

Lecture 12: Review 2

COMP90059 Introduction to Programming

Wally Smith

School of Computing & Information Systems

Lecture Overview

Last week we reviewed the subject by looking at the Mock Exam - Sample 2, with a focus on solving coding problems.

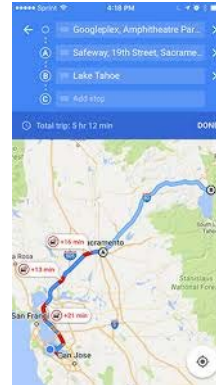
The review this week focuses on the **concepts and theory of programming**.

What follows is a selection of slides from the subject across all of the lectures.

I have selected and re-organized them into core themes of the subject. In this way I am indicating what I think is important for you to concentrate on in your exam preparation. Use this lecture and the slides as prompts to revise the topics in more depth by going back to the original lectures where you are unsure.

What is programming?

What is a computer program?



A set of instructions given to a computer for it to execute a task.

Programming involves ...

- Understanding the task to be carried out.
- Designing a structured and precise process to carry out the task.
- Writing instructions to the computer in its special language for how to execute this process.

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

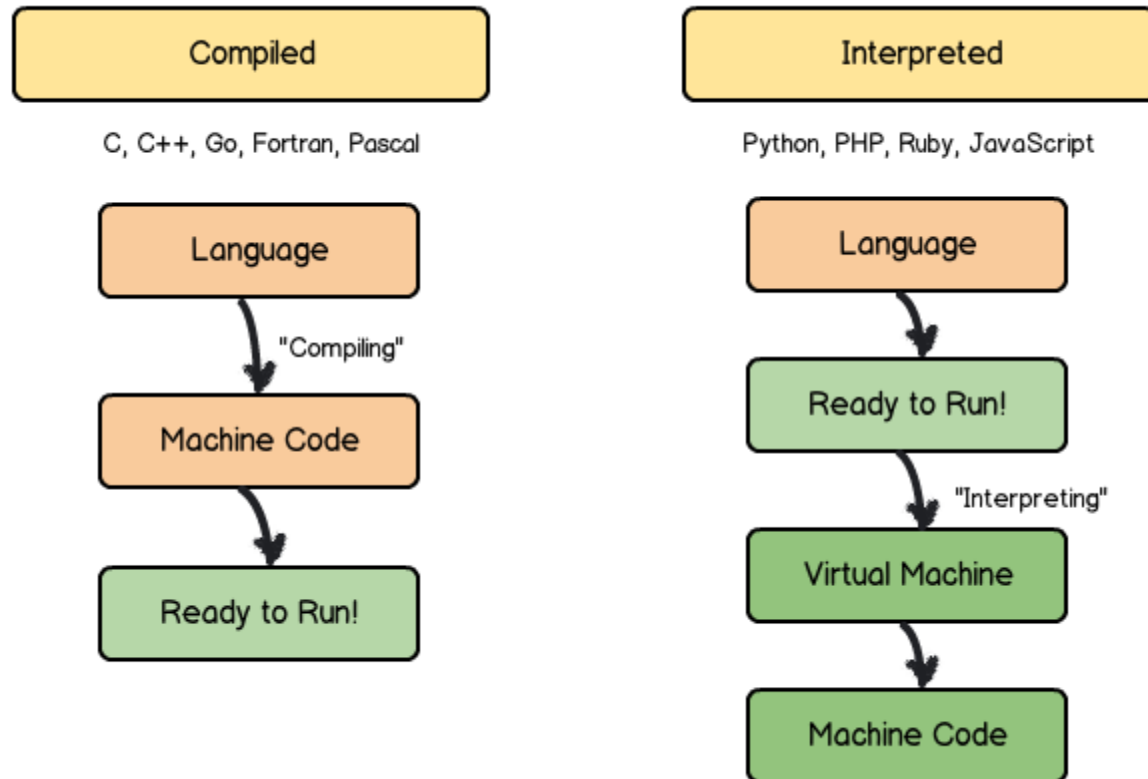
machine code

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

binary

COMPUTER

LECTURE 03



Natural Language vs Programming Language

Natural languages

- complex but flexible grammar rules
- meaning relies on context, and ambiguity is resolved by the listener
- missing or wrong elements might not matter too much
- extensive vocabulary, with subtle differences in meaning
- is understood at the speed of conversation

Programming languages

- have simple but strict grammar rules
- meaning must be clear and precise
- missing or wrong elements leads to a complete breakdown
- a small vocabulary of powerful expressions
- understood and acted on VERY QUICKLY in milliseconds or less

What is good coding?

What makes a piece of code good?

```
principal = eval(input('Enter the initial principal: '))  
apr = eval(input('Enter the annual interest rate: '))  
numberYears = int(input('Enter the number of years: '))  
apr = apr/100  
for i in range(numberYears):  
    principal = principal * (1 + apr)  
print('The final value is: ', round(principal))
```

What makes a piece of code good?

- obeys the syntax of the language
- effective - does the task correctly
- efficient - does the task with the minimum amount of code or computer time or user time and effort
- easy to understand
- easy to maintain
- easy to re-use

#program to tell which of three int numbers is biggest

```
num1 = int(input('Enter number 1: '))
```

```
num2 = int(input('Enter number 2: '))
```

```
num3 = int(input('Enter number 3: '))
```

```
if num1 > num2:
```

```
    if num1 > num3:
```

```
        biggest = num1
```

```
    else:
```

```
        if num3 > num2:
```

```
            biggest = num3
```

```
        else:
```

```
            biggest = num2
```

biggest = num3

```
else:
```

```
    if num2 > num3:
```

```
        biggest = num2
```

```
    else:
```

```
        if num3 > num1:
```

```
            biggest = num3
```

```
        else:
```

```
            biggest = num1
```


biggest = num3

```
print('Biggest: ',biggest)
```

How to make
this code more
elegant?


```
if num1 > num2 and num1 > num3:  
    biggest = num1  
elif num2 > num1 and num2 > num3:  
    biggest = num2  
else:  
    biggest = num3
```

Better because it's
READABLE
(i.e. a person can
easily understand it)



```
biggest = num1  
if num2 > biggest:  
    biggest = num2  
if num3 > biggest:  
    biggest = num3
```

Even better because it's not only
VERY READABLE,
but also **EXTENSIBLE** to more than
three numbers



MORAL: There are many correct solutions to a coding problem.
Some are **ELEGANT**, many are not.

ELEGANT CODE is **efficient**, while still being **readable**, and if possible is **extensible** to more complex cases.

Functions - how do they help program design?

Defining functions is a core technique of programming.

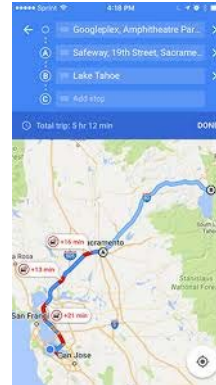
Functions are like sub-programs within the main program.

Organising programs as a set of functions supports problem decomposition.

Functions thereby allow us to create programs with:

- **elegance** (by breaking down complexity into simple parts)
- **readability** (by meaningful function names)
- **traceability** (by reducing code duplication)
- **extensibility** (by providing building blocks of code)

What is a computer program?



A set of instructions given to a computer for it to execute a task.

Programming involves ...

- Understanding the task to be carried out.
- Designing a structured and precise process to carry out the task.
- Writing instructions to the computer in its special language for how to execute this process.

Problem decomposition

Problem decomposition is the way we take a complex problem or task (that is difficult to solve) and break it down into smaller components (each one being easier to solve than the whole).

The secret of good programs is effective problem decomposition, also known as **stepwise refinement**.

Programs decompose the task through **modularity** - meaning the program is made up of separate **functions** of code - each solving a part of the overall task.

An effective program design creates functions that are **loosely coupled** - meaning that they have clearly defined inputs and outputs which are kept as simple as possible.

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()`

useful functions that convert a string to a list

- `list()`
- `sorted()`

... and from a list to a string

- `join()`

ALGORITHM

An algorithm is a sequence of steps for the task, or a part of the task, that will work for every situation that we might encounter:

- Consists of a finite number of instructions
- Each individual instruction is well defined
- Describes a process that eventually halts after arriving at a solution to a problem
- Solves a general class of problems

Coding rules and style

SYNTAX

Syntax is the name for the rules of a language.

It defines what counts as a correct statement in that language.

There are many different computer languages:

e.g. Python, JAVA, C, C++, COBOL, FORTRAN, ...

Each one is suited to writing instructions for different kinds of tasks, each one has its own syntactic rules.

Variable Names - Rules in Python

- Reserved words cannot be used as variable names
Examples: if, for, while and import
- Don't use built-in functions as variable names
Examples: enumerate, print, input
- Names must begin with a letter or underscore _
- The rest of the name can contain zero or more occurrences of the following things:
 - digits (0 to 9)
 - alphabetic letters
 - underscores
- Names are case sensitive
Example: **WEIGHT** is different from **weight**

Good practice for writing code

Style Guide of PEP8 (Python Enhancement Proposals)

<http://www.python.org/dev/peps/pep-0008>

- Code is read much more often than it is written
- Aim for greater consistency & readability

Variable Naming Conventions

When choosing names for variables ...

- be succinct
 - find the most meaningful word or term that most clearly and unambiguously identifies what the variable represents in the task
 - but also try to keep it short!
 - depends on what other variables you need to define
 - **amount** - might be a good name in one program, but confusing in another
- use lower case with “camel casing” for multiple words
 - Example: **target**, **interestRate**, **daysFirstYear**
- use all uppercase letters for symbolic constants
 - Examples: **TAX_RATE** and **STANDARD_DEDUCTION**

Some tips from PEP8

- Lines must not exceed 79 characters (use \ to indicate that the statement is continued on the next line)
- Use blank lines to separate logical sections (e.g. between functions and major sections of code associated with a given comment)
- 4 spaces per indent line level
- Variable names: lower case, meaningful names, camel casing (Lecture 2)
- Function names should be lowercase, with words separated by underscores
- Constants should be written in all capitals, with words separated by underscores (e.g., TAX_RATE) (Lecture 2)
- Don't compare Boolean values to True or False using ==

The coding process

IDLE, scripts, & the Python shell ...

Python IDLE (Interactive DeveLopment Environment) - this is where you develop your programs ...

... you can write directly into the Python shell

... or save your code in a .py file, called a 'script' or 'module' and then run it

... double clicking on a .py file will run the script in a new window

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.

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.

The structure of programs

The different elements of code ...

```
principal = int(input('Enter the initial principal: '))
apr = float(input('Enter the annual interest rate: '))
numberYears = int(input('Enter the number of years: '))
apr = apr/100
for i in range(numberYears):
    principal = principal * (1 + apr)
print('The final value is: ', round(principal))
```

built-in functions, of Python

reserved words, of Python

variable names, created by programmer

prompt, created by programmer

literals, created by programmer

Reserved Words & Built-in Functions

Reserved Words

False	class	finally	is	return
None	continue	for	lambda	try
True	def	from	nonlocal	while
and	del	global	not	with
as	elif	if	or	yield
assert	else	import	pass	
break	except	in	raise	

Built-in Functions

abs()	dict()	help()	min()	setattr()
all()	dir()	hex()	next()	slice()
any()	divmod()	id()	object()	sorted()
ascii()	enumerate()	input()	oct()	staticmethod()
bin()	eval()	int()	open()	str()
bool()	exec()	isinstance()	ord()	sum()
bytearray()	filter()	issubclass()	pow()	super()
bytes()	float()	iter()	print()	tuple()
callable()	format()	len()	property()	type()
chr()	frozenset()	list()	range()	vars()
classmethod()	getattr()	locals()	repr()	zip()
compile()	globals()	map()	reversed()	__import__()
complex()	hasattr()	max()	round()	
delattr()	hash()	memoryview()	set()	

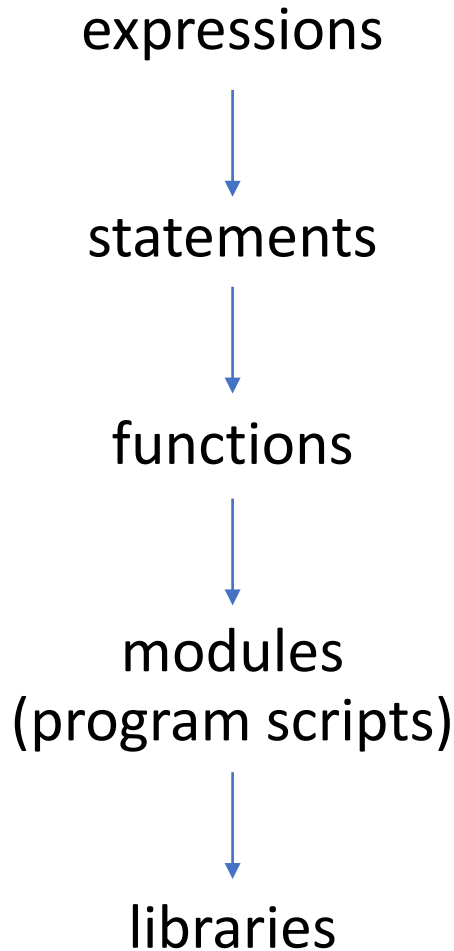
The anatomy of programs

- **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



```
# 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 ...

Expressions

Expressions are 'fragments of code that produce or calculate new data values' (Zelle, 2017)

- `name * 5`
- `yearOfBirth + yearsElapsed`
- `timeAtRaceEnd - timeAtRaceStart`

Libraries, modules and functions

- Module - the name for program code stored in a file
- Libraries - are special modules that contain an extra set of valuable functions that we can use
- For example, the Python math module/library contains many useful mathematical functions
- To use the math library in your program, you need to first tell Python ... add the following line to your code
 - `import math`
- Later in the program you can call any of its functions by adding 'math.' to the front
 - e.g., `number = math.sqrt (200)`

Using the math module in the python shell

```
>>> import math
>>> dir(math)
['__doc__', '__file__', '__loader__', '__name__', '__package__', '__spec__'
__, 'acos', 'acosh', 'asin', 'asinh', 'atan', 'atan2', 'atanh', 'ceil',
'copysign', 'cos', 'cosh', 'degrees', 'e', 'erf', 'erfc', 'exp', 'expm1',
'fabs', 'factorial', 'floor', 'fmod', 'frexp', 'fsum', 'gamma', 'gcd',
'hypot', 'inf', 'isclose', 'isfinite', 'isinf', 'isnan', 'ldexp', 'lgamm
a', 'log', 'log10', 'log1p', 'log2', 'modf', 'nan', 'pi', 'pow', 'radian
s', 'remainder', 'sin', 'sinh', 'sqrt', 'tan', 'tanh', 'tau', 'trunc']
>>>

>>> help(round)
Help on built-in function round in module builtins:

round(number, ndigits=None)
    Round a number to a given precision in decimal digits.

    The return value is an integer if ndigits is omitted or None.  Otherwise
    the return value has the same type as the number.  ndigits may be negative.
```

- To use a resource from a module, you write the name of a module as a qualifier, followed by a dot (.) and the name of the resource
- – Example: math.pi

```
>>> print(math.pi * 2)
6.283185307179586
```

Data types & data structures

Data Types

Type of data	Python type name	Examples (literals)
Integers	int	-1, 0, 6895
Real numbers	float	-0.101011, 567738.009187
Character strings	str	""', 'd', "I'm a string", '3'
Boolean	bool	True, False

- 'The data type of an item determines what values it can hold and what operations it supports' (Zelle, 2017)
- A literal is a specific or 'actual' data value

Floating-point numbers

DECIMAL NOTATION	SCIENTIFIC NOTATION	MEANING
3.78	3.78e0	3.78×10^0
37.8	3.78e1	3.78×10^1
3780.0	3.78e3	3.78×10^3
0.378	3.78e-1	3.78×10^{-1}
0.00378	3.78e-3	3.78×10^{-3}

Dynamic typing

- A distinctive feature of Python is that it decides what types a variable is when you first assign a value to it ...

```
name = 'Cindy'
```

Python creates a variable called `name`, and designates it as a string variable, and associates the variable with the literal value `'Cindy'`.

Type conversion

- To “cast” a literal or variable to a different type, we use functions of the same name as the type:
- `int()`, `float()`, `str()`

```
amountAUDstr = input('Enter amount in Australian Dollars  
and Cents: ')
```

```
amountAUD = float(amountAUDstr)
```

```
rateAUDtoIC = 84.32
```

```
amountIK = 84.32 * amountAUD
```

```
print('Equivalent to', amountIK, 'in Icelandic Krona')
```

```
age = int( input('Enter your name: ') )
```


Sequences

Data structures in which elements exist in a strict sequence

strings: a sequence of characters, e.g., 'This is a sentence'

lists: a sequence of data objects

```
fruits = ['apple', 'banana', 'orange', 'grape', 'tomato', 'lemon']
```

```
customers = ['Johnson', 'Wang', 'Zhu', 'Agarwal', 'Williams',  
             'Nguyen', 'vom Lehn', 'Wilson', 'Trudeau']
```

Similar operations and methods apply to sequences, as we will see.

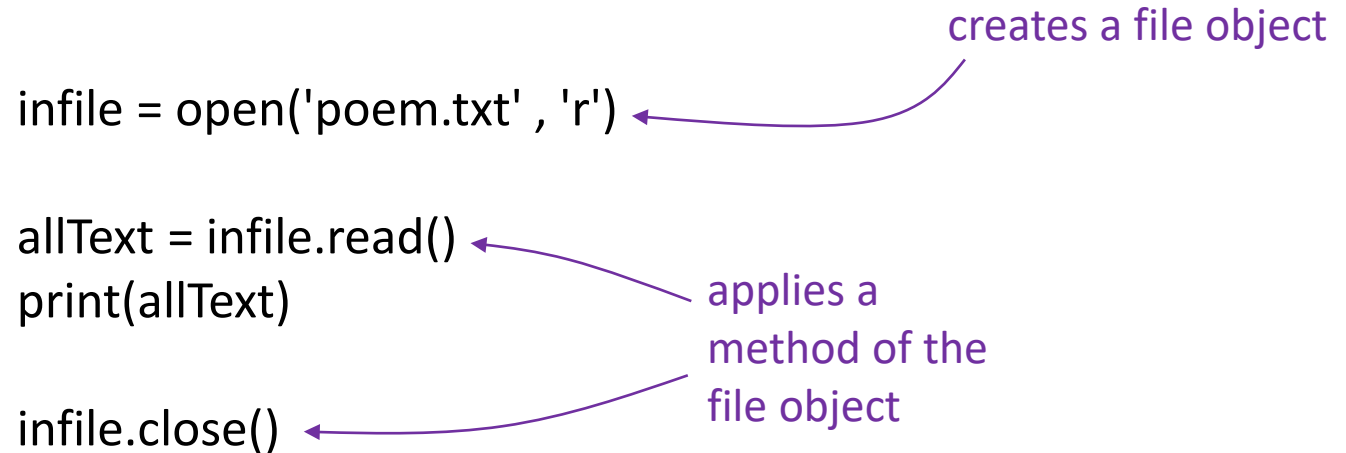
Reading from files

Suppose we want to read from a file called 'poem.txt' ...

```
infile = open('poem.txt' , 'r')  
  
allText = infile.read()  
print(allText)  
  
infile.close()
```

creates a file object

applies a method of the file object



Alternative methods for reading in the data from the file:


- `read()` - reads everything in the file as one long string
- `readline()` - reads the next single line in the file (up to `\n`)
- `readlines()` - reads all of the (remaining) lines in the file

Writing to files

We can also write data back to create new files ...

```
outfile = open('test_data.txt' , 'w')  
  
for i in range(1000):  
    print(i, file=outfile)  
  
outfile.close()
```

creates a file object



Collections

Collections are data structures that contain many data items.

There are four different kinds of collections shown below.

In other languages, collections are usually called 'arrays' and they are much less flexible than Python's collections.

- **lists**

ordered and **mutable** collections of values

- **tuples**

ordered and **immutable** collections of values

- **dictionaries**

un-ordered and **mutable** collections of **key-value pairs**

- **sets**

unordered collection of unique values that are **immutable**
(though the set overall is **mutable**)

Multidimensional lists

We have already talked about how lists can contain a mixture of data types, including lists.

Lists within lists is a very powerful technique for handling large data sets.

```
customers = [  
    ['James', 'Johnson', 'jd@gmail.com', '23A Cedar Avenue, VIC 3078', 25],  
    ['Xinyu', 'Wang', 'xw@gmail.com', '17B Fountain Court, VIC 3000', 37],  
    ['Aesha', 'Acharya', 'aa@gmail.com', '22 Franklin Rd, VIC 3043', 23],  
    ['John', 'Dang', 'jd@gmail.com', '102 Princess Lane, VIC 3002', 28]  
]
```

```
print(customers[2])    ['Aesha', 'Acharya', 'aa@gmail.com', '22 Franklin Rd, VIC 3043', 23]
```

```
print(customer[2][0])  Aesha
```

Mutability

Mutability

Mutability is the property of a data type that determines if its value(s) can be changed *in place*.

mutables: lists, dictionaries

immutables: integers, floats, strings, Booleans, tuples

Lists are **mutable** ...

```
tennis_players = [Caroline Wozniacki, Karolína Plíšková, Simona Halep]  
tennis_players[0] = 'Ashleigh Barty' # this works fine
```

Tuples are **immutable** ...

```
day_names = (Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday)  
day_names[4] = 'Vendredi' # this DOES NOT WORK
```

Strings are **immutable** ...

```
player = 'Karolína Plíšková'  
player[3] = 'o' # this DOES NOT WORK
```

Hold on !! Aren't strings and integers changeable?

It seems like we can change strings, integers, (& floats, Booleans) ...

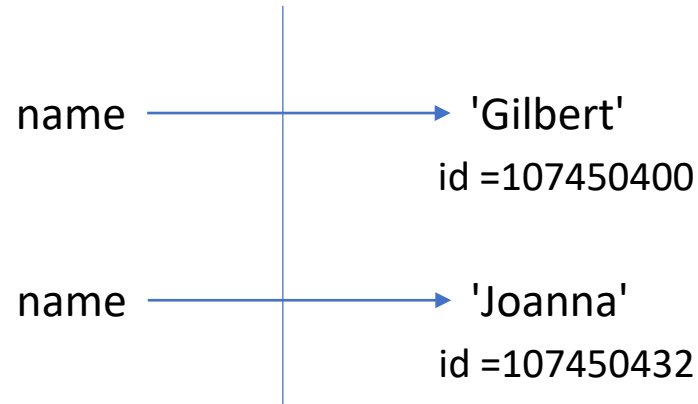
```
name = 'Gilbert'  
print(name)  
name = 'Joanna'  
print(name)
```

Gilbert
Joanna

To understand why this does not count as *changing in place* ...
... we have to think about what happens behind the scenes in Python

```
name = 'Gilbert'  
print(id(name))
```

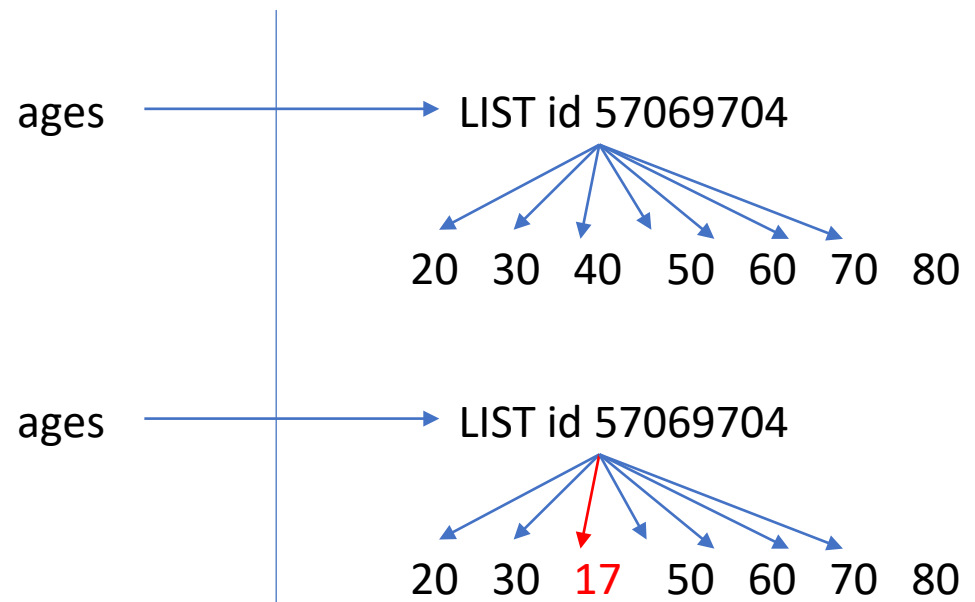
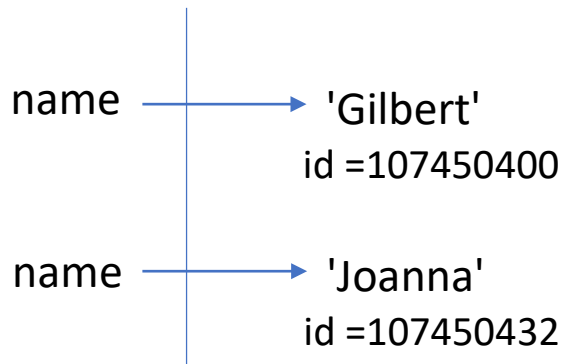
```
name = 'Joanna'  
print(id(name))
```



HOWEVER, if we change the values within a list the pattern is different ...

```
ages = [20,30,40,50,60,70,80]
print(ages, id(ages))           [20, 30, 40, 50, 60, 70, 80]  57069704
ages[2] = 17
print(ages, id(ages))           [20, 30, 17, 50, 60, 70, 80]  57069704
```

This is now a *change in place* ... because the underlying object is the same.



Why does mutability matter: aliasing

One important consequence of the mutability of lists is aliasing.
Assigning a list to another list, does NOT create a new list object ...
.. it just creates an ALIAS - a second name for the same underlying list object

```
ages = [20,30,40,50,60,70,80]
specials = ages
ages[2] = 17
print(ages, id(ages))           [20, 30, 17, 50, 60, 70, 80]  99255944
print(specials, id(specials))   [20, 30, 17, 50, 60, 70, 80]  99255944
```

This does NOT happen for immutables ...

```
age = 20
special = age
age = 17
print(age, id(age))
print(special, id(special))
```

```
17    1437853872
20    1437853920
```

```
age = 'young'
special = age
age = 'not so young'
print(age, id(age))
print(special, id(special))
```

```
not so young  96272496
young         99324000
```

Why does mutability matter: lists as parameters

Another important consequence of mutability ...

what happens to **lists** when they are passed to functions as parameters

```
def main():
    ages = [20,30,40,50,60,70,80]
    print(ages, id(ages))           [20, 30, 40, 50, 60, 70, 80]      83218568
    double_values(ages)
    print(ages, id(ages))           [40, 60, 80, 100, 120, 140, 160]  83218568
```

```
def double_values(x):
    for i in range(len(x)):
        x[i] = 2 * x[i]
```

This does not happen to immutables:
integers, strings, floats, booleans or tuples

```
main()
def main():
    age = 20
    print(age, id(age))           20  1437853920
    double_values(age)
    print(age, id(age))           20  1437853920
```

```
def double_values(x):
    x = 2 * x
```

```
main()
```

Control flow

Conditionals

Conditionals are a core technique in all programming languages to specify what actions to take under different conditions

- *if a parking fine has been paid, send a thank you message*
- *if it has NOT been paid, send a reminder to pay*

if, else, elif ... are reserved words in Python to handle conditionals

Boolean logic - is the way we handle if conditions are true or not. Every conditional statement depends on Boolean expressions to determine which action to take.

Operator precedence for highest to lowest

TYPE OF OPERATOR	OPERATOR SYMBOL
Exponentiation	**
Arithmetic negation	-
Multiplication, division, remainder	*, /, %
Addition, subtraction	+, -
Comparison	==, !=, <, >, <=, >=
Logical negation	not
Logical conjunction and disjunction	and, or
Assignment	=

Iteration

Repeating the same process many times to achieve a desired outcome.

In programming, iteration means creating a block of code that is repeated cyclically in a loop.

There are three main forms of iteration ...

- **eternal** - repeat something forever
(e.g., Windows)
- **definite or counted** - repeat **for** a fixed number of times
(e.g., move Mario forward 10 pixels; print 7 copies)
- **indefinite or conditional** - repeat **while** some condition remains true
(e.g., scroll while button pressed)

Choosing between **for** and **while** loops

for loops

- Given a choice between the two, for loops are generally more elegant/safer/easier to understand
- Use a for loop, if you need to iterate over all items of an iterable,

while loops

- Consider a while loop, if there is a well defined end-point to the iteration which doesn't involve iterating over all items,
- While loops become powerful when unpredictable results or events arise during the iteration (often involving user input)

Using a sentinel loop for data entry

A better approach is to use a sentinel value - a data item value which is recognized as the end point of the data set being entered.

A good sentinel value is the empty string - detecting that just return was entered to signal the end of the data.

program with sentinel loop

to take data from the user and calculate mean

```
count = 0
```

```
total = 0
```

```
entryStr = input('Enter a number (press \'Enter\' to quit): ')
```

```
while entryStr != "":
```

```
    number = float(entryStr)
```

```
    total += number
```

```
    count += 1
```

```
    entryStr = input('Enter a number (press \'Enter\' to quit): ')
```

```
print(f'The mean of your data is {total/count:10.3f}')
```

Enter a number (press 'Enter' to quit): 4

Enter a number (press 'Enter' to quit): 5.6

Enter a number (press 'Enter' to quit): 6

Enter a number (press 'Enter' to quit): 7.2

Enter a number (press 'Enter' to quit):

The mean of your data is 5.700

Using a while loop with break to check for valid data

A very common use of while loops is to check if data entry is valid before continuing.

`break` - is a Python reserved word to that breaks out of a loop before it has finished

while True:

```
    number = float(input('Enter a number between 1 and 10 : '))
```

```
    if number >= 1 and number <= 10:
```

```
        break
```

```
    else:
```

```
        print('Sorry that is out of range.')
```

Enter a number between 1 and 10 : 0

Sorry that is out of range.

Enter a number between 1 and 10 : 34

Sorry that is out of range.

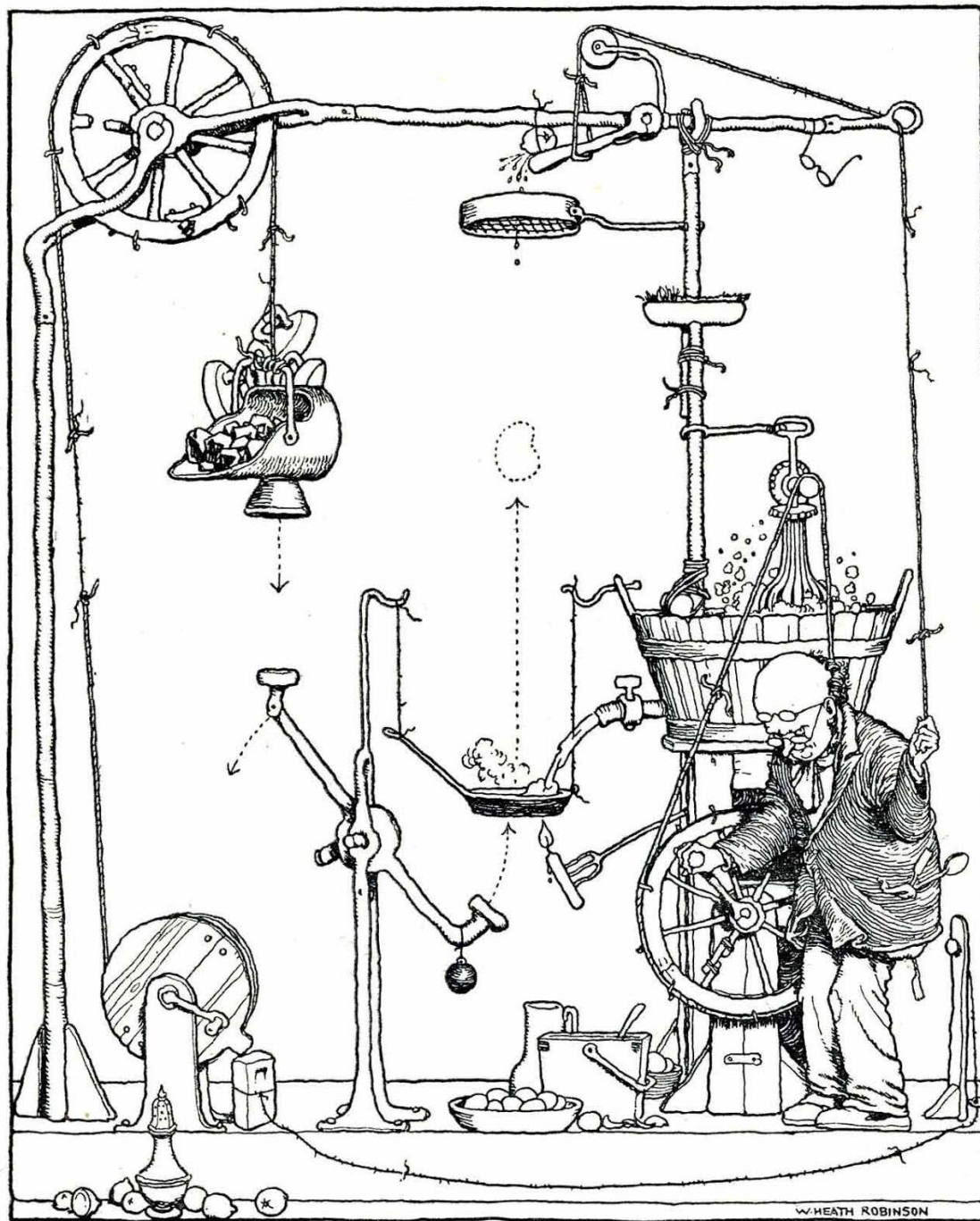
Enter a number between 1 and 10 : 55

Sorry that is out of range.

Enter a number between 1 and 10 : 7

>>>

Modular program design



Functions - how do they help program design?

Defining functions is a core technique of programming.

Functions are like sub-programs within the main program.

Organising programs as a set of functions supports problem decomposition.

Functions thereby allow us to create programs with:

- **elegance** (by breaking down complexity into simple parts)
- **readability** (by meaningful function names)
- **traceability** (by reducing code duplication)
- **extensibility** (by providing building blocks of code)

Parameters and arguments

```
def greet(name, age, address):  
    print('Hello', name)  
    print('Please confirm that we have your details correct:')  
    print('Age:', age, 'years')  
    print('Address:\n', address)
```

parameters
= variables in definition

...

```
greet('Susan', 26, ['2/32 Johnston Street', 'Collingwood', 'VIC 3066'])
```

```
n = 'Susan'  
a = 26  
addr = ['2/32 Johnston Street', 'Collingwood', 'VIC 3066']  
greet(n, a, addr)
```

arguments = values sent when function called

Hello Susan

Please confirm that we have your details correct:

Age: 26 years

Address:

['2/32 Johnston Street', 'Collingwood', 'VIC 3066']

Positional Arguments vs Keyword Arguments

So far, we have been providing positional arguments in our function calls. This means that the arguments are in the same order as the original function parameters.

```
def greet(name, age, address)
...
greet('Susan', 26, ['2/32 Johnston Street', 'Collingwood'])
```

Another technique that Python allows is through keywords, allowing a different order of parameters.

```
def greet(name, age, address)
...
greet(age =26, address =['2/32 Johnston Street', 'Collingwood'], name ='Susan')
```

The power of modular program 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.

Functions - variable scope: local and global

Variables that are created inside a function are called local.

Local variables do not exist outside of the function.

Local may be created with the same name as an outside variable, but they are actually different.

```
def showCube(x):  
    x = x ** 3  
    print('Cube is', x)
```

```
x = 4
```

```
showCube(x)
```

```
print(x)
```

Cube is 64

4

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.)

Lecture 12: Review 2

COMP90059 Introduction to Programming

Wally Smith

School of Computing & Information Systems