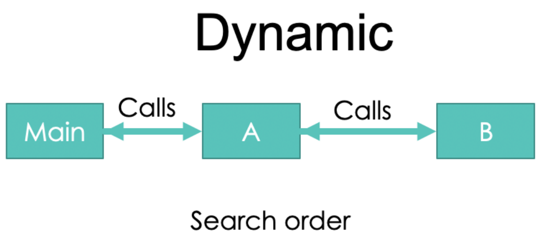


Dynamic

- determined by order of run after compiled
- sequence of fn calls to determine scope
    - B can see everything prior that's on the call stack
    - can access if the variable is currently "active"

```
func A():
    int x
    B()
func B():
    x = 2
```

- path the compiler checks:



- dynamic scope is tricky and closures are favored in most modern languages

Questions:

1. **Name an instance where order within a block matters. Name an instance where it does not.**
   In c++, if there is not a header file with method prototypes, the methods need to be called in order (main needs to be at the bottom of the file). Declaration and definition was the solution for the ordering issue.
   In java classes, order within a block does not matter at all.

2. **With the following pseudocode, what happens with a negative or a positive input with static scope? What about a negative or positive input with dynamic scope?**

```
int n
func first()
    n = 1
func second()
    int n
    first()
n = 2
x = input()
if x > 0: second()
else: first()
print(n)
```

   Static scope will print 1, whether the input is positive or negative. (b/c the n in first() will be mapped to the global n regardless)
   Dynamic scope will print 2 if the input is positive, but 1 if it is negative. (b/c second() has the ability to change what n it is)

3. **What does this print? (assume dynamic scoping with closure)**

```
def A(x, P):
    def B():
        print(x)
    if x  > 1: P()
    else: A(2, B)

    def C():
        pass # could also pass a null to make the compiler happy

A(1, C)
```

   prints 1 b/c when A(1, C) is called, the current reference environment is saved for C, so when C is called, x will be on the stack as 1

4. **What does the following print, assume dynamic scoping with closure? What about without closure?**

```
func A(P):
    int x = 1
    P()
func C(x):
    int y = 2
    func B():
        int z = 3
        print(x, y, z)
    A(B)

C(0)
```

dynamic w/closure: 0, 2, 3
dynamic w/o closure: 1, 2, 3

5. **Consider the following pseudocode, assuming nested functions and static scope. What does this program print?**

```
int g
func B(int a):
    int x
    func A(int n):
        g = n
    func R(int m):
        print(x)
        x = x/2 # integer division
        if x > 1: R(m+1)
        else: A(m)
    # back in B
    x = a*a
    R(1)
# back in main
B(3)
print(g)
```

9 4 2 3

6. **Consider the following pseudo code.**

```
x = 1
y = 2
func add():
    x = x+ y
func second(m):
    local x = 2
    m()
```

```
func first()
    local y = 3
    second(add)
first()
print(x)
```

a. **What does this program print if the language uses static scoping?**
   3

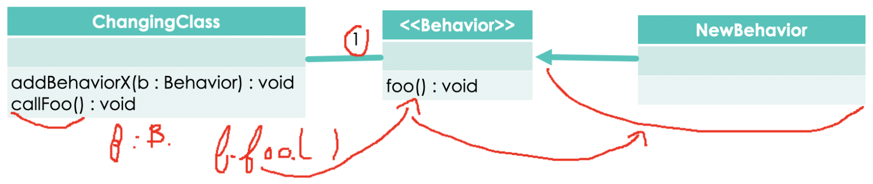
b. **What does it print if the language uses dynamic scoping with deep binding (closure-based dynamic scoping)?**
   5

c. **What does it print if the language uses dynamic scoping with shallow binding (standard dynamic scoping)?**
   1

Strategy Pattern

- swap out individual pieces of functionality
- competes with…
  - general hierarchy
  - visitor pattern
- strategy v.s. visitor
  - visitor
    - does an operation
    - can change over time
    - is type specific & type changes
    - can interact w/multiple classes
  - strategy
    - does an operation
    - can change over time
    - type of class is generally known
    - affects only one class
    - task is reused or changes over time
- strategy v.s. general inheritance
  - technically, you could just derive a class every time you needed a new function
  - however, this can lead to a ‘ridiculous’ inheritance tree & *more* repetition in inheritance than if using the strategy pattern
  - key to strategy is that a behavior is reused across classes
- strategy & d.i. are closely related
  - some argue that one is part of the other or vice versa
- why strategy pattern?
  - helps code reuse

    - SOLID
        - O (open-closed)
        - I (interface segregation)
        - but all the others too if you try hard enough
    - better allows future behavior changes
- *must haves* for when to use strategy
    - you know the type (or at least a parent with the known behavior)
    - there is an operation
    - the operation ‘belongs’ to the instance (not an operation ON the instance)
    - the operation/task changes over time (multiple modes) OR the operation/task is reused across different classes/instances (cleaner workaround for multiple inheritance)
- how does the strategy pattern work?
    - pass the behavior to the instance rather than code it in the class
    - if functions are first class obj ⇒ pass in a function
    - otherwise ⇒ use inheritance!
- basic structure
    - place to store the behavior
        - interface or nested class if first class functions are allowed
        - need to guarantee the availability of a function
    - place to put it
    - make a concrete version & call it
        - the link is to the *interface* this time
    - foo() will be overridden in the NewBehavior class since it inherits from Behavior



Questions:

1. **Select all of the following that should use a strategy pattern rather than inheritance or the visitor pattern:**
    - ☑ ~~**a plant blooms only in a given season which can change with the hemisphere. its blooming method is dependent on the specific flower object**~~
    - ☐ **all plants grow**
    - ☑ ~~**a car engine processes fuel a certain way, and this can depend on the vehicle and can change depending on how the vehicle is driven**~~
    - ☐ **you have several dogs all with their own size**

    plants growing all have the same function & the function remains the same for all instances of the class so inheritance works. dog size should be an instance variable.
    blooming changes with the individual plant, so inheritance doesn’t work. fuel process also changes with the individual vehicle, so inheritance doesn’t work either

Iterator Pattern

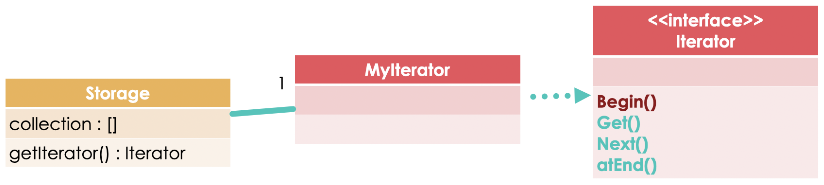
- support for a for-each loop
- 4 tasks:
    - points to a location
    - moves the pointer
    - returns the obj @ current location
    - signals when it is done
- always has a `begin()` and either `end()` or `hasNext()`
    - `begin()` → first element
    - `end()` → element *right after* last element
    - `hasNext()` → true/false 🙂
- also have a `get()` and/or `next()`
    - `get()` → returns an element
    - `next()` → returns the location of the next element
- why use the iterator pattern??
    - for-each loops!!
    - allows the collection storage implementation to vary without affecting the class's use (yay SOLID!)
    - allows more control over how/what is accessed (safety & encapsulation!)
    - SOLID
        - I → interface segregation
        - D → dependency inversion (separate storage from use)
    - to easily traverse a 2D array
    - to only display some elements in a list by a certain criteria
- *must haves* for when to use an iterator
    - there is a collection
    - what to forbid access to the underlying structure OR want to use a for-each loop
- ITERATORS SHOULD NOT EDIT THE COLLECTION
    - make a temporary list to note what to edit later if editing is really necessary, otherwise the program will probs crash 🤷
- iterators & generators are *extremely* close in design
    - a generator makes elements as it goes
    - an iterator pulls elements from a separate collection
- main parts
    - collection
        - or generator
    - iterator
        - initializes everything
        - controls where in the collection we are

The location of the jobs

START of the iterator is in the Iterator class

ACCESS to the iterator is in the collection class

Control is the iterable part



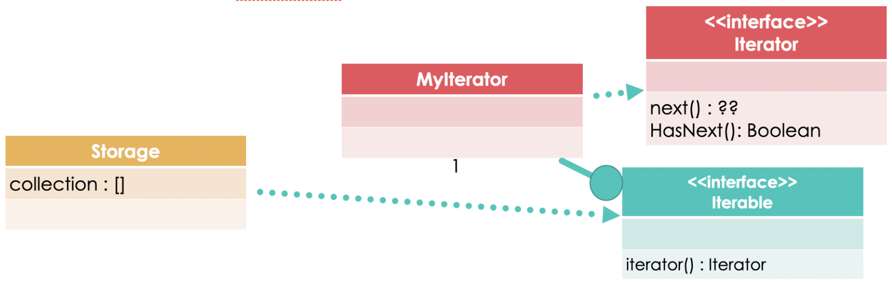
- some languages, like java, split up the iterator further
  - iterable
    - the controller
    - only starts up the iterator & gets everything initialized
  - iterator
    - what actually controls the iterator

This changes the location of the jobs

START of the iterator in the Iterator class

ACCESS to the iterator is in the collection class

Control is the iterable



- encapsulation warning
  - do NOT ever want to allow access to the collection
  - if you ever think you need to return the collection, STOP!
  - if you have many iterators, an outer class is better → cleanliness
    - with just a couple and inner class is good (plus scope benefits in java)

Questions:

1. **How does a generator compare to an iterator?**
   a generator will run forever while an iterator will stop at the end of it's collection

2. **Python allows for easy implementation of the iterator pattern since functions are first class objects. What are the two main parts of the pattern if we do not have first class functions?**
   the interface to place the functions & setting up the main class to accept & call a concrete

instance of the interface

3. **How does an iterator aid in encapsulation?**
   it protects the collection from direct access

4. **Name one SOLID principle the iterator pattern aids in keeping and why.**
   the iterator pattern preserves dependency injection by separating the storage of the collection from its use

5. **Give an example of a good time to add *multiple* iterators to a problem, other than a serial for-each loop over a collection.**
   any class that needs to return a subset of items can benefit from additional iterators, in addition to the simple serial loop for printing\debugging

---

Basic OOP Diagraming

- motto of OOP: "those that know, do!"
  - if a class knows abt something, any functions on that data should be in that class too
- steps:
  - find nouns & adjectives
    - become classes & attributes
  - figure out what needs what
    - become connections & links
  - figure out verbs or tasks
    - become functions
    - adverbs become attributes needed by the tasks
  - diagram it!
  - don't add "what ifs"
- application diagrams will change as you code & realize you missed something
- just a graphical representation of the program, shouldn't include any language specific stuff

class
attributes
functions or descriptions of functions

- class name matches actual class name
- attributes
  - name : data type
  - arrays: name : datatype []
- functions
  - major functions for the class
  - or just a description of the class's general jobs/tasks

    - fn (name, name…) : return type
    - if you KNOW you need it, add the function
- inheritance
    - shown with an arrow
    - interface implementation shown with a dashed arrow
    - interfaces tagged with << interface >>
- connection
    - shown with a line (no arrow) between two classes
    - the role is on the far end of the connection so it reads well in english
    - has-a
        - 1 ⇒ just 1
        - 0:1 ⇒ 0 or 1
        - x:y ⇒ x to y instances
        - * ⇒ many
    - its okay to have more than one connection between the same to classes
    - you MUST have a multiplicity or a very clear role, but having both is better
    - visitors have a use-a relationship with the main program, mark this as a note on the diagram
- add additional notes if the diagram is not sufficient
- public/private/protected is not mentioned
- abstract is not mentioned → requires a note
- don't put a connection line AND a member variable → redundant 🙁
- don't add getters/setters

Type Systems

- strong/weak types
- static/dynamic
- type systems
    - components
- type strength
    - strong → the meaning cannot change (java & python) → (c++ is less strong than java, but more strong than c…)
    - weak → the meaning can change (c & assembly)
- static
    - type doesn't change
- dynamic
    - type can change
- static pros (dynamic cons):
    - compiler checks
    - faster
    - early detection of errors (unit tests)
    - takes less space

- dynamic pros (static cons):
    - input parsing is easier (database)
    - out-of-bounds on data type (overflow) doesn't exist
- c++ ⇒ strong, static
- java ⇒ strong, static
- python ⇒ strong, dynamic

	static	dynamic
strong	c++, java types are set in stone	python, lisp type meanings hold, but can reuse variable names
weak	c, assembly lower level languages	some versions of lisp (very, very rare) → free for all

(no typing: some versions of assembly)

- type system
    - set of typing rules
    - also the types
- type equivalence
    - what happens with the ==
        - 1 == 1.0 ????
- type compatibility
    - casting rules
- type inference
    - guess the type → guess the final type
        - x = 1 + 1.0 ⇒ 2.0?? float or double??
        - auto x = 1 + 1.0 ???
- type checking
    - application of the rules
    - coercion
- three ways to define types:
    - mathematically
        - N e[0-255]
    - implementation
        - 32 bits, which bit defines what
    - range
        - [0-255], $0 \leq i \leq 255$
- ex. types → int:
    - implementation
        - 32 bit 2's complement
    - mathematical
        - n e N & in the range below

- range
    - ±2,000,000,000
- others?
    - null
    - none
    - undefined
    - all a ‘flag’
- aliasing
    - type alias m3 int[][][]
    - DANGEROUS!!!
- equivalence
    - a == b
    - 1 == 1.0
        - true in c++
        - true in php, but 1 === 1.0 is false
    - type equivalence
        - name does not matter
    - data sets
        - if stores as a hash → true
- compatibility
    - coercion
        - foo(int x) → foo(1.0) ??
    - explicit?
    - polymorphism
    - “will it accept it”
- inference
    - “can the compiler figure out the data type for you”
    - auto (c++)
    - it's on the programmer to know the rules 🙁

Questions:

1. **What is type compatibility?**
   a type rule that deals with which types a construct will accept or not

2. **Explain how data types would be used/handled in a language that is *both* weakly typed and dynamically typed. Give an example variable declaration.**
   you would not need to explicitly declare the datatype since it would be determined by the value assigned to it; however, since it is weakly typed, this also means we can cast w/o explicit notation since data types are fluid. since it is dynamic, it is almost guaranteed to have garbage collection as well.
   x = 2
   x = ‘c’ + 2 → will result in ‘e’ or the ascii value of ‘e’

3. **Why do many modern languages include properties rather than only traditional member variables?**
   ease of use. adding a getter/setter is assumed in everything that has to be accessed outside the class, so this simplifies use

---