Lecture 10: Errors:

COMP90059 Introduction to Programming

Wally Smith

School of Computing & Information Systems

Lecture Overview

Exam Preparation

A detailed review of what form the exam will take, looking at the MOCK exam questions on Canvas, and thinking about how best to prepare.

Modular program design - local & global scope

A review of the very important idea of modular program design:

- scales of program code
- local and global scope of variables - looking at this in more detail

Errors and de-bugging the code

A review of the different kinds of errors that can occur in a program, and an exploration of the craft of detecting and fixing errors in our code.

The card game challenge

One solution for a program to deal out random hands of playing cards from a shuffled deck of cards. (From Lecture 7)

Exam Preparation

ASSESSMENT

% contributions to overall grade for the subject are shown for each piece of assessment

Assignment 1 (10%): a set of programming tasks on Grok, due on Friday 24th April .
Mid-Semester Test (0%): takes place on Week 8, at 2 - 3 pm on Monday 4th May. A one-hour test online via Grok in place of the usual Monday lecture. Your score on the test will NOT contribute to your overall grade for the subject, but it will provide feedback on your learning so far.
Assignment 2 (20%): a set of programming tasks on Grok. Released on Wednesday 6th May (during Week 8) , and due on Friday 29th May (Week 11).
Final Examination (70%): held in the examination period after semester (date to be advised). Delivered online in a way to be advised. You will carry out the exam during a specified 3 hours.
Hurdle requirement: 35/70 or greater on the Final Examination is needed to pass the subject.

Please rest assured that the Final Examination is not intended to present fiendishly complex problems. Instead, it will present a mixture of questions that are designed to allow you demonstrate the knowledge and skills you have developed through the lectures, tutes and assignments. The nature of the Final Examination will be discussed in lectures in the latter part of semester, and we will look at many examples of the kinds of questions that will be included.

Programming Concepts and Techniques

COMP90059 Introduction to Programming - Semester 1, 2020

WHAT IS PROGRAMMING?

- comparison with natural languages
- problem analysis, problem decomposition
- modular program design
- algorithms
- code quality: correct syntax, effectiveness, efficiency, readability, traceability, extensibility, elegance
- programming environments: IDLE, the shell, the script
- interpreted vs compiled languages

CODING RULES, CONVENTIONS & STYLE

- variable naming rules
- commenting the code
- reserved words
- PEP8 style guidelines

SCALES OF PROGRAM CODE

- expressions
- statements
- functions
- modules (ie a program script)
- libraries

COMMON BUILT-IN FUNCTIONS

- print()
- input()
- abs()

... more

FORMATTING OUTPUT

- f-string

DATA TYPES & STRUCTURES

- literals
- variables: string, integer, real numbers, boolean
- collections: lists, tuples, sets, dictionaries
- dynamic typing in Python
- type casting: int(), str(), float(), list()
- type identification: type()

CHARACTER OPERATIONS

- escape sequences
- UniCode character set
- ord(), chr()

EXPRESSIONS

- assignment, simultaneous assignment
- strings: concatenation, repetition
- arithmetic expressions: operators and precedence rules (**, * , / , // , % + , -)
- boolean expressions: comparison operators (==, !=, >, <, >=, <=, and, or, not, in)

in: substring in string, item in list, key in dict

not: the inverse of a boolean, eg. 'a' not in 'castle'

SEQUENCE OPERATIONS

- len()
- indexing: string[index], list[index]
- slicing: string[start, stop, step], list[start, stop, step],
- multidimensional indexing & slicing
- string methods: center(), count(), endswith(), find(), isalpha(), isdigit(), join(), lower(), replace(), split(), startswith(), strip(), upper()
- list methods: append(), sort(), reverse(), index(), insert(), count(), remove(), pop()
- dictionary methods: keys(), values(), items(), get(key, <default>), del dict[key], clear()
- min(), max() - for lists

CONTROL FLOW

- default line by line
- conditionals: if, elif, else; nested conditionals
- iteration: for loops, while loops, nested loops
- break

ITERABLES - The things we iterate through

- range(start, stop, step), range(len(string))
- strings
- lists, tuples, dictionaries

DEFINED FUNCTIONS & MODULAR PROGRAM DESIGN

- function definition
- parameters, arguments
- local and global scope
- return
- positional vs keyword arguments
- modular program design using main( )

MUTABILITY

- mutable vs immutable data structures
- aliasing
- mutables as parameters

FILES

- reading from a data file: read(), readline(), readlines()
- write to a data file: print('entry', file =<file_object>)

ERRORS

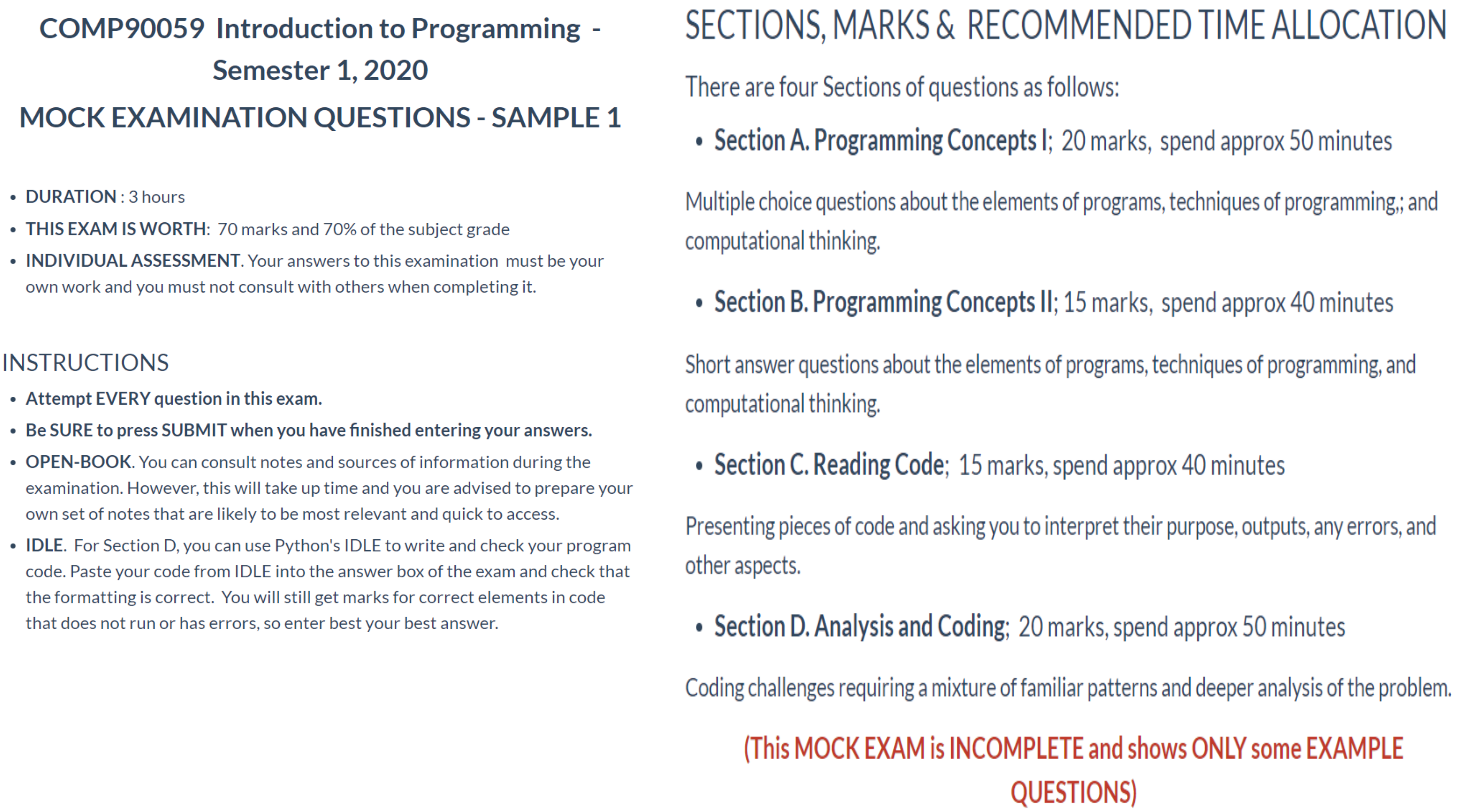
- types of error: syntax, run-time, logic
- debugging techniques

CODING PATTERNS

- sentinel while loop
- building a string
- building a list - using append()
- working through a temporary list - using pop()
- accumulators
- boolean flags
- converting a string to a list - using list() and split()

useful methods

- sort()
- count()
- upper(), lower()
- isalpha(), isdigit()



COMP90059 Introduction to Programming - Semester 1, 2020

MOCK EXAMINATION QUESTIONS - SAMPLE 1

- **DURATION** : 3 hours
- **THIS EXAM IS WORTH**: 70 marks and 70% of the subject grade
- **INDIVIDUAL ASSESSMENT**. Your answers to this examination must be your own work and you must not consult with others when completing it.

INSTRUCTIONS

- **Attempt EVERY question in this exam.**
- **Be SURE to press SUBMIT when you have finished entering your answers.**
- **OPEN-BOOK**. You can consult notes and sources of information during the examination. However, this will take up time and you are advised to prepare your own set of notes that are likely to be most relevant and quick to access.
- **IDLE**. For Section D, you can use Python's IDLE to write and check your program code. Paste your code from IDLE into the answer box of the exam and check that the formatting is correct. You will still get marks for correct elements in code that does not run or has errors, so enter best your best answer.

SECTIONS, MARKS & RECOMMENDED TIME ALLOCATION

There are four Sections of questions as follows:

- **Section A. Programming Concepts I**; 20 marks, spend approx 50 minutes

Multiple choice questions about the elements of programs, techniques of programming,; and computational thinking.

- **Section B. Programming Concepts II**; 15 marks, spend approx 40 minutes

Short answer questions about the elements of programs, techniques of programming, and computational thinking.

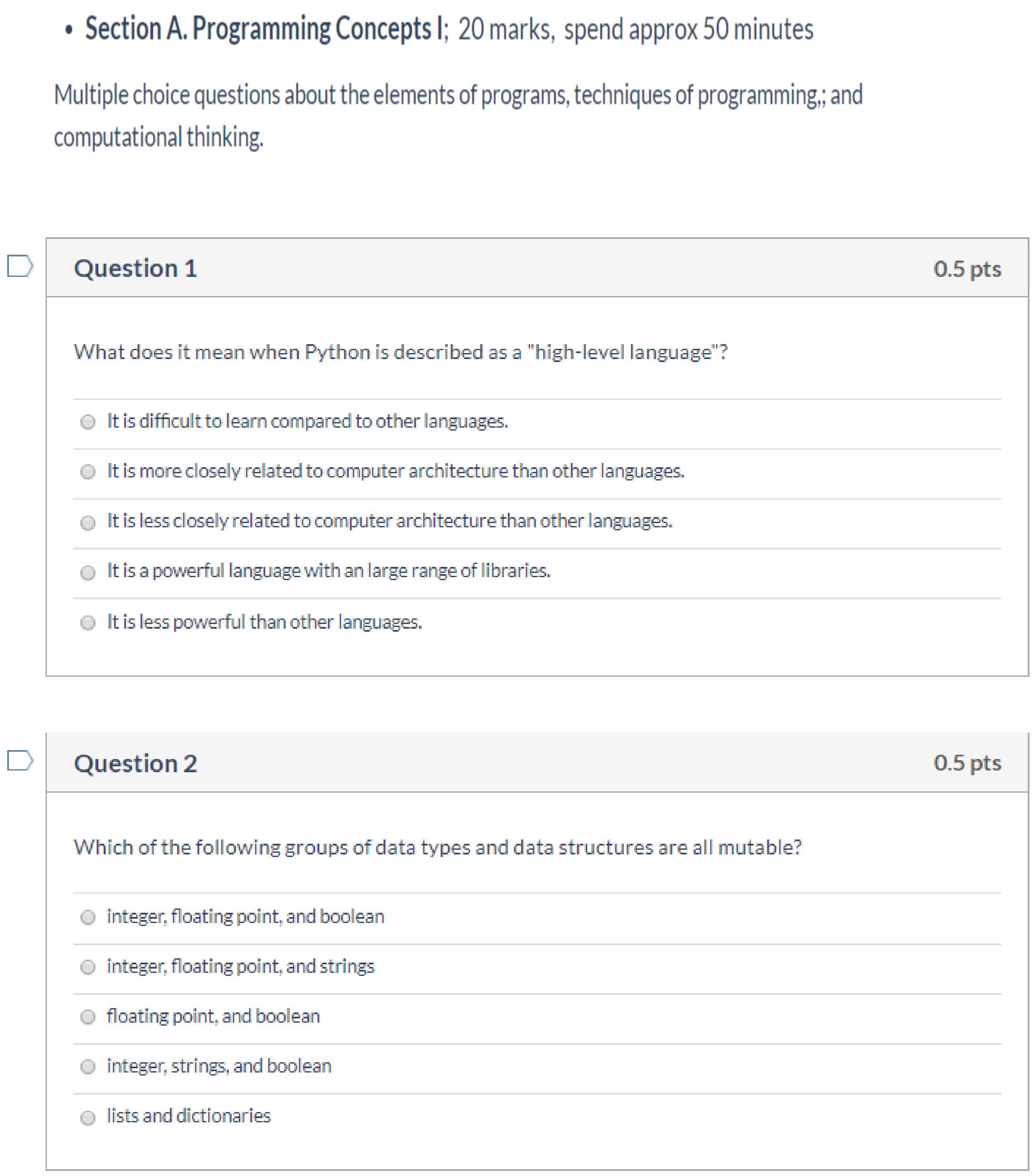
- **Section C. Reading Code**; 15 marks, spend approx 40 minutes

Presenting pieces of code and asking you to interpret their purpose, outputs, any errors, and other aspects.

- **Section D. Analysis and Coding**; 20 marks, spend approx 50 minutes

Coding challenges requiring a mixture of familiar patterns and deeper analysis of the problem.

(This MOCK EXAM is INCOMPLETE and shows ONLY some EXAMPLE QUESTIONS)



- **Section A. Programming Concepts I**; 20 marks, spend approx 50 minutes

Multiple choice questions about the elements of programs, techniques of programming,; and computational thinking.

Question 1

0.5 pts

What does it mean when Python is described as a "high-level language"?

○ It is difficult to learn compared to other languages.

○ It is more closely related to computer architecture than other languages.

○ It is less closely related to computer architecture than other languages.

○ It is a powerful language with an large range of libraries.

○ It is less powerful than other languages.

Question 2

0.5 pts

Which of the following groups of data types and data structures are all mutable?

○ integer, floating point, and boolean

○ integer, floating point, and strings

○ floating point, and boolean

○ integer, strings, and boolean

○ lists and dictionaries

HUMAN (PROGRAMMER)

LECTURE 03

natural language

high-level languages

Python

C, C++, C#, Java

these languages use **abstraction** to get closer to natural language and are easier for us to understand and debug, but less efficient and harder to optimize for the machine

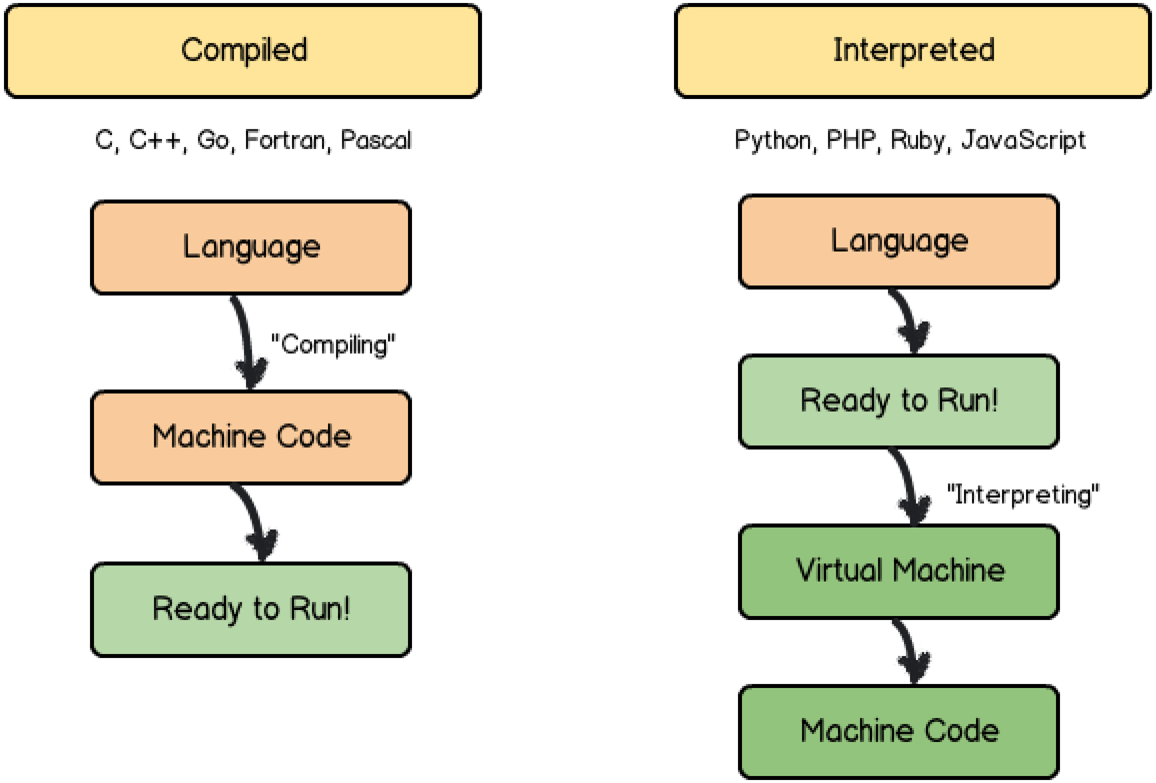
assembly language

machine code

these languages are closer to the architecture of the machine, and are therefore more efficient and powerful to achieve optimisation

binary

COMPUTER



Compiled

C, C++, Go, Fortran, Pascal

Language

"Compiling"

Machine Code

Ready to Run!

Interpreted

Python, PHP, Ruby, JavaScript

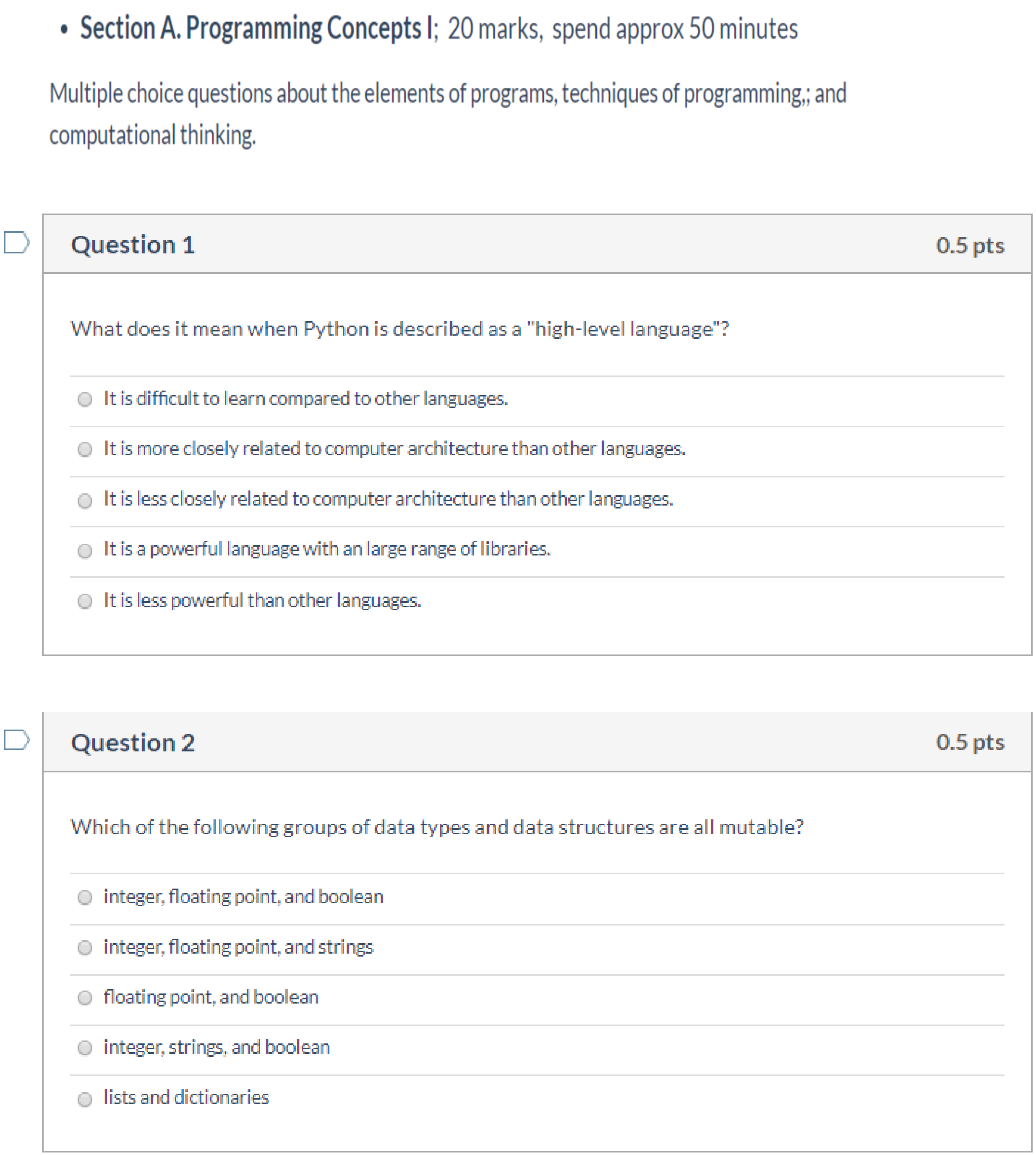
Language

Ready to Run!

"Interpreting"

Virtual Machine

Machine Code



- **Section A. Programming Concepts I**; 20 marks, spend approx 50 minutes

Multiple choice questions about the elements of programs, techniques of programming,; and computational thinking.

Question 1 0.5 pts

What does it mean when Python is described as a "high-level language"?

○ It is difficult to learn compared to other languages.

○ It is more closely related to computer architecture than other languages.

○ It is less closely related to computer architecture than other languages.

○ It is a powerful language with an large range of libraries.

○ It is less powerful than other languages.

LECTURE 03

Question 2 0.5 pts

Which of the following groups of data types and data structures are all mutable?

○ integer, floating point, and boolean

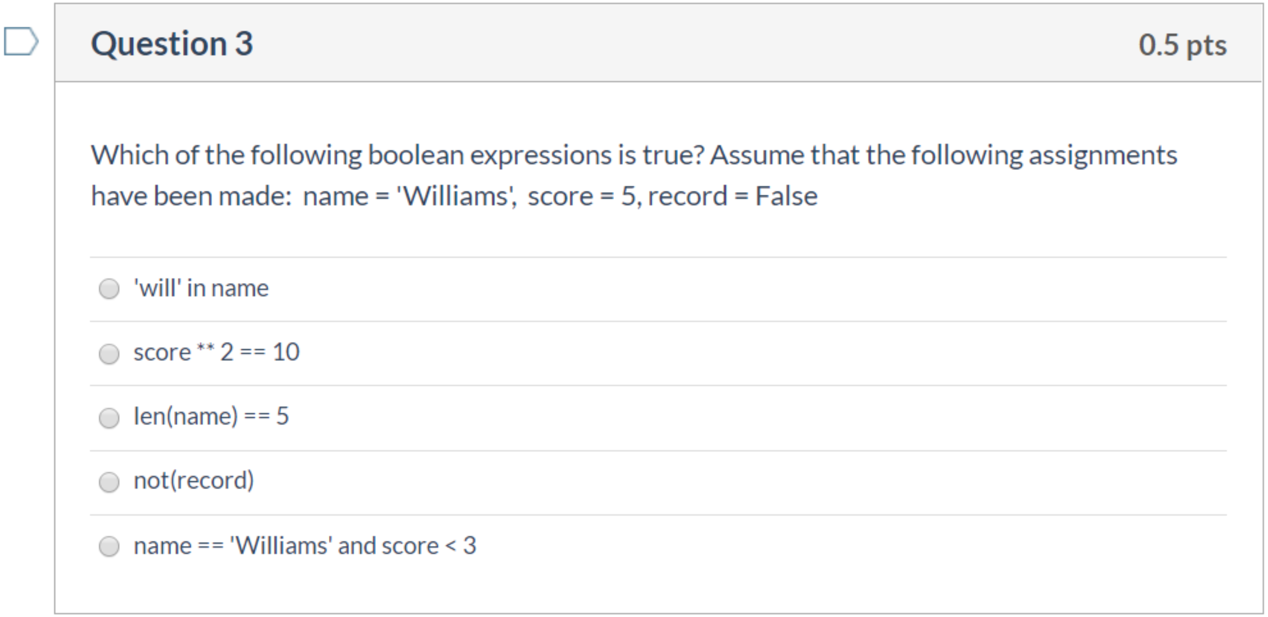
○ integer, floating point, and strings

○ floating point, and boolean

○ integer, strings, and boolean

○ lists and dictionaries

LECTURE 07
Collections:
mutability



Question 3

0.5 pts

Which of the following boolean expressions is true? Assume that the following assignments have been made: name = 'Williams', score = 5, record = False

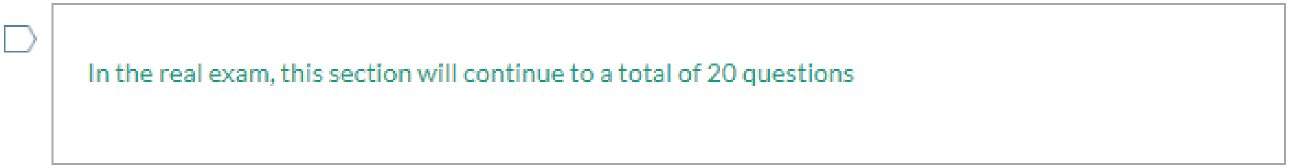
○ 'will' in name

○ score ** 2 == 10

○ len(name) == 5

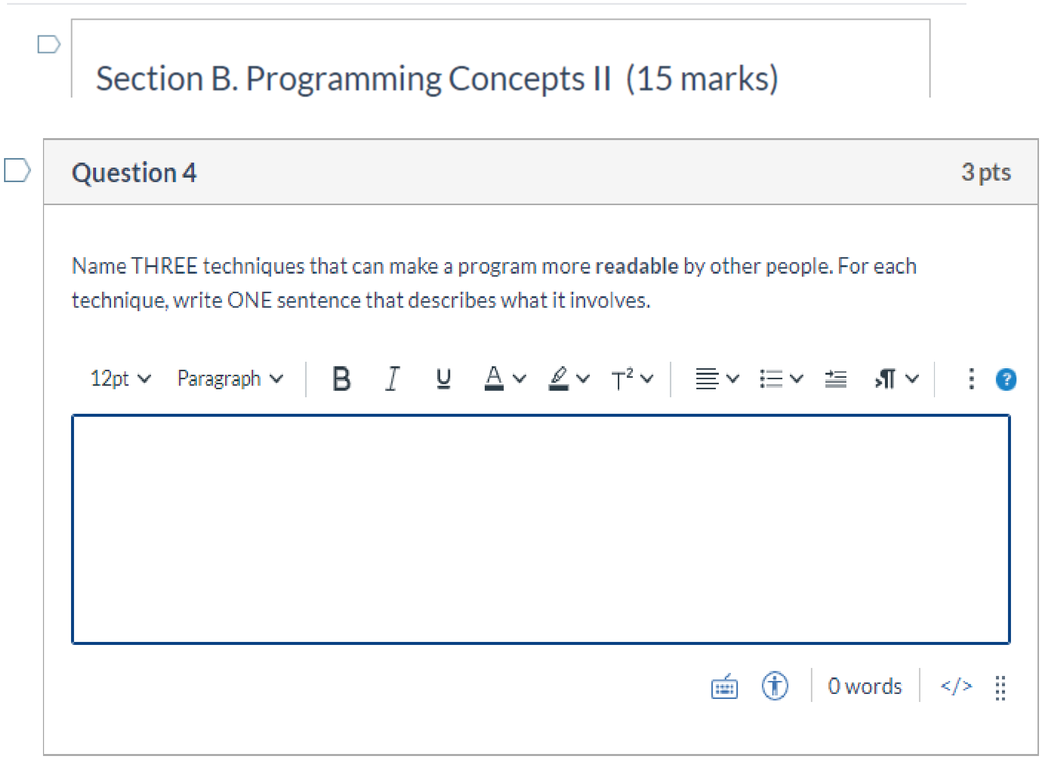
○ not(record)

○ name == 'Williams' and score < 3



In the real exam, this section will continue to a total of 20 questions

LECTURE 03
Conditionals:
Boolean expressions



Section B. Programming Concepts II (15 marks)

Question 4 3 pts

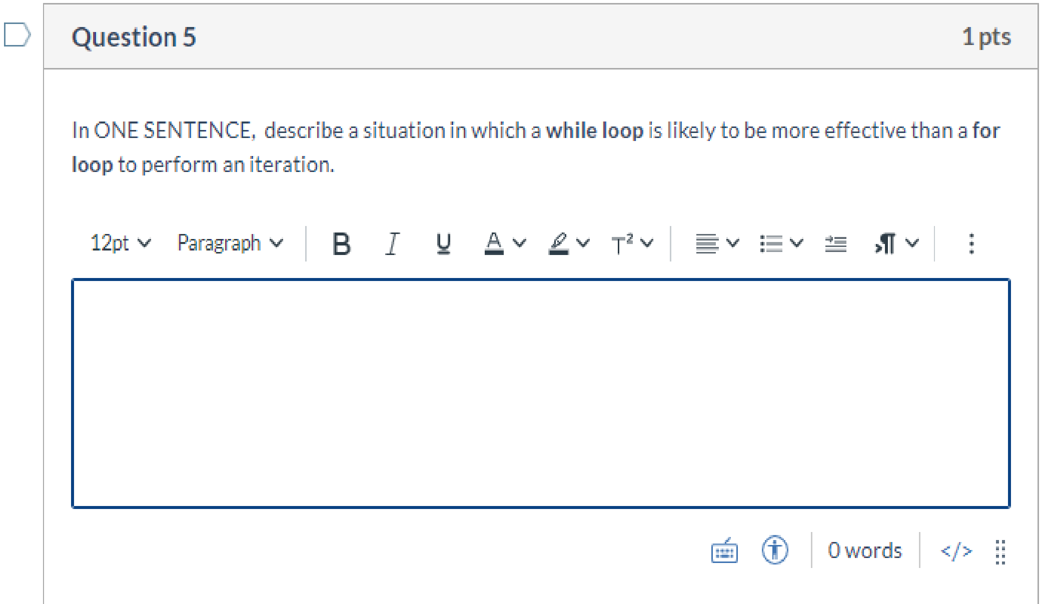
Name THREE techniques that can make a program more **readable** by other people. For each technique, write ONE sentence that describes what it involves.

commenting
Using # to add non-code comments that explain what the code is doing.

meaningful variable names
Choosing names, like 'customers' or 'payments', rather than 'x' or 'p', that are easily understood

modular design through functions
Dividing the code into separate functions, with each function being relatively simple and easily understood.

LECTURE - VARIOUS



Question 5 1 pts

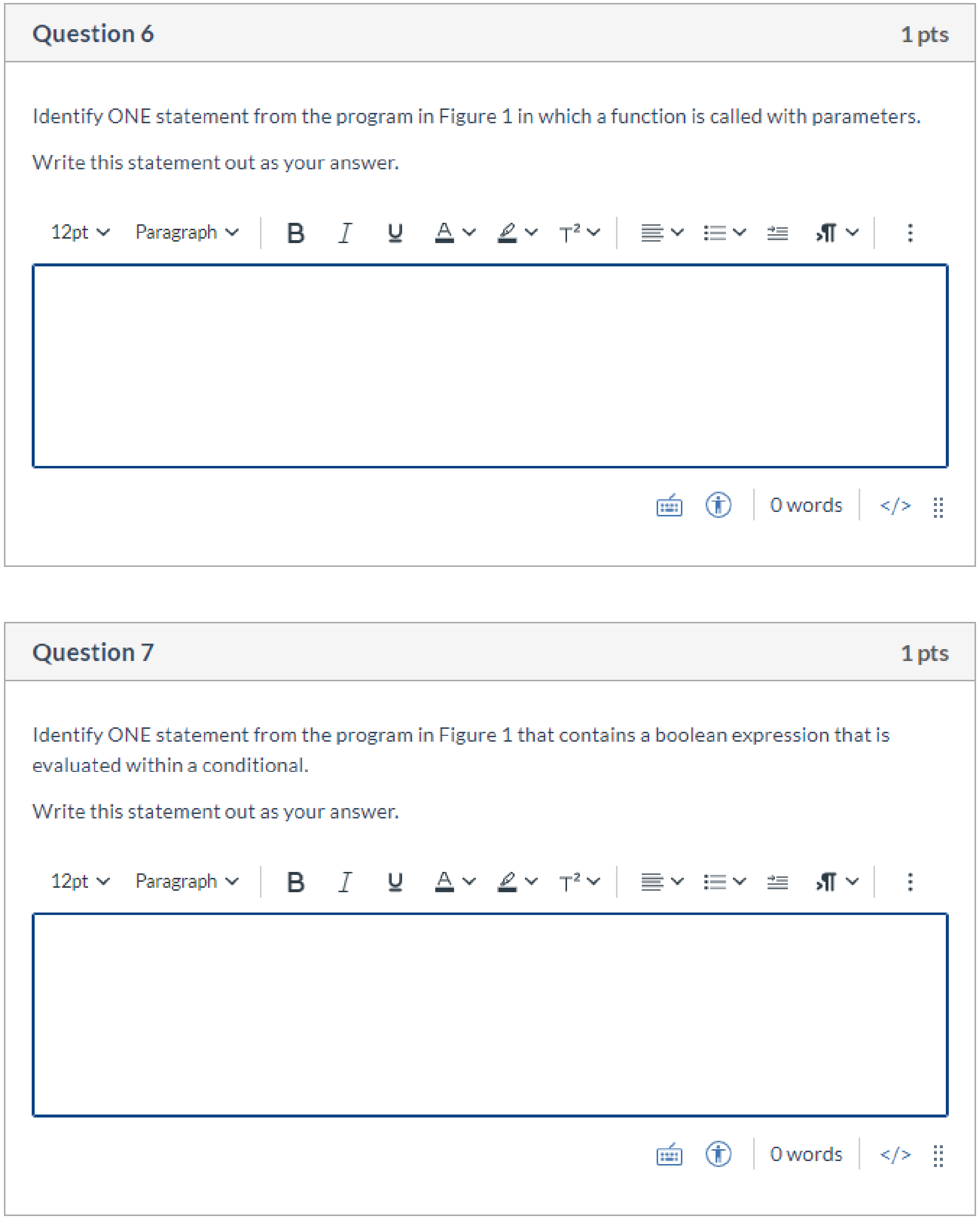
In ONE SENTENCE, describe a situation in which a **while loop** is likely to be more effective than a **for loop** to perform an iteration.

A while loop is likely to be more effective than a for loop when a user enters as many data items as they want, and the programmer doesn't know how many.

LECTURE 03

- **Section C. Reading Code**; 15 marks, spend approx 40 minutes

Presenting pieces of code and asking you to interpret their purpose, outputs, any errors, and



Question 6

1 pts

Identify ONE statement from the program in Figure 1 in which a function is called with parameters.

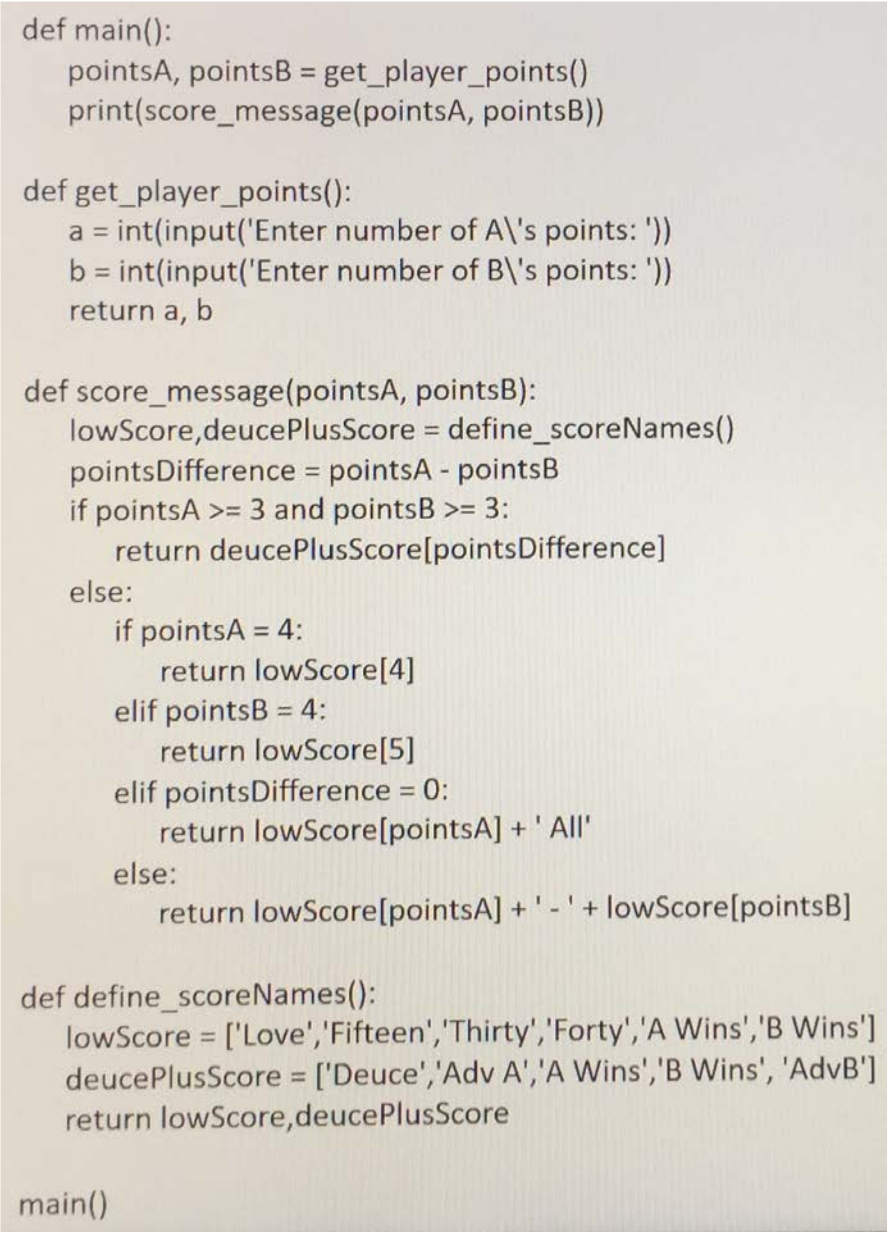
Write this statement out as your answer.

Question 7

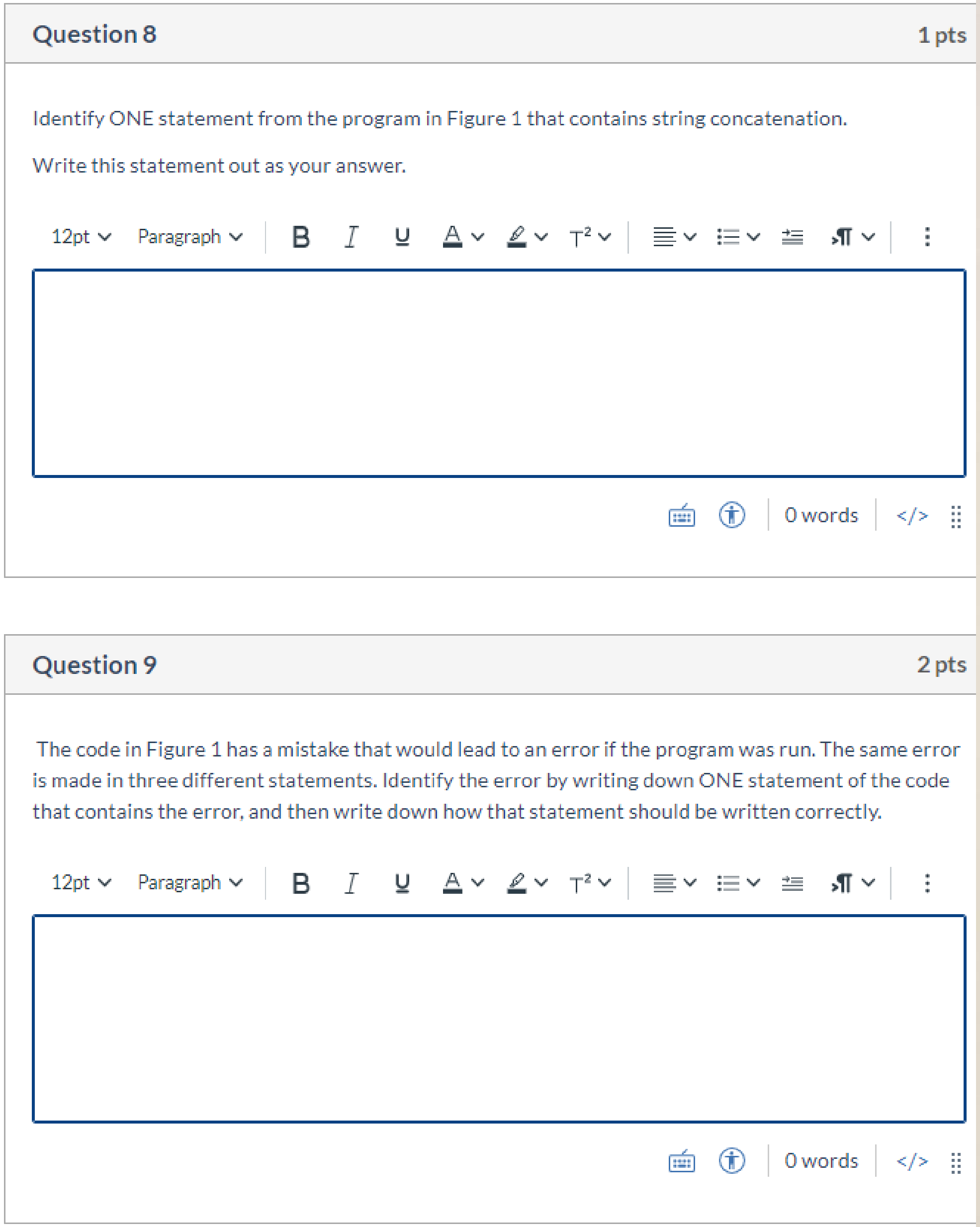
1 pts

Identify ONE statement from the program in Figure 1 that contains a boolean expression that is evaluated within a conditional.

Write this statement out as your answer.



```
def main():
    pointsA, pointsB = get_player_points()
    print(score_message(pointsA, pointsB))

def get_player_points():
    a = int(input('Enter number of A\'s points: '))
    b = int(input('Enter number of B\'s points: '))
    return a, b

def score_message(pointsA, pointsB):
    lowScore,deucePlusScore = define_scoreNames()
    pointsDifference = pointsA - pointsB
    if pointsA >= 3 and pointsB >= 3:
        return deucePlusScore[pointsDifference]
    else:
        if pointsA = 4:
            return lowScore[4]
        elif pointsB = 4:
            return lowScore[5]
        elif pointsDifference = 0:
            return lowScore[pointsA] + ' All'
        else:
            return lowScore[pointsA] + ' - ' + lowScore[pointsB]

def define_scoreNames():
    lowScore = ['Love','Fifteen','Thirty','Forty','A Wins','B Wins']
    deucePlusScore = ['Deuce','Adv A','A Wins','B Wins', 'AdvB']
    return lowScore,deucePlusScore

main()
```



Question 8

1 pts

Identify ONE statement from the program in Figure 1 that contains string concatenation.

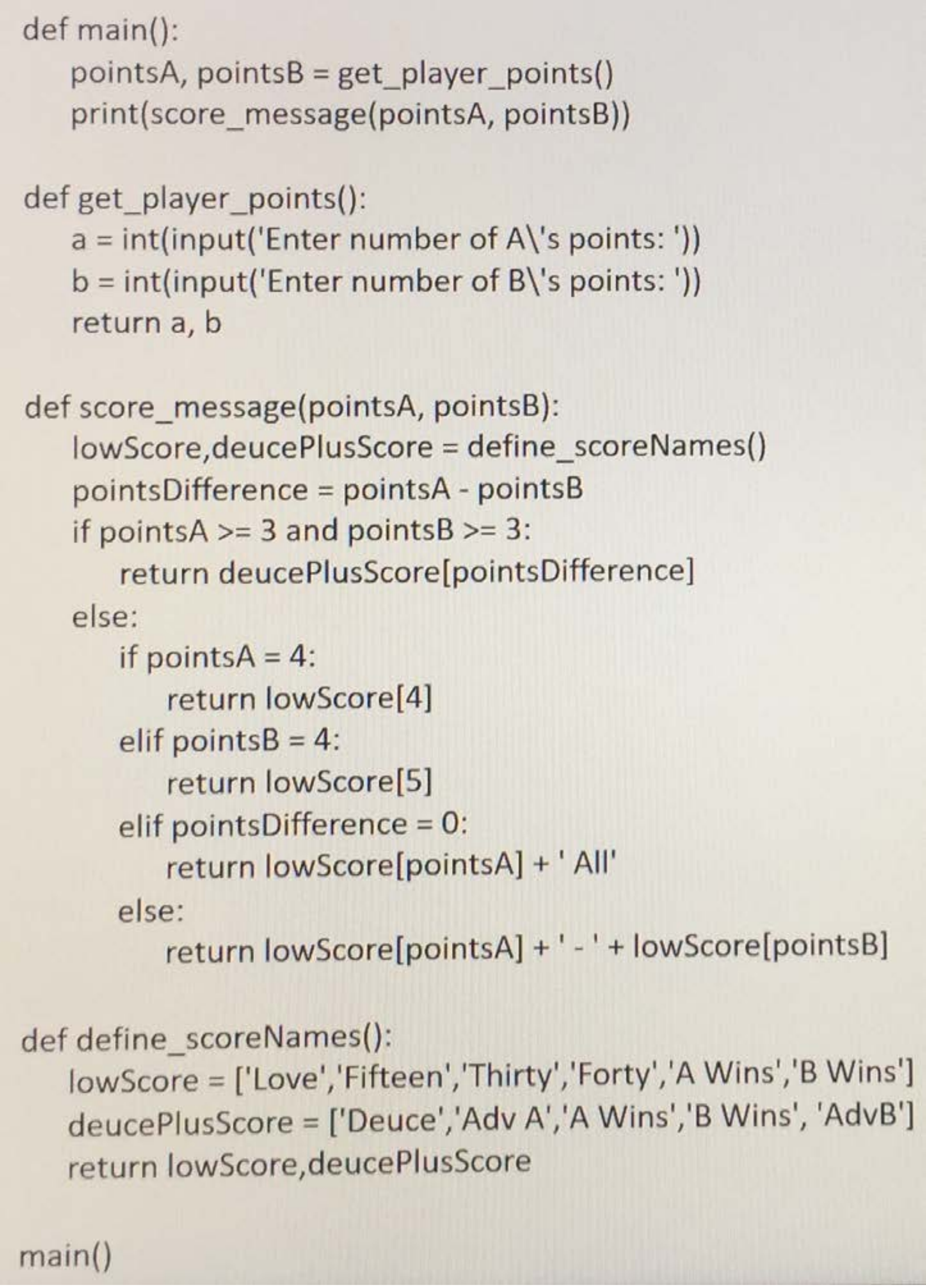
Write this statement out as your answer.

Question 9

2 pts

The code in Figure 1 has a mistake that would lead to an error if the program was run. The same error is made in three different statements. Identify the error by writing down ONE statement of the code that contains the error, and then write down how that statement should be written correctly.

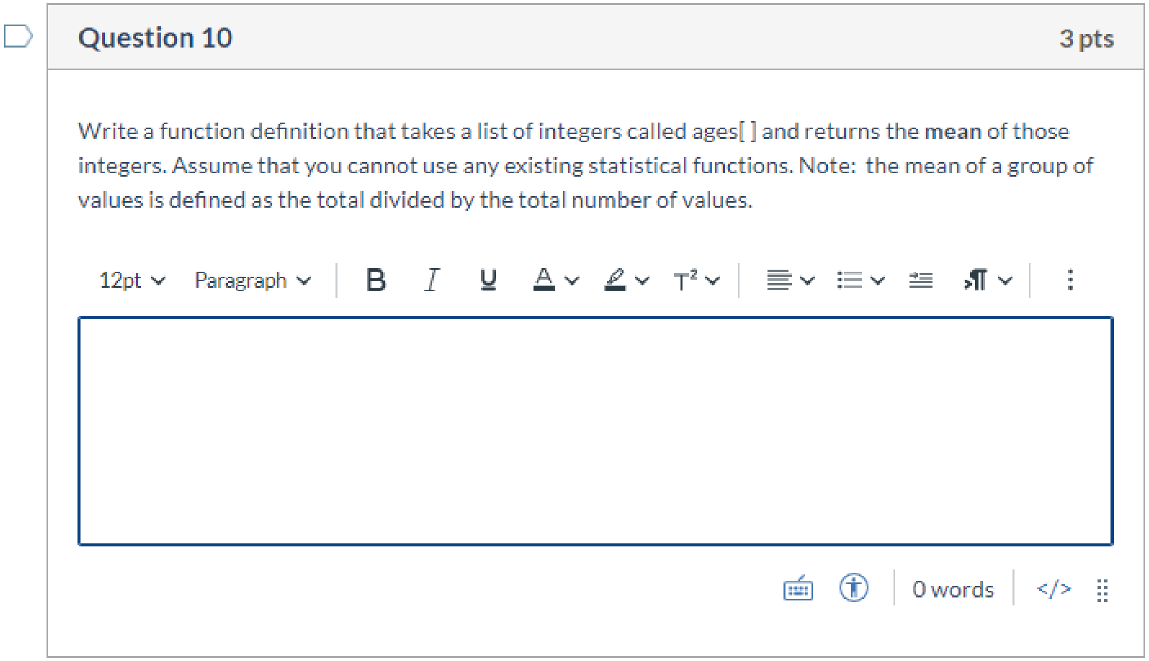
12pt
Paragraph
0 words



```
def main():
    pointsA, pointsB = get_player_points()
    print(score_message(pointsA, pointsB))

def get_player_points():
    a = int(input('Enter number of A\'s points: '))
    b = int(input('Enter number of B\'s points: '))
    return a, b

def score_message(pointsA, pointsB):
    lowScore,deucePlusScore = define_scoreNames()
    pointsDifference = pointsA - pointsB
    if pointsA >= 3 and pointsB >= 3:
        return deucePlusScore[pointsDifference]
    else:
        if pointsA = 4:
            return lowScore[4]
        elif pointsB = 4:
            return lowScore[5]
        elif pointsDifference = 0:
            return lowScore[pointsA] + ' All'
        else:
            return lowScore[pointsA] + ' - ' + lowScore[pointsB]

def define_scoreNames():
    lowScore = ['Love','Fifteen','Thirty','Forty','A Wins','B Wins']
    deucePlusScore = ['Deuce','Adv A','A Wins','B Wins', 'AdvB']
    return lowScore,deucePlusScore

main()
```

- **Section D. Analysis and Coding;** 20 marks, spend approx 50 minutes

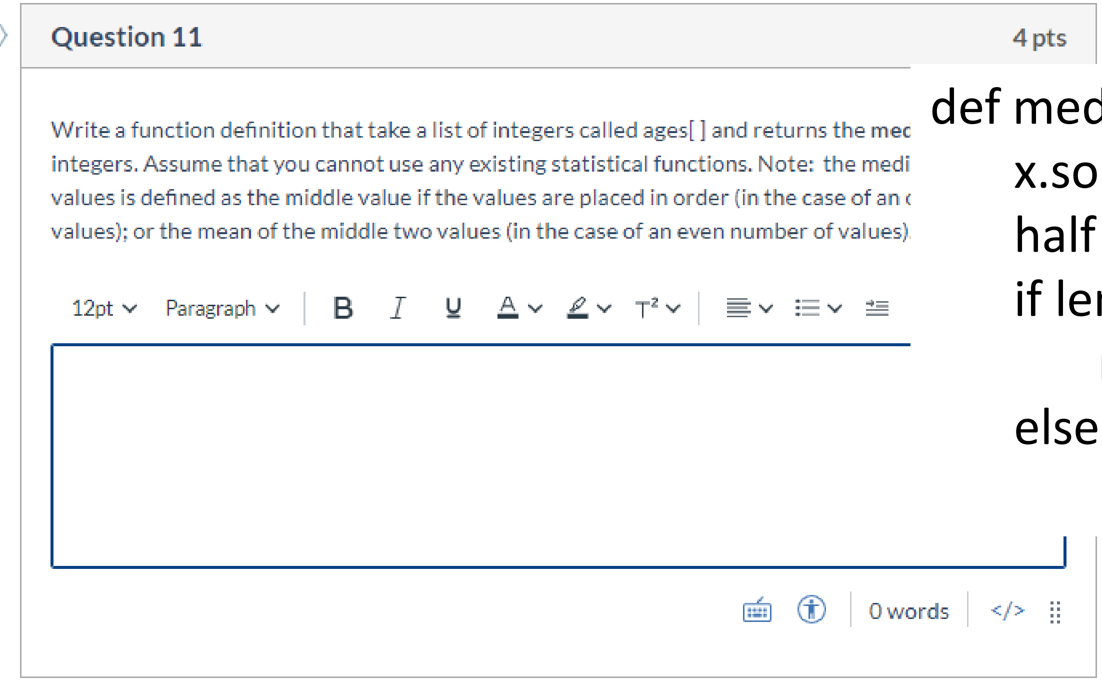
Coding challenges requiring a mixture of familiar patterns and deeper analysis of the problem.



Question 10 3 pts

Write a function definition that takes a list of integers called ages[ ] and returns the **mean** of those integers. Assume that you cannot use any existing statistical functions. Note: the mean of a group of values is defined as the total divided by the total number of values.

```
def mean(x):
    N = len(x)
    total = 0
    for item in x:
        total += item
    return total / N
```

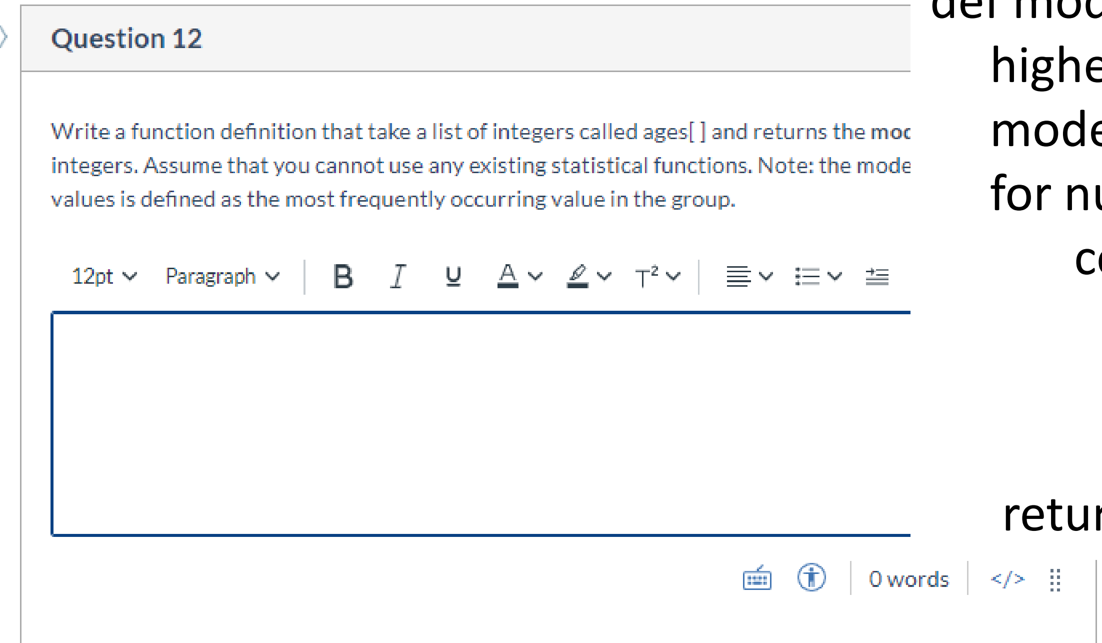


Question 11

4 pts

Write a function definition that take a list of integers called ages[ ] and returns the med integers. Assume that you cannot use any existing statistical functions. Note: the medi values is defined as the middle value if the values are placed in order (in the case of an values); or the mean of the middle two values (in the case of an even number of values).

```
def median(x):
    x.sort()
    half = len(x)//2
    if len(x)%2 == 0:
        return = (x[half] + x[half + 1])/2
    else:
        return =x[half + 1]
```



Question 12

Write a function definition that take a list of integers called ages[ ] and returns the mod integers. Assume that you cannot use any existing statistical functions. Note: the mode values is defined as the most frequently occurring value in the group.

```
def mode(x):
   highestCount = 0
   mode = 0
   for num in range(min(x) to max(x)+1):
       count = x.count(num)
          if count > highestCount:
              highestCount = count
              mode = num
    return mode
```

Modular Program Design

The anatomy of programs

LECTURE 07

- **data structures** of different types**,**

  variables: strings, integers, floating points, booleans,

  collections: lists, tuples, sets, dictionaries

- **functions** - actions to be carried out on data

  - e.g., print(), input(), abs(), ... *and many more*

- **methods**

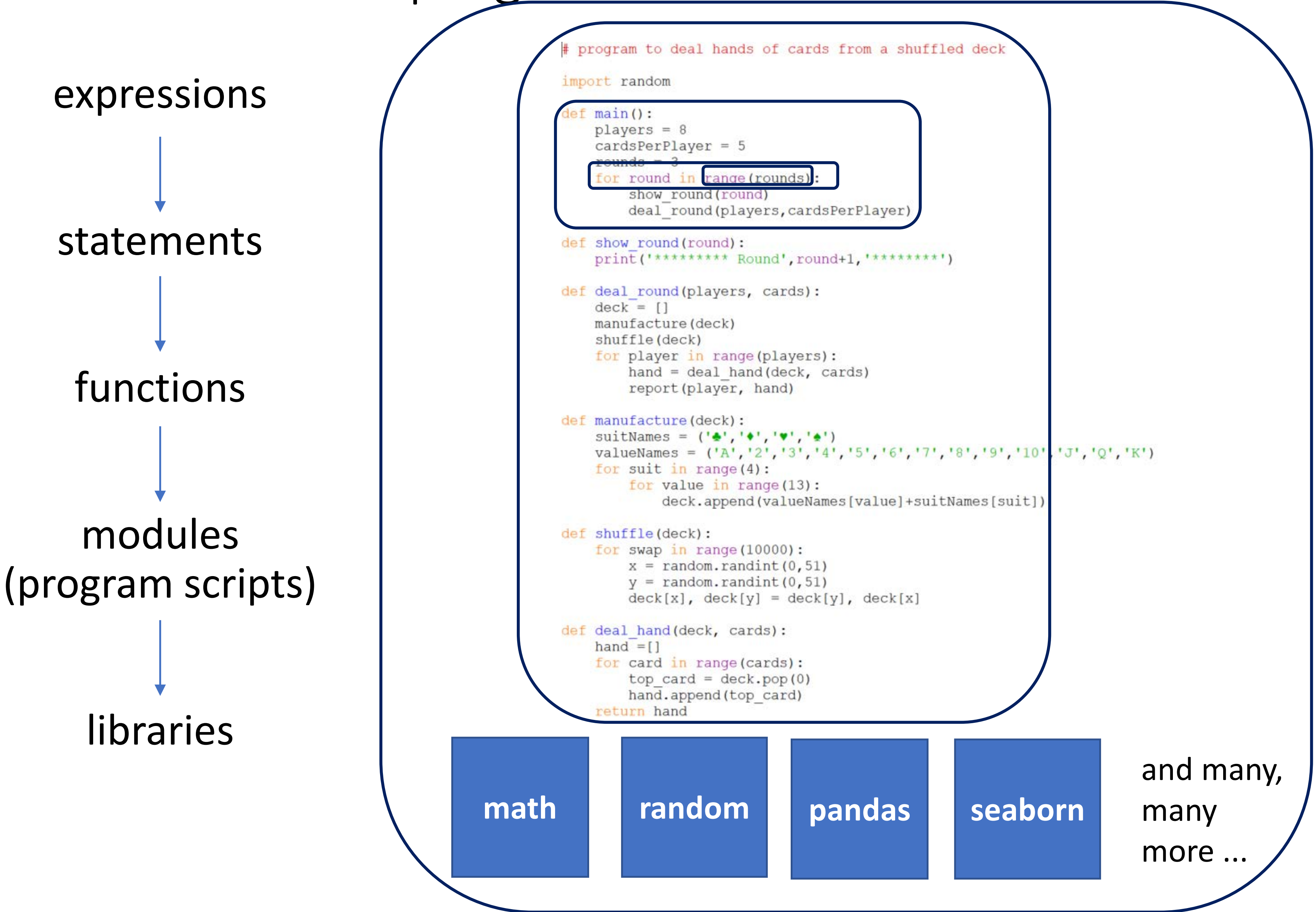
  - e.g., <string>.split(), <list>.sort() ... *and many more*

- **control flow** - redirecting the line-by-line flow of code

  conditionals (if-elif-else)

  iteration (for loops, while loops)

The scales of program code



expressions

statements

functions

modules
(program scripts)

libraries

```
# program to deal hands of cards from a shuffled deck

import random

def main():
    players = 8
    cardsPerPlayer = 5
    rounds = 3
    for round in range(rounds):
        show_round(round)
        deal_round(players,cardsPerPlayer)

def show_round(round):
    print('********* Round',round+1,'*********')

def deal_round(players, cards):
    deck = []
    manufacture(deck)
    shuffle(deck)
    for player in range(players):
        hand = deal_hand(deck, cards)
        report(player, hand)

def manufacture(deck):
    suitNames = ('♣','♦','♥','♠')
    valueNames = ('A','2','3','4','5','6','7','8','9','10','J','Q','K')
    for suit in range(4):
        for value in range(13):
            deck.append(valueNames[value]+suitNames[suit])

def shuffle(deck):
    for swap in range(10000):
        x = random.randint(0,51)
        y = random.randint(0,51)
        deck[x], deck[y] = deck[y], deck[x]

def deal_hand(deck, cards):
    hand =[]
    for card in range(cards):
        top_card = deck.pop(0)
        hand.append(top_card)
    return hand
```

math

random

pandas

seaborn

and many,
many
more ...

Local and Global Scope

The power of modular design

Complex programming challenges are solved by problem decomposition...

... with each function solving part of the task.

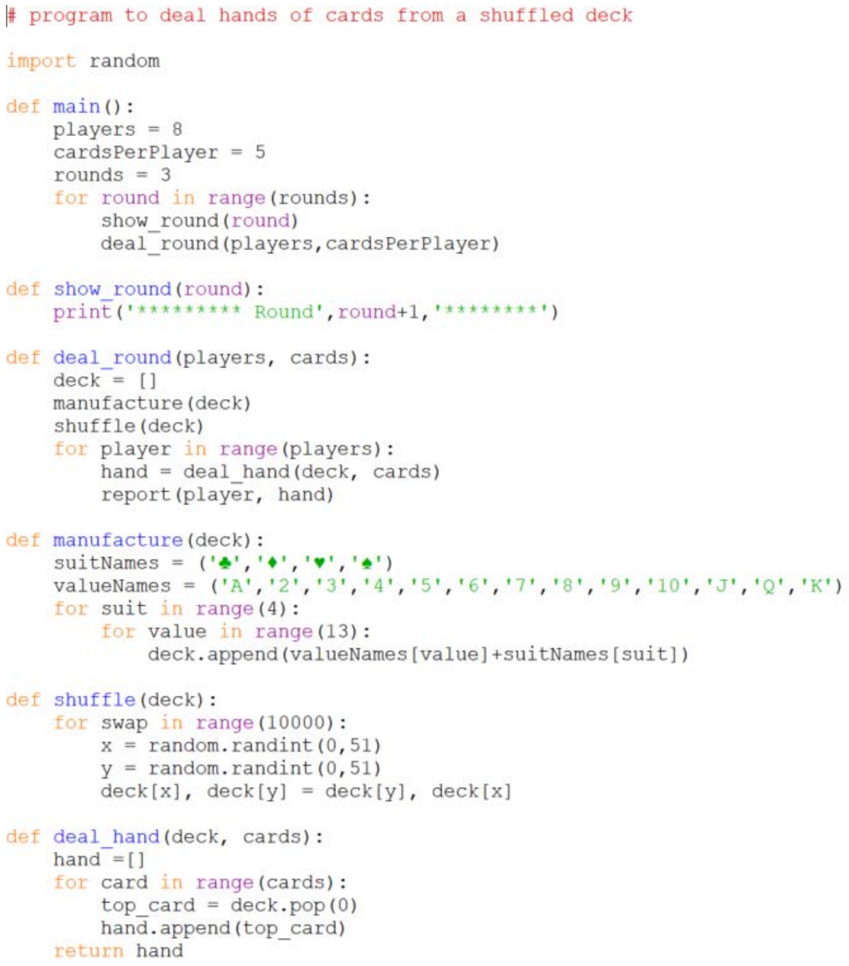
But this raises new challenges:

Getting ONLY the right data into a function ...

- parameter passing
- global variables, but not local variables

Getting ONLY the right data out of a function...

- return values
- work done on mutables, but not work done on immutables



```
# program to deal hands of cards from a shuffled deck

import random

def main():
    players = 8
    cardsPerPlayer = 5
    rounds = 3
    for round in range(rounds):
        show_round(round)
        deal_round(players,cardsPerPlayer)

def show_round(round):
    print('********* Round',round+1,'*********')

def deal_round(players, cards):
    deck = []
    manufacture(deck)
    shuffle(deck)
    for player in range(players):
        hand = deal_hand(deck, cards)
        report(player, hand)

def manufacture(deck):
    suitNames = ('♣','♦','♥','♠')
    valueNames = ('A','2','3','4','5','6','7','8','9','10','J','Q','K')
    for suit in range(4):
        for value in range(13):
            deck.append(valueNames[value]+suitNames[suit])

def shuffle(deck):
    for swap in range(10000):
        x = random.randint(0,51)
        y = random.randint(0,51)
        deck[x], deck[y] = deck[y], deck[x]

def deal_hand(deck, cards):
    hand =[]
    for card in range(cards):
        top_card = deck.pop(0)
        hand.append(top_card)
    return hand
```

This code generates an error because **prices** and **GST** are local to the function **add_gst**, and therefore they are invisible to the function **main**

```
def main():
    for item in range(len(prices)):
        add_gst(item)
        print('Price plus GST: ', prices[item])
        print('GST =', GST)

def add_gst(item):
    prices = [35, 40, 45, 50]
    GST = 15
    prices[item] = prices[item] * (1 + GST/100)

main()
```

```
NameError: name 'prices' is not defined

NameError: name 'GST' is not defined
```

This code generates an error because **prices** and **GST** are local to the function **main**, and therefore they are invisible to the function **add_gst**

```
def main():
    prices = [35, 40, 45, 50]
    GST = 15
    for item in range(len(prices)):
        add_gst(item)
        print('Price plus GST: ', prices[item])
        print('GST =', GST)

def add_gst(item):
    prices[item] = prices[item] * (1 + GST/100)

main()
```

```
NameError: name 'prices' is not defined

NameError: name 'GST' is not defined
```

Parameter passing is the most common solution...

```
def main():
    prices = [35, 40, 45, 50]
    GST = 15
    for item in range(len(prices)):
        add_gst(item, prices, GST)
        print('Price plus GST: ', prices[item])
        print('GST =', GST)

def add_gst(item, prices, GST):
    prices[item] = prices[item] * (1 + GST/100)

main()
```

```
Price plus GST:  40.25
GST = 15
Price plus GST:  46.0
GST = 15
Price plus GST:  51.749
GST = 15
Price plus GST:  57.499
GST = 15
```

Creating **global variables in the top-area** of the program is another solution ...

```
def main():
    for item in range(len(prices)):
        add_gst(item)
        print('Price plus GST: ', prices[item])
        print('GST =', GST)

def add_gst(item):
    prices[item] = prices[item] * (1 + GST/100)

prices = [35, 40, 45, 50]
GST = 15
main()
```

```
Price plus GST:  40.25
GST = 15
Price plus GST:  46.0
GST = 15
Price plus GST:  51.749
GST = 15
Price plus GST:  57.499
GST = 15
```

(Note if **prices** or **GST** are re-assigned inside the functions, it will override the global versions.)

Using the **global** command to give global status to **prices** and **GST** is another solution

```
def main():
    global prices
    global GST
    prices = [35, 40, 45, 50]
    GST = 15
    for item in range(len(prices)):
        add_gst(item)
        print('Price plus GST: ', prices[item])
        print('GST =', GST)

def add_gst(item):
    prices[item] = prices[item] * (1 + GST/100)

main()
```

```
Price plus GST:  40.25
GST = 15
Price plus GST:  46.0
GST = 15
Price plus GST:  51.749
GST = 15
Price plus GST:  57.499
GST = 15
```

(Note if **prices** or **GST** are re-assigned inside the functions, it will override the global versions.)

The power of modular design

Complex programming challenges are solve by problem decomposition...

... with each function solving part of the task.

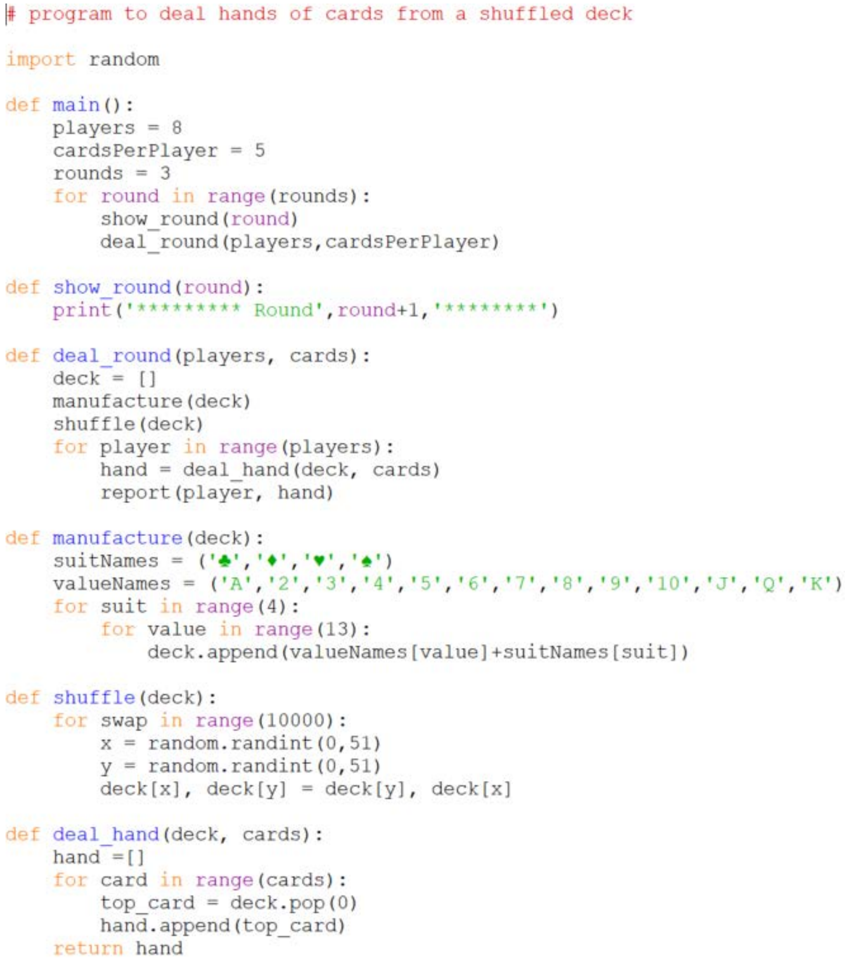
But this raises new challenges:

Getting ONLY the right data into a function ...

- parameter passing
- global variables, but not local variables

Getting ONLY the right data out of a function...

- return values
- work done on mutables, but not work done on immutables



```
# program to deal hands of cards from a shuffled deck

import random

def main():
    players = 8
    cardsPerPlayer = 5
    rounds = 3
    for round in range(rounds):
        show_round(round)
        deal_round(players,cardsPerPlayer)

def show_round(round):
    print('********* Round',round+1,'*********')

def deal_round(players, cards):
    deck = []
    manufacture(deck)
    shuffle(deck)
    for player in range(players):
        hand = deal_hand(deck, cards)
        report(player, hand)

def manufacture(deck):
    suitNames = ('♣','♦','♥','♠')
    valueNames = ('A','2','3','4','5','6','7','8','9','10','J','Q','K')
    for suit in range(4):
        for value in range(13):
            deck.append(valueNames[value]+suitNames[suit])

def shuffle(deck):
    for swap in range(10000):
        x = random.randint(0,51)
        y = random.randint(0,51)
        deck[x], deck[y] = deck[y], deck[x]

def deal_hand(deck, cards):
    hand =[]
    for card in range(cards):
        top_card = deck.pop(0)
        hand.append(top_card)
    return hand
```

Global variables are DANGEROUS - because they may have OTHER effects in the program that are HARD to UNDERSTAND and TRACE.

Parameter passing and **return values** are SAFE and EASY to TRACE.

Parameter passing is the most common solution...

```
def main():
    prices = [35, 40, 45, 50]
    GST = 15
    for item in range(len(prices)):
        add_gst(item, prices, GST)
        print('Price plus GST: ', prices[item])
        print('GST =', GST)

def add_gst(item, prices, GST):
    prices[item] = prices[item] * (1 + GST/100)

main()
```

```
Price plus GST:  40.25
GST = 15
Price plus GST:  46.0
GST = 15
Price plus GST:  51.749
GST = 15
Price plus GST:  57.499
GST = 15
```

Parameter passing is the most common solution...

```
def main():
    prices = [35, 40, 45, 50]
    GST = 15
    for item in range(len(prices)):
        add_gst(item, prices, GST)
        print('Price plus GST: ', prices[item])
        print('GST =', GST)

def add_gst(item, prices, GST):
    prices[item] = prices[item] * (1 + GST/100)
    GST += 1

main()
```

```
Price plus GST:  40.25
GST = 15
Price plus GST:  46.0
GST = 15
Price plus GST:  51.749
GST = 15
Price plus GST:  57.499
GST = 15
```

The important differences between mutables and immutables ...

* work done on the mutable list **prices** made inside **add_gst** take effect in **main**

* work done on the immutable integer variable **GST** do NOT take effect in **main**

Errors and debugging

Code Errors : Bugs

A (software) “bug” is an error/flaw in a piece of code

Mars Climate Orbiter, 1999

- *Goal:* establish an orbit around Mars, and study the weather, climate, etc.
- *What happened:* attempted to orbit too low and crashed as a result
- *Bug:* metric vs. Imperial conversion in calculations
- *Cost:* US$165m

$v = at + v_0$
$r = r_0 + v_0 t + \frac{1}{2}at^2$
$r = r_0 + \frac{1}{2}(v + v_0)t$
$v^2 = v_0^2 + 2a(r - r_0)$
$r = r_0 + vt - \frac{1}{2}at^2$
CD

Even the most carefully-engineered software will include at least 5 errors/1000 lines of code: e.g., *Windows XP contained roughly 45 m lines of code*

Three types of errors

- syntax errors

  Incompatibility with the syntax of the programming language. Often typing mistakes, but can mean there is same problem with the structure of the program.

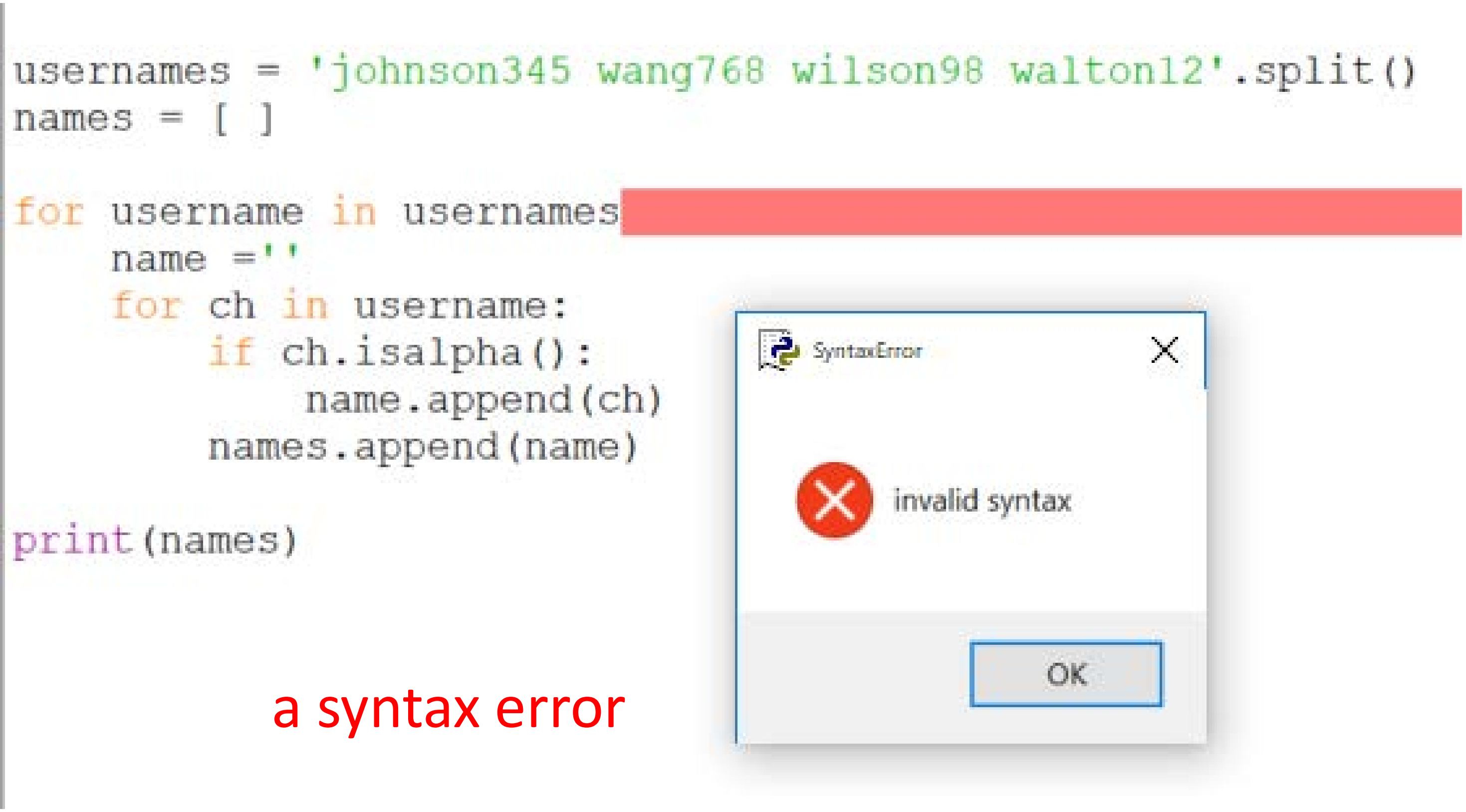
- run-time errors

  Errors occurring as the program is running, causing the code to crash. In an interpreted language, like Python, these errors will not occur until the flow of control in your program reaches the line with the problem. In Python, it takes the form of an “exception” being “raised”.

- logic errors

  Mistakes in the design (before any coding is done), so that the code runs fine, but it doesn’t do what it is supposed to.

Exercise 1. Why won't this program work?



```
usernames = 'johnson345 wang768 wilson98 walton12'.split()
names = [ ]

for username in usernames
    name =''
    for ch in username:
        if ch.isalpha():
            name.append(ch)
        names.append(name)

print(names)
```

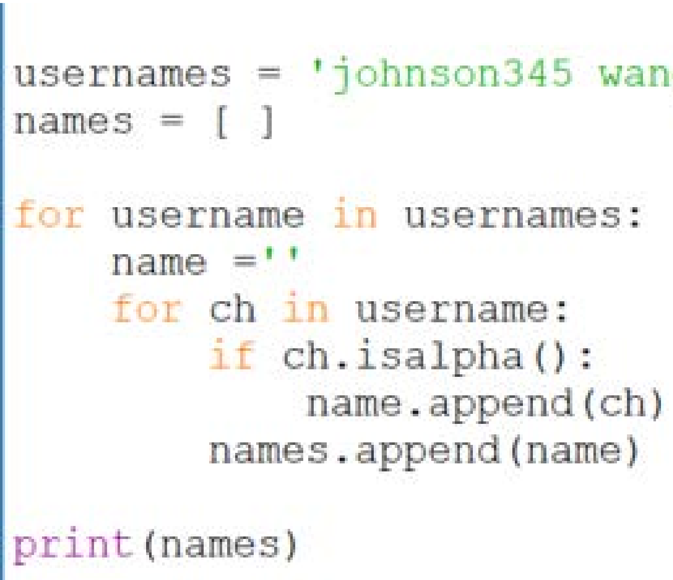
SyntaxError

invalid syntax

a syntax error

['johnson', 'wang', 'wilson', 'walton']

Exercise 1. Why won't this program work?



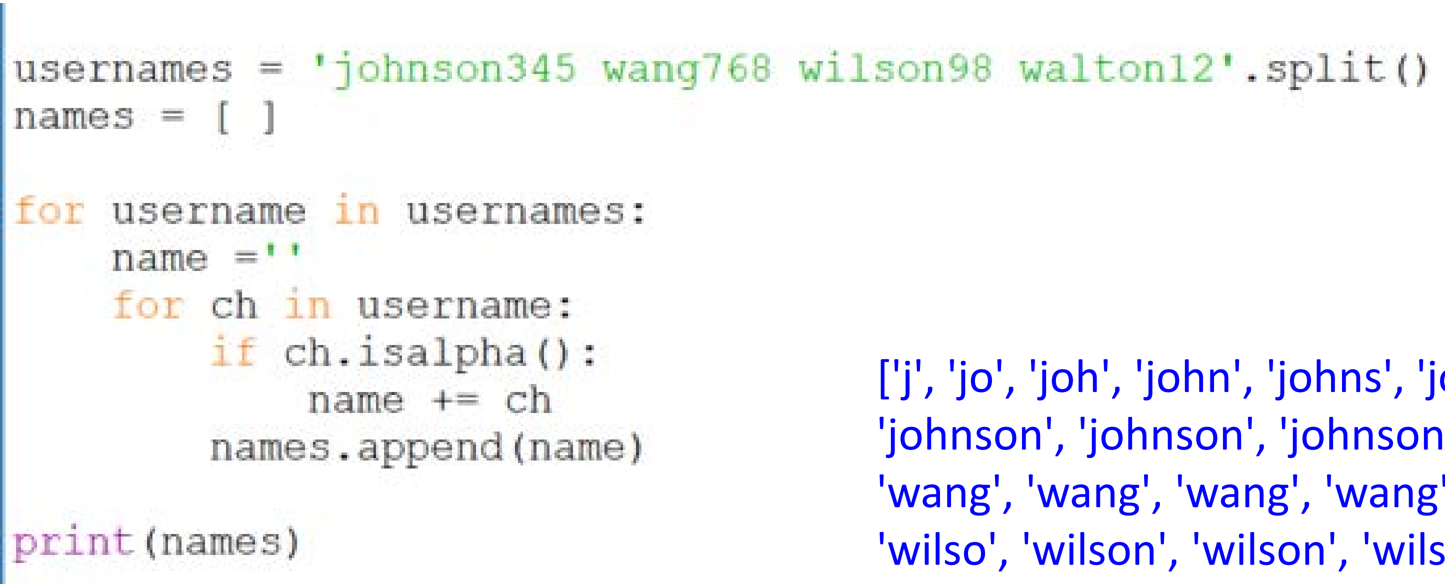
```
usernames = 'johnson345 wang768 wilson98 walton12'.split()
names = [ ]

for username in usernames:
    name =''
    for ch in username:
        if ch.isalpha():
            name.append(ch)
        names.append(name)

print(names)
```

a run-time error

```
Traceback (most recent call last):
  File "G", line 15, in <module>
    name.append(ch)
AttributeError: 'str' object has no attribute 'append'
```

```
['johnson', 'wang', 'wilson', 'walton']
```

Exercise 1. Why won't this program work?



```
usernames = 'johnson345 wang768 wilson98 walton12'.split()
names = [ ]

for username in usernames:
    name =''
    for ch in username:
        if ch.isalpha():
            name += ch
        names.append(name)

print(names)
```

```
['j', 'jo', 'joh', 'john', 'johns', 'johnso', 'johnson',
'johnson', 'johnson', 'johnson', 'w', 'wa', 'wan',
'wang', 'wang', 'wang', 'wang', 'w', 'wi', 'wil', 'wils',
'wilso', 'wilson', 'wilson', 'wilson', 'w', 'wa', 'wal',
'walt', 'walto', 'walton', 'walton', 'walton']
```

a logic error

```
['johnson', 'wang', 'wilson', 'walton']
```

Detecting errors & debugging

- **Syntax errors** can be detected before your program begins to run. These types of errors are usually typing mistakes, but more generally it means that there is some problem with the structure of your program.
- **Runtime errors** occur as your program executes. Since Python is an interpreted language, these errors will not occur until the flow of control in your program reaches the line with the problem.
- **Logic errors** are the hardest to detect and solve. The only way you can identify logic errors is by observing that the output of the program is incorrect.

Errors are found by tracing through the code by hand or using a tool like http://pythontutor.com/

Test your code thoroughly across all the cases that it should solve.

Common Python errors

- equality (==) vs. assignment (=)
- failing to return a value from functions
- incorrect use of types, e.g, forgetting that **input()** returns a string
- incorrect use of a function or method
- mis-spelling a variable name
- indexing and slicing - forgetting to count from zero
- loops and incrementing (out by one!)
- wrong indentation in function definitions, conditionals, loops: shifting lines of code into the wrong block
- forgetting to put () on a function call
- forgetting to put main() in the top-area of the program

Minimize errors occurring

- Build up your code gradually
- Test each function to see that it works on its own
- Choose variables names that are meaningful and easy to distinguish
- Type slowly and carefully, always be on the lookout for common mistakes
- Use brief comments to remind you about any assumptions made

Tips for debugging

- diagnostic print statements
- use of # to turn statement on and off

```
def main():
    usernames = 'johnson345 wang768 wilson98 walton12'.split()
    family_names = extract_names(usernames)
    report(family_names)

def extract_names(usernames):
    names = [ ]
    for username in usernames:
        name =''
        for ch in username:
            if ch.isalpha():
                name += ch
            names.append(name)
   return names

def report(family_names):
     for name in family_names:
         print(name)

main()
```

```
johnson
wang
wilson
walton
```

meant to print this ...

but actually prints this?

```
j
jo
joh
john
johns
johnso
johnson
johnson
johnson
johnson
w
wa
wan
wang
wang
wang
wang
w
wi
wil
wils
wilso
wilson
wilson
wilson
w
wa
wal
walt
walto
walton
walton
walton
```

diagnostic print statements

commenting to
turn statements off

```
def main():
    usernames = 'johnson345 wang768 wilson98 walton12'.split()
    print(usernames)
    family_names = extract_names(usernames)
    print(family_names)
    # report(family_names)

def extract_names(usernames):
    names = [ ]
    for username in usernames:
        name =''
        for ch in username:
            if ch.isalpha():
                name += ch
            print(name)
            names.append(name)
    print(names)
    return names

def report(family_names):
    for name in family_names:
        print(name)
```

```
['johnson345', 'wang768', 'wilson98', 'walton12']

['j', 'jo', 'joh', 'john', 'johns', 'johnso', 'johnson',
'johnson', 'johnson', 'johnson', 'w', 'wa', 'wan',
'wang', 'wang', 'wang', 'wang', 'w', 'wi', 'wil', 'wils',
'wilso', 'wilson', 'wilson', 'wilson', 'w', 'wa', 'wal',
'walt', 'walto', 'walton', 'walton', 'walton']

['j', 'jo', 'joh', 'john', 'johns', 'johnso', 'johnson',
'johnson', 'johnson', 'johnson', 'w', 'wa', 'wan',
'wang', 'wang', 'wang', 'wang', 'w', 'wi', 'wil', 'wils',
'wilso', 'wilson', 'wilson', 'wilson', 'w', 'wa', 'wal',
'walt', 'walto', 'walton', 'walton', 'walton']
```

Exceptions

Python detects when unexpected things happen (called 'exceptions') at run time.

Errors are one kind of exception event.

```
num1 = int(input('Enter a number: '))
num2 = int(input('Enter a number: '))
num3 = int(input('Enter a number: '))

total = num1 + num2 + num3
print('The sum of your numbers is: ',total)
```

```
Enter a number: 3
Enter a number: 4
Enter a number: 5
The sum of your numbers is:  12

Enter a number: 3
Enter a number: 4
Enter a number: t

Traceback (most recent call last):
  File "G", line 3, in <module>
    num3 = int(input('Enter a
number: '))
ValueError: invalid literal for int()
with base 10: 't'
```

Some common exceptions

ValueError – the value of an object is invalid for that type

```
>>> a = int("a")

Traceback (most recent call last):
File "<stdin>", line 1, in <module>

ValueError: invalid literal for int() with base 10: 'a'
```

NameError – an undefined variable has been used

```
>>> b = a

Traceback (most recent call last):
File "<stdin>", line 1, in <module>

NameError: name 'a' is not defined
```

Some common exceptions

IndexError — an out-of-range list or tuple index has been used

```
>>> a = [1,2,3]
>>> a[3]

Traceback (most recent call last):
File "<stdin>", line 1, in <module>

IndexError: list index out of range
```

KeyError — a non-existent dictionary key has been used

```
>>> a = {1:2}
>>> a[2]

Traceback (most recent call last):
File "<stdin>", line 1, in <module>

KeyError: 2
```

UnboundLocalError – referencing a local variable inside a function become variable assignment statement

```
>>> def funct(lst):
        for x in range(len(lst)):
            val = lst[i]

UnboundLocalError: local variable
'i' referenced before assignment
```

TypeError — an operation has been attempted which is invalid for the type of the target object or object type combination

```
a = 1 + "2“

Traceback (most recent call last):
File "<stdin>", line 1, in <module>

TypeError: unsupported operand
type(s) for +: 'int' and 'str'
```

Exception handling: try … except

We can trap errors using the following …

```
try:
    <statements>
except <ErrorType>:
     <handler>
```

```
try:
    num1 = int(input('Enter a number: '))
    num2 = int(input('Enter a number: '))
    num3 = int(input('Enter a number: '))
    total = num1 + num2 + num3
    print('The sum of your numbers is: ',total)
except ValueError:
    print('You entered an incorrect value')
```

```
Enter a number: 2
Enter a number: 4
Enter a number: 5
The sum of your numbers is:  11

Enter a number: 3
Enter a number: t
You entered an incorrect value
```

Lecture Overview

Exam Preparation

A detailed review of what form the exam will take, looking at the MOCK exam questions on Canvas, and thinking about how best to prepare.

Modular program design - local & global scope

A review of the very important idea of modular program design:

- scales of program code
- local and global scope of variables - looking at this in more detail

Errors and de-bugging the code

A review of the different kinds of errors that can occur in a program, and an exploration of the craft of detecting and fixing errors in our code.

The card game challenge

One solution for a program to deal out random hands of playing cards from a shuffled deck of cards. (From Lecture 7)

The card game challenge

Challenge

Write a program that can print out hands of cards from a 'shuffled' pack of playing cards.

```
********** Round 1 *********
Player 1 : 9♥   4♥   3♥   2♦   10♠
Player 2 : J♣   7♦   K♦   6♠   K♣
Player 3 : 7♠   7♣   9♦   K♠   5♣
Player 4 : 8♦   10♣  J♦   Q♠   8♠
Player 5 : 5♠   4♣   5♦   8♥   2♣
Player 6 : 6♦   A♣   Q♥   8♣   J♠
Player 7 : 4♠   4♦   3♣   Q♣   5♥
Player 8 : Q♦   10♥  A♦   9♣   2♠
```

```
suitNames = ('♣','♦','♥','♠')
valueNames = ('A','2','3','4','5','6','7','8','9','10','J','Q','K')
```

Random function

The random library is useful for generating pseudo-random values.

```
import random

for i in range(10):
        print(random.randint(1, 10))
```

```
6
6
8
7
1
2
1
3
10
8
```

```
for card in range(52):
        suit = random.randint(0,3)
        value = random.randint(0,12)
        card = valueNames[value] + suitNames[suit]

suitNames = ('♣','♦','❤','♠')
valueNames = ('A','2','3','4','5','6','7','8','9','10','J','Q','K')
```

manufacture(deck) - round 1

```
['A♣', '2♣', '3♣', '4♣', '5♣', '6♣', '7♣', '8♣', '9♣', '10♣', 'J♣', 'Q♣', 'K♣', 'A♦', '2♦', '3♦', '4♦', '5♦', '6♦', '7♦',
'8♦', '9♦', '10♦', 'J♦', 'Q♦', 'K♦', 'A♥', '2♥', '3♥', '4♥', '5♥', '6♥', '7♥', '8♥', '9♥', '10♥', 'J♥', 'Q♥', 'K♥', 'A♠',
'2♠', '3♠', '4♠', '5♠', '6♠', '7♠', '8♠', '9♠', '10♠', 'J♠', 'Q♠', 'K♠']
```

shuffle(deck) - round 1

```
['K♠', 'K♥', '6♠', '7♠', 'Q♣', '7♦', '3♦', '8♣', '4♦', '9♠', '4♥', '4♣', 'A♥', 'J♥', '2♥', '7♣', '10♣', 'K♣', 'K♦', 'J♣',
'8♦', '5♠', '2♠', 'Q♠', 'A♠', '3♠', '8♠', '5♦', '4♠', '6♣', '7♥', '6♥', '9♦', '10♠', '10♥', '2♦', 'A♣', '8♥', 'Q♦', '2♣',
'10♦', '6♦', '9♣', 'Q♥', '5♥', '9♥', 'A♦', '3♣', '5♣', '3♥', 'J♠', 'J♦']
```

```
********* Round 1 ********
Player 0 : K♠  K♥  6♠  7♠  Q♣
Player 1 : 7♦  3♦  8♣  4♦  9♠
Player 2 : 4♥  4♣  A♥  J♥  2♥
Player 3 : 7♣  10♣  K♣  K♦  J♣
Player 4 : 8♦  5♠  2♠  Q♠  A♠
Player 5 : 3♠  8♠  5♦  4♠  6♣
Player 6 : 7♥  6♥  9♦  10♠  10♥
Player 7 : 2♦  A♣  8♥  Q♦  2♣
```

```
********* Round 2 ********
Player 0 : 9♥  4♥  3♥  2♦  10♠
Player 1 : J♣  7♦  K♦  6♠  K♣
Player 2 : 7♠  7♣  9♦  K♠  5♣
Player 3 : 8♦  10♣  J♦  Q♠  8♠
Player 4 : 5♠  4♣  5♦  8♥  2♣
Player 5 : 6♦  A♣  Q♥  8♣  J♠
Player 6 : 4♠  4♦  3♣  Q♣  5♥
Player 7 : Q♦  10♥  A♦  9♣  2♠
```

```
********* Round 3 ********
Player 0 : J♦ A♣ 6♠ 2♠ J♠
Player 1 : 9♠ Q♦ 3♥ J♣ 8♣
Player 2 : 4♠ 7♣ 3♦ K♦ 5♦
Player 3 : 9♣ 8♦ 4♥ 2♥ 10♥
Player 4 : J♥ 6♦ 10♣ 4♣ 3♣
Player 5 : 5♠ 9♦ 2♣ 5♣ 7♥
Player 6 : 8♥ A♥ 10♠ 6♥ 10♦
Player 7 : 4♦ A♦ 7♠ 8♠ 3♠
```

```
def main():
    players = 8
    cardsPerPlayer = 5
    rounds = 3
    for round in range(rounds):
        show_round(round)
        deal_round(players, cardsPerPlayer)
```

```
def show_round(round):
    print('********* Round',round+1,'********')

def deal_round(players, cards):
    deck = [ ]
    manufacture(deck)
    shuffle(deck)
    for player in range(players):
        hand = deal_hand(deck, cards)
        report(player, hand)
```

```
********* Round 1 ********
Player 0 : K♠  K♥  6♠  7♠  Q♣
Player 1 : 7♦  3♦  8♣  4♦  9♠
Player 2 : 4♥  4♣  A♥  J♥  2♥
Player 3 : 7♣  10♣ K♣  K♦  J♣
Player 4 : 8♦  5♠  2♠  Q♠  A♠
Player 5 : 3♠  8♠  5♦  4♠  6♣
Player 6 : 7♥  6♥  9♦  10♠ 10♥
Player 7 : 2♦  A♣  8♥  Q♦  2♣

********* Round 2 ********
Player 0 : 9♥  4♥  3♥  2♦  10♠
Player 1 : J♣  7♦  K♦  6♠  K♣
Player 2 : 7♠  7♣  9♦  K♠  5♣
Player 3 : 8♦  10♣ J♦  Q♠  8♠
Player 4 : 5♠  4♣  5♦  8♥  2♣
Player 5 : 6♦  A♣  Q♥  8♣  J♠
Player 6 : 4♠  4♦  3♣  Q♣  5♥
Player 7 : Q♦  10♥ A♦  9♣  2♠

********* Round 3 ********
Player 0 : J♦ A♣ 6♠ 2♠ J♠
Player 1 : 9♠ Q♦ 3♥ J♣ 8♣
Player 2 : 4♠ 7♣ 3♦ K♦ 5♦
Player 3 : 9♣ 8♦ 4♥ 2♥ 10♥
Player 4 : J♥ 6♦ 10♣ 4♣ 3♣
Player 5 : 5♠ 9♦ 2♣ 5♣ 7♥
Player 6 : 8♥ A♥ 10♠ 6♥ 10♦
Player 7 : 4♦ A♦ 7♠ 8♠ 3♠
```

```
def deal_round(players, cards):
    deck = [ ]
    manufacture(deck)
    shuffle(deck)
    for player in range(players):
        hand = deal_hand(deck, cards)
        report(player, hand)
```

```
def manufacture(deck):
    suitNames = ('♣','♦','❤','♠')
    valueNames = ('A','2','3','4','5','6','7','8','9','10','J','Q','K')
    for suit in range(4):
        for value in range(13):
            deck.append(valueNames[value]+suitNames[suit])

def shuffle(deck):
    for swap in range(10000):
        x = random.randint(0,51)
        y = random.randint(0,51)
        deck[x], deck[y] = deck[y], deck[x]
```

deck

```
['A♣', '2♣', '3♣', '4♣', '5♣',
'6♣', '7♣', '8♣', '9♣', '10♣',
'J♣', 'Q♣', 'K♣', 'A♦', '2♦',
'3♦', '4♦', '5♦', '6♦', '7♦', '8♦',
'9♦', '10♦', 'J♦', 'Q♦', 'K♦',
'A❤', '2❤', '3❤', '4❤', '5❤',
'6❤', '7❤', '8❤', '9❤', '10❤',
'J❤', 'Q❤', 'K❤', 'A♠', '2♠',
'3♠', '4♠', '5♠', '6♠', '7♠',
'8♠', '9♠', '10♠', 'J♠', 'Q♠',
'K♠']
```

```
['K♠', 'K❤', '6♠', '7♠',
'Q♣', '7♦', '3♦', '8♣', '4♦',
'9♠', '4❤', '4♣', 'A❤', 'J❤',
'2❤', '7♣', '10♣', 'K♣',
'K♦', 'J♣', '8♦', '5♠', '2♠',
'Q♠', 'A♠', '3♠', '8♠', '5♦',
'4♠', '6♣', '7❤', '6❤', '9♦',
'10♠', '10❤', '2♦', 'A♣',
'8❤', 'Q♦', '2♣', '10♦',
'6♦', '9♣', 'Q❤', '5❤', '9❤',
'A♦', '3♣', '5♣', '3❤', 'J♠',
'J♦']
```

```
def deal_round(players, cards):
    deck = [ ]
    manufacture(deck)
    shuffle(deck)
    for player in range(players):
        hand = deal_hand(deck, cards)
        report(player, hand)
```

```
def deal_hand(deck, cards):
    hand =[ ]
    for card in range(cards):
        top_card = deck.pop(0)
        hand.append(top_card)
    return hand

def report(player, hand):
    print('Player',player,': ',end='')
    for card in range(len(hand)):
        print(f'{hand[card]:4}',end='')
    print()
```

```
********* Round 1 ********
Player 0 : K♠  K♥  6♠  7♠  Q♣
Player 1 : 7♦  3♦  8♣  4♦  9♠
Player 2 : 4♥  4♣  A♥  J♥  2♥
Player 3 : 7♣  10♣ K♣  K♦  J♣
Player 4 : 8♦  5♠  2♠  Q♠  A♠
Player 5 : 3♠  8♠  5♦  4♠  6♣
Player 6 : 7♥  6♥  9♦  10♠ 10♥
Player 7 : 2♦  A♣  8♥  Q♦  2♣

********* Round 2 ********
Player 0 : 9♥  4♥  3♥  2♦  10♠
Player 1 : J♣  7♦  K♦  6♠  K♣
Player 2 : 7♠  7♣  9♦  K♠  5♣
Player 3 : 8♦  10♣ J♦  Q♠  8♠
Player 4 : 5♠  4♣  5♦  8♥  2♣
Player 5 : 6♦  A♣  Q♥  8♣  J♠
Player 6 : 4♠  4♦  3♣  Q♣  5♥
Player 7 : Q♦  10♥ A♦  9♣  2♠

********* Round 3 ********
Player 0 : J♦  A♣  6♠  2♠  J♠
Player 1 : 9♠  Q♦  3♥  J♣  8♣
Player 2 : 4♠  7♣  3♦  K♦  5♦
Player 3 : 9♣  8♦  4♥  2♥  10♥
Player 4 : J♥  6♦  10♣ 4♣  3♣
Player 5 : 5♠  9♦  2♣  5♣  7♥
Player 6 : 8♥  A♥  10♠ 6♥  10♦
Player 7 : 4♦  A♦  7♠  8♠  3♠
```