Moodle is experiencing some network issues which may cause page loading to be slow or time out. We are working to get this resolved as soon as possible. Please accept our sincere apologies for the inconvenience. Should you require any assistance please email IT service desk itservicedesk@rhul.ac.uk . Thank you for your patience.

https://royalholloway.ac.uk

Moodle home

Lab Sheet 4 (Due 11 Nov)

Outcomes

The objective of this week's lab is to learn about the process of *abstraction*, which is a crucial element of all programming. You will become familiar with creating different kinds of abstractions, including using meaningful variable names to structure expressions, procedures to abstract sequences of control flow, and classes to abstract collections of data.

Instructions

Each lab sheet consists of six checkpoints. In each week you will be required to complete two Computational Thinking checkpoints and two Programming Checkpoints. In addition there will be two advanced exercises which explore interesting topics relating to the material covered during the week.

Checkpoint 1: Abstracting Expressions [Computational Thinking]

Consider the following print statements

```
print ((-3 - math.sqrt(3 * 3 - 4 * 2 * -5)) / (2 * 2))
print ((-3 + math.sqrt(3 * 3 - 4 * 2 * -5)) / (2 * 2))
```

1. Abstract the numeric constants using appropriately named variables. [Hint: the expression is computing a well-known equation!]
2. There is a complex sub-expression that is common to both lines; abstract this into a variable (can you find a meaningful name?) that is initialised by the sub-expression, so that it only appears once.

Checkpoint 2: Abstracting Procedures [Computational Thinking]

The following Python code calculates the probability of seeing exactly 3 sixes when throwing 5 dice.

```
n = 5
k = 3

x = 1
for i in range(1, n):
   x = x * i

y = 1
for i in range(1, k):
   y = y * i

z = 1
for i in range(1, n - k):
   z = z * i

c = x / (y * z)

p = 1
for i in range(1, k):
   p = p * (1 / 6)

q = 1
for i in range(1, n - k):
   q = q * (5 / 6)

result = c * p * q
```

It is quite complex and, consequently, it is hard to understand the computation and why it works. You will notice that there is a lot of similarity between different parts of the code.

Improve this program by using abstraction to introduce the following procedures.

1. Notice that the first three `for` loops carry out the same operation, but on different variables; write a procedure that contains a loop performing this operation on its parameter and returns the result. Don't forget to give it a meaningful name! Now call this procedure with appropriate arguments and store the results in variables `x`, `y`, and `z`.
2. The remaining two loops in the program also carry out the same operation. Define another procedure (appropriately named!) so that you only have to write this loop once, and then call the procedure to compute the values to be stored in variables `p` and `q`.

3. The code above computes the probability of a particular event, but what if we wanted to also compute the chance of obtaining exactly 3 sixes when we throw 10 dice? Or of obtaining 11 heads in a sequence of 15 coin tosses? It would be inefficient to duplicate the code. Provide a solution to this problem by abstracting your code futher into a procedure that takes three parameters: one for the number of trials, one for the number of outcomes we wish to observe, and a third parameter for the probability of that outcome occurring in any gven trial. (Notice that although the code given above uses four distinct constants, we are actually modelling a situation in which one of the constants, `5 / 6`, is dependent on another one, `1 / 6`; this is because `1 / 6` denotes the probaility of the desired outcome occurring and `5 / 6` the probability of it *not* occurring, and these must sum to `1`). Now call this procedure to compute the chances of the events that were just given as examples.

Checkpoint 3: Abstracting Classes [Computational Thinking]

Consider the Python code below, which calculates and displays the characteristics of a number of the planets of the solar system.

```
import math

# The standard gravitational parameter for the sun
mu = 1.327 * math.pow(10, 20)

def volume(r):
  v = (4 / 3) * math.pi * math.pow(r, 3)
  return str(v)

def surface(r):
  area = 4 * math.pi * math.pow(r, 2)
  return str(area)

name = "Earth"
radius = 6371 # in km
moons = 1

# Print physical characteristics of the Earth
if moons == 1:
```

```
  moons = str(moons) + " moon"
else:
  moons = str(moons) + " moons"
print (name + " has a volume of " + volume(radius) + " cubic km, a surfa
ce area of " + surface(radius) + " sq. km, and " + moons)

# Print dynamic characteristics of the Earth
orbital_radius = 1.496 * math.pow(10, 11)
period = 2 * math.pi * orbital_radius * math.sqrt(orbital_radius / mu)
print (name + " has a year of approximately " + str(period // (60 * 60 *
24)) + " days")

name = "Jupiter"
radius = 69911 # in km
moons = 79

# Print physical characteristics of Jupiter
if moons == 1:
  moons = str(moons) + " moon"
else:
  moons = str(moons) + " moons"
print (name + " has a volume of " + volume(radius) + " cubic km, a surfa
ce area of " + surface(radius) + " sq. km, and " + moons)

# Print dynamic characteristics of Jupiter
orbital_radius = 7.786 * math.pow(10, 11)
period = 2 * math.pi * orbital_radius * math.sqrt(orbital_radius / mu)
print (name + " has a year of approximately " + str(period // (60 * 60 *
24)) + " days")
```

Don’t worry if you don’t understand all the calculations being carried out. This exercise is only about reorganising the existing code into classes and methods, to make it easier to read. If you are interested, you can find out more about the characteristics of Earth and Jupiter, as well as how to compute properties of spheres and orbital periods.

Improve this program by using abstraction to introduce the following.

1. Define a class called `Planet` which contains fields for the properties of planets used in the program: i.e. its name, radius, number of moons, and orbital radius (distance from the sun). Also define an `__intit__` method for the class to allow you to set the values of these fields for each instance. Now create two instances of this class corresponding to the different planets in the program, and initialise two new variables `earth` and `jupiter`.
2. Modify the `volume` and `surface` functions to take a `Planet` object as an argument, instead of a value for the radius.
3. Define two new functions that take a `Planet` object as an argument and return a string containing the physical and dynamic characteristics of that planet, respectively. Modify the program so that it prints out the values computed by these functions.

Sometimes, the number of moons that a planet has can change. For example, the prevailing theory of the origin of Earth's moon is that it is the result of a collision with another small planet.

- Model this situation by adding a method to the `Planet` class, called `collide`, which increments the number of moons a planet has. Remember that this method should take a `self` parameter, but notice that it does not need any other parameters: it only needs to modify `self`.
- Add a call to this method to simulate a collision with Jupiter, and then print out the physical characteristics of Jupiter again.

Checkpoint 4: Perfect Powers [Programming]

Write a Python program `powers.py` or a Java program `Powers.java` that will determine whether a given integer `n` input by the user is a perfect square or a perfect cube, i.e. whether there is an integer `s` such that `s * s = n` or `s * s * s = n`. **Hint**: you can do this using a `while` loop to check, starting from `n = 0`, whether squaring or cubing each successive number is equal to the input. (Be careful with the loop condition to make sure that the `while` loop will always exit, even if the input is not a perfect square or cube!)

Recall that you can get input from the user as follows.

In Python:

```
n = int(input())
```

In Java:

```
java.util.Scanner scanner = new java.util.Scanner(System.in);
int n = scanner.nextInt();
```

You should implement the following.

- Write two functions (in Python) or methods (in Java), one called `perfectSquare` and one called `perfectCube`, which will perform the appropriate check as described above and then return a boolean (true/false).
- Your program should then read an integer input by the user, and then call both of these functions/methods (it is possible for a number to be both a perfect square and a perfect cube) and output the result to the user, e.g.

```
25 is a perfect square
25 is not a perfect cube
```

You might also consider how your program works in the following cases.

- The input is zero.
- The input is a negative integer.

Going Further [Not for Credit]: Can you modify your program so that it takes in an extra number `k`, and computes whether `n` is a perfect k^{th} power?

Checkpoint 5: Number Palindromes [Programming]

Write a Python program `reverse.py` or a Java program `Reverse.java` which will take an integer input by the user, and output whether that number is a *palindrome*, i.e. whether the decimal representation of the number is the same both forward and backwards.

You can do this in two stages.

1. Write a function/method called `reverse`, which takes an integer argument and computes its reverse. You can do this with a `while` loop (**Hint**: use the modulo operator `%` and floor/integer division to compute each individual digit, and build up the

reverse number in an accumulator by multiplying by 10 and then adding the next digit of the input).

2. Determine whether the input is a palindrome by checking whether it is equal to its reverse.

Question: does it make a difference to your program whether the input is positive, negative or zero?

Checkpoint 6: String Prefixes [Programming]

A string P is a *prefix* of another string S when S starts with P. For example, “turn” is a prefix of “turnover”.

Write a Python program `prefix.py` or a Java program `Prefix.java` taking two strings input by the user and which computes whether the first is a prefix of the second using a `for` loop that checks whether each character of the first string is equal to the character at the same position in the second string.

Recall that you can obtain the length of a string as follows.

In Python:

```
s = "This is a string"
l = len(s)
```

In Java

```
String s = "This is a string";
int l = s.length();
```

You may obtain the character at a given position `i` in a string as follows.

In Python:

```
s[i]
```

In Java:

```
s.charAt(i)
```

Remember that the first character in a string is at position (or index) zero!

Be careful to first check that the length of the first string is less than or equal to that of the second string, otherwise you may encounter an error.

Note that you can return a value for a function from inside a for loop.

Checkpoint 7: Repeated Substrings [Advanced Exercise]

A *substring* is a sequence of characters that appears inside a string, e.g. the sequence “C”, “A”, “D” appears in the string “ABRACADABRA”.

In this exercise, you will write a program to determine whether a string input by the user is composed of a sequence of characters repeated some number of times. For example, the string “AAA” is composed of the singleton sequence “A” repeated three times; the string “ABCDABCD” is composed of the sequence “A”, “B”, “C”, “D” repeated twice.

Create either a Python program `repeated_substring.py` or a Java program `RepeatedSubstring.java`, which checks whether its given input string whether is composed of a repeated substring. One way of doing this is to check if it is a repeat of a substring of length 1, a substring of length 2, etc. To check if it is a repeat of a substring of a particular length n, we can test whether the characters that are at a distance of `n` from each other are the same. We will construct the program using methods that abstract these different parts of the computation.

N.B. As above, for a Python variable `s` containing a string, or a Java variable `s` of type `String`, you can use the expression `s[i]` and `s.charAt(i)`, respectively, to access the character at the i^{th} position of the string.

1. Define a method `check_positions` that, for a given string, checks whether each of the characters a certain distance from one another are the same, starting at a given index. For example, taking the string “ABCDABCD”, starting at index 0, and checking the characters at distance 3, the method `check_positions` should return `False`, since the characters “A” (at index 0), “D” (at index 3), and “C” (at index 6), are not all the same.

On the other hand, starting at at index 1, and checking the characters at distance 4, `check_positions` should return true, since the characters at indices 1 and 5 are both “B”.

2. Define a method `check_length` which will check if a given string `s` is made of a repeat of a substring of a particular length `n`. To do this you should check whether the characters n positions apart are all the same, starting at each index 0 through to n - 1. Use the method `check_positions` that you already defined.
3. Finally, defined a method `check_string` which checks whether there is some number `n` such that its input string is made of a repeat of a substring of length n. ([Hint]: think about any optimisations that you can make here; you don’t need to check substrings of *all* lengths!)

Your program should take in a string input by the user and, if it is composed of a repeated substring, outputs the length of that substring.

Going Further: This algorithm is not really efficient: as the size of the input string grows, it can take a very long time to compute its answer. See if you can test this out. For strings up to a few (tens of) thousands of characters you probably cannot notice how long it takes. However, try testing it on a string composed of exactly 362879 "A"s and then 1 "B" (in Python, you can easily create such a string with the expression `362879 * "A" + "B"`). You will notice that it takes a few seconds. Now try with an input string of 3628799 "A"s and then 1 "B"; this takes **much** longer. Many string searching and analysis problems can be efficiently computed using a data structure called a suffix tree, which can be constructed in *linear time*.

Checkpoint 8: A Class for Arbitrary Size Integers [Advanced Exercise]

You may recall from Lab 2 that Python natively supports arbitrary size integers whereas Java’s basic `int` and `long` datatypes do not. However Java does have a class called `BigInteger` for representing arbitrary size whole numbers.

For this exercise, write either a Python program `arbitrary_int.py` or a Java program `ArbitraryInt.java`, in which you implement your own class to represent integers of unbounded size. Internally, your class could use an array of standard integers to store the digits of the number in base 10 (i.e. each element should contain an integer between 0 and 9).

- Add methods to your class, taking two instances of your class, which determine whether they represent equal numbers, and whether the first number is less than the second. This can be done by comparing the digits, from most to least significant, using a loop.
- Add methods to your class to implement addition and multiplication. These should use the procedures of long addition and multiplication with which you are familiar from primary school.
- Now add a method for performing subtraction of two integers of arbitrary size, again using the method you learned in school. In this case, subtraction might result in a negative number (when performing subtraction by hand, this will be indicated by having to 'borrow' from a non-existent digit). Note then in this situation, the magnitude of the negative value is given simply by swapping the arguments and performing the subtraction as usual (i.e. the value of `x - y` when `y` is larger than `x` is simple `-(y - x)`). Add to your class a boolean field that indicates whether the number is positive or negative. N.B. You will now have to modify your implementations of addition and multiplication so that the sign of the result is correct. Also, subtraction may result in zeros in some of the significant digits; to keep your implementation efficient, these digits should be removed from the internal representation.
- Finally, add methods to compute integer division and the remainder of division of two arbitrary size integers.

Going Further: In general, division results in a fraction. Python has a Fraction class for precisely representing any rational number, as the quotient of two arbitrary precision integers: the *numerator* and the *denominator*. (Note, Java's standard library does not have an equivalent). As an extra exercise, see if you can implement your own `Fraction` class. This class should have two fields, `numerator` and `denominator`, which each hold and instance of the class for arbitrary sized integers that you created above. Now, you can add a division method to that class, which returns an instance of your new fraction class.

- Implement methods for adding, subtracting, multiplying and dividing fractions. Remember that dividing one fraction $^{a}/_{b}$ by another $^{c}/_{d}$ is given by multiplying by the inverse fraction $^{d}/_{c}$. Note that for adding and subtracting fractions, the numerators can simply be added/subtracted provided that the denominators are the same; a simple way to ensure this is by multiplying both the numerator and denominator of each fraction by the denominator of the other.
- It will generally be the case that the numerator and denominator of a fraction grow quite large after only a few operations of addition or multiplication. It In order to keep these operations efficient, it is desirable to *reduce* the representation of the fraction by dividing both the numerator and denominator by their greatest common divisor; this can be computed using Euclid's algorithm.

- Finally, see if you can add a method that will return a string containing the decimal representation of the fraction. Each digit can be computed using long division, but you must be careful to detect when the decimal representation has a recurring sequence of digits!

Last modified: Tuesday, 22 October 2019, 11:56 AM

◀ Lab Video 03 - Accumulators

Jump to...

Lab Video 04 - Functions ▶

Navigation

Home
My home
My courses
CS1811-201920
CS1822-201920
Participants
Checkpoints
Lab Sheet 4 (Due 11 Nov)
CS1860-201920
CS1890-201920
CS3001-201920
Library
Careers and Employability
Maths, Stats & Numeracy
CeDAS Self-Study Resources
Avoiding Plagiarism
Advanced Topics
Computer Science Student Information

Activities

 Forums
 Quizzes
 Resources

Royal Holloway, University of London,
Egham, Surrey, TW20 0EX

T: +44 (0)1784 434455