IFB104 — Building IT Systems
Topic 10 — How to Pass the Final Exam

School of Computer Science

Semester 1, 2025

Housekeeping

- Marking is getting underway for Assessment Task 2B
  - We will do our best to have your grades released before the exam, but there is no guarantee
  - But either way this shouldn't affect your study plan – simply aim to do your best. Higher grades will increase your chances of accessing rewarding projects and internship opportunities
- There are no more IFB104 workshops

Housekeeping

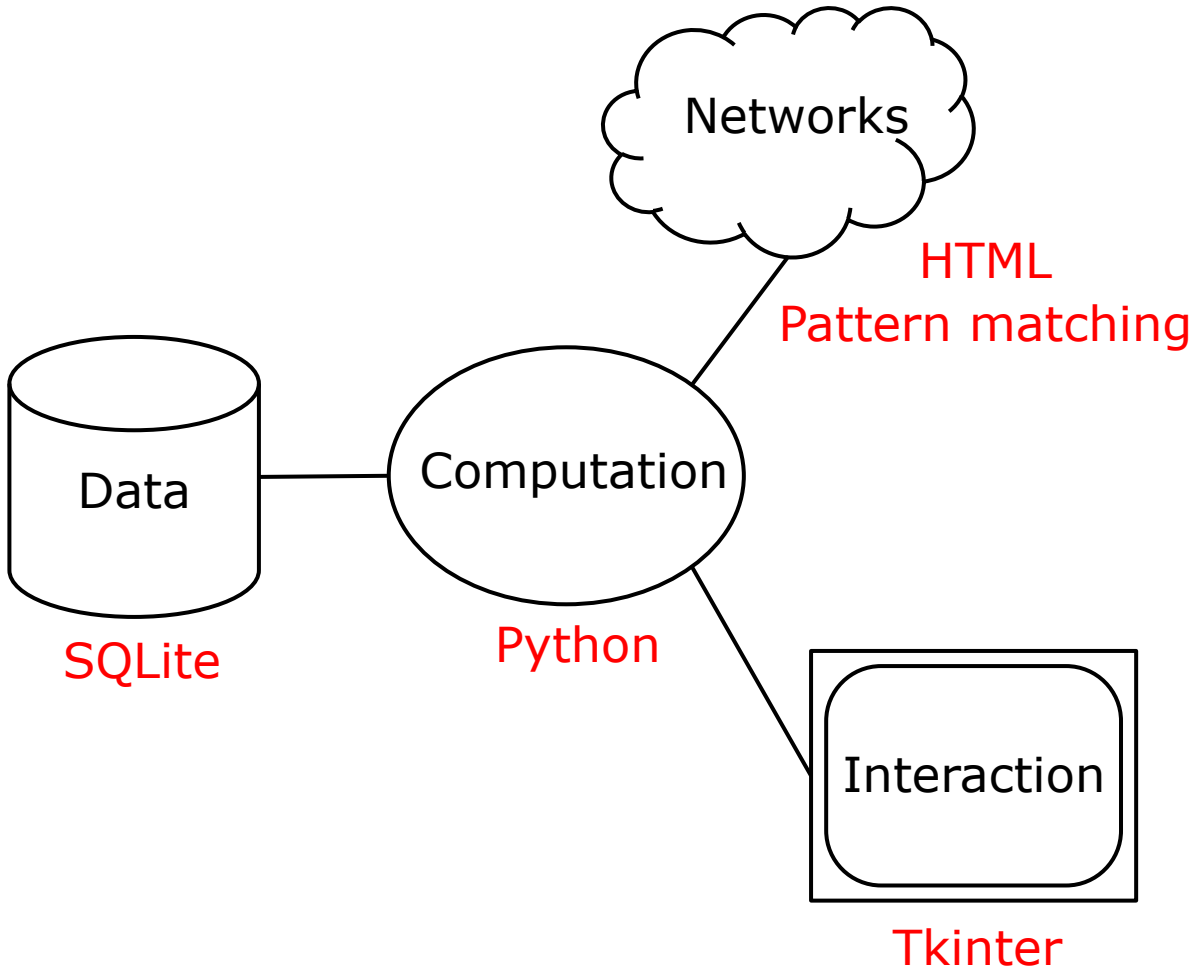
- Your next task is to get ready for the final exam
  - When: **Saturday, 7th June 2025, 8:30am–11:10am** for most students*
  - Where: **Canvas | IFB104 | Assignments | Assessment Task 3**
  - What: See the **Week 13 Practice Exam** (available from Wednesday)
    - attempt the weekly self-assessment quizzes
- You **must** be available to sit the exam at the time above

  * Check your personal exam timetable at https://qutvirtual4.qut.edu.au/group/student/study/exams/exam-timetables

Part A — Unit summary

What the heck was that all about?

- IFB104 - Building IT Systems gave you a *broad*, but *shallow*, introduction to some core programming concepts and applications
  - It was intended as a taster to help you decide what areas to study in depth in later IT teaching units
  - Assignment 1 was a warm-up coding exercise with a focus on data processing
  - Assignment 2 gave you an opportunity to develop a complete, stand-alone desktop application

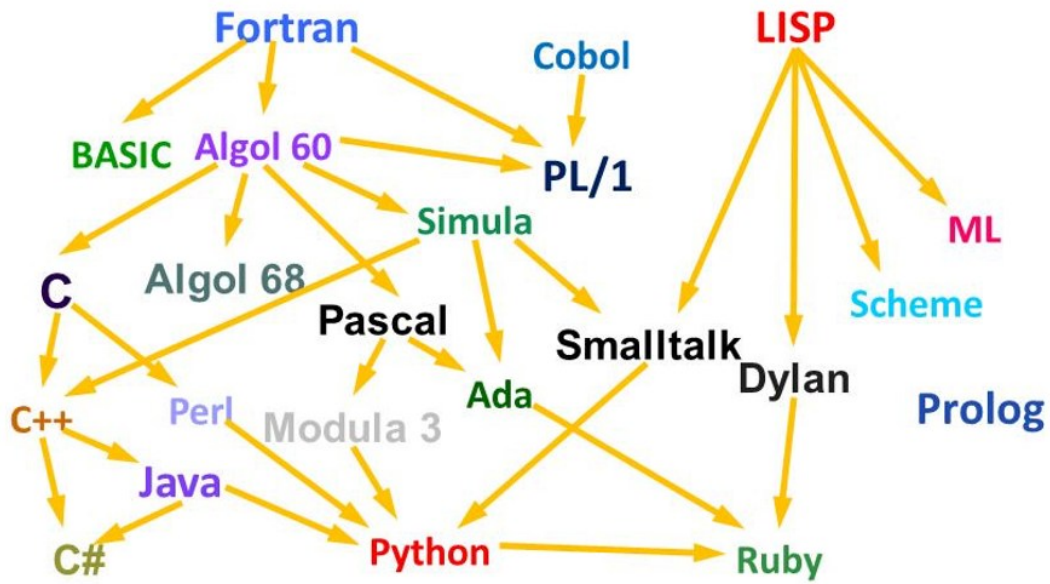


What did we see?

- Basic programming/scripting concepts:
  - Assignment, sequence, choice, iteration, functions
  - Integers, floats, strings, lists
  - Graphical user interfaces, widgets
  - Mark-up languages, web documents
  - Data querying, pattern matching
- Tools to help us code our solutions:
  - An interactive program code development environment
  - Regular expression development tools
  - An interactive database interface

What didn't we cover?

- Almost everything there is in the field of Computer Science!
- More programming/scripting *languages*
  - Function-oriented languages
  - Logic-based languages
  - List-based languages



- System development *processes*
  - Algorithm design
  - Stepwise refinement
  - Test-driven development
- Software *architectures*
  - Enterprise systems
  - Cloud-based applications
- Computing *theory*
  - Program semantics and compilation
  - Software verification (correctness proofs)
  - Software complexity (efficiency) analysis

Part B — The final exam

Structure of the exam

- Duration: 2h40 mins total
  - You must complete all questions by 11:10am
  - There is ample time to complete the exam in this period, even allowing for temporary technical issues
- Online, open book, individual
  - A test of skills and knowledge acquired, *not* memory
  - The exam is a “Timed Online Assessment – Fixed (TOA1)”
- A mix of large (general skills) and small ones (practical knowledge), worth 40% in total

Online, open book, individual assessment

- **“Online”** means you can do the exam anywhere that has a reliable Internet connection
- **“Open book”** means you’re welcome to make use of any of the IFB104 teaching materials, recommended textbooks, and online resources suggested during the semester
- **“Individual”** means you may submit only your own work, not that of a “third party”, including *“another person, a commercial service, or an artificial intelligence tool”*
    - QUT’s penalties for “contract cheating” are severe

Questions and Fires

- The Discussion Board will be **closed** during the exam
  - This is to prevent accidental sharing of exam answers
- Email will be for **urgent personal queries only**
  - We cannot help you figure out the solution to any questions

Relevant topics to study

- Week 1 – Evaluating Expressions
- Week 2 – Functions and Booleans
- Week 4 – Lists and Condition Controlled Iteration
- Week 5 – Unit Tests and Regular Expressions
- Week 6 – Tuples and Dictionaries
- Week 7 – Graphical User Interfaces
- Week 9 – Relational Databases
- Week 10 – Defensive Coding and Exception Handling

Quick Recap

- We’re doing a quick recap of some of the core content
- This is not exhaustive
- You should study beyond the context of these slides!

Week 1 Quick Recap — Evaluating Expressions

Key programming concept: Types

- In daily life we distinguish two *types* of numbers:
  - Whole numbers such as 4, 2, 0 and −3
  - Numbers with a fractional part such as 3.145926, 67.2 and −11.5
- Computer languages distinguish these number types too, calling them *integer* and *floating point* numbers
  - But computers are *very* fussy: integer 4 is *not* considered the same as floating point number 4.0

```
>>> 3 + 4 # add two integers
7
>>> 3.2 * 1.7 # multiply two floats
5.44
>>> type(3 + 4) # query the expression's type
<class 'int'>
>>> type(3.2 * 1.7)
<class 'float'>
>>> 11 / 4 # division producing a float
2.75
>>> 11 // 4 # integer division
2
>>> 11 % 4 # remainder after int division
3
```

Non-Arithmetic Expressions

Arithmetic expressions

- When we use artithmetic operators (eg. +, -, *, /) and numbers, the result is a number.
- The following expressions return either “True” or “False”.

Comparison expressions

- Operators: **==**, !=, >, <, >=, <

Logical expressions

- Operators: and, or, not

Not to be mistaken for a “=“

```
>>> 5 == 5 # comparison between 2 numbers, returns True if they are equal
True
>>> 10 > 5 # comparison between 2 numbers, returns True if the first is higher than the second
True
>>> (5 > 3) and (2 < 4) # combines 2 comparison expressions, returns True if they are both True
True
>>> not (5 == 5) # reverses the result of a logical or comparison expression
False
```

Some more numerical operations and functions

- Apart from the familiar arithmetic operators +, −, ∗ and /, Python offers many more ways of calculating using numbers
- Some operators are expressed using symbols and some using named “functions”

```
>>> 2 ** 3 # exponentiation
8
>>> 2 * 3 + 4
10
>>> 2 * (3 + 4) # brackets to change precedence
14
>>> int(3.4 + 2.3) # return just the int part
5
>>> float(3 * 2) # return a float
6.0
>>> round(1 / 3, 2) # round to 2 decimal places
0.33
>>> # ... etc
```

Key programming concept: Expressions

- The preceding examples illustrate the use of *expressions*
  - Expressions consist of one or more *operands* and zero or more *operators*
  - Each expression returns a *value* of a certain type when it is *evaluated*

```
>>> 4 # a primitive int-valued expression
4
>>> 4 * 7 # a more complex int expression
28
>>> (4 * 7) / 3 # a float-valued expression
9.333333333333334
>>> memory = 9 + 23 # remember this value
>>> memory # another primitive expression
32
>>> memory / 5 # another float expression
6.4
```

Week 2 Quick Recap — Functions & Booleans

Key programming concept: Functions

- A *function* is a *named*, *parameterised* sequence of statements
  - We can refer to a function by its name to invoke or “call” it
  - The statements forming the function’s “body” determine what it does
  - The parameters allow us to customise the function’s behaviour when we call it
  - Functions must be *defined* before they can be *called*
- Groups of related functions can be packaged into modules or “Application Programming Interfaces”


```
>>> # A function for computing heights
>>> def height(distance, angle):
        from math import tan, radians
        height_of_object = \
          round(distance * tan(radians(angle)))
        return height_of_object

>>> # Find the height of a tree
>>> distance_to_tree = 75 # feet
>>> angle_to_top = 40 # degrees
>>> print('The tree is',
          height(distance_to_tree, angle_to_top),
          'feet high')
The tree is 63 feet high
```

Key programming concept: Returning values

- Some functions are called for their side effects only, e.g., to draw some graphics on the screen
- Other functions are defined to perform a calculation on some data and return a result
- A **`return`** statement in a function's body determines what value the function returns when it is called
- A common mistake when coding in Python is to confuse merely printing a value with returning one
  - Both actions *look* the same in IDLE's shell interpreter because it *prints* returned values!

```
# Add emphasis and print the result
def exclaim_1(phrase):
    print(phrase + '!')

# Add emphasis and return the result
def exclaim_2(phrase):
    return phrase + '!'
```

```
>>> # Can use a returned value in an expression
>>> exclaim_2('Hello') + '?'
'Hello!?'
>>> # But this function call returns nothing
>>> exclaim_1('Hello') + '?'
Hello!
Traceback (most recent call last):
  File "<pyshell#125>", line 1, in <module>
    exclaim_1('Hello') + '?'
TypeError: unsupported operand type(s) for +: '
```

Key programming concept: Parameter passing

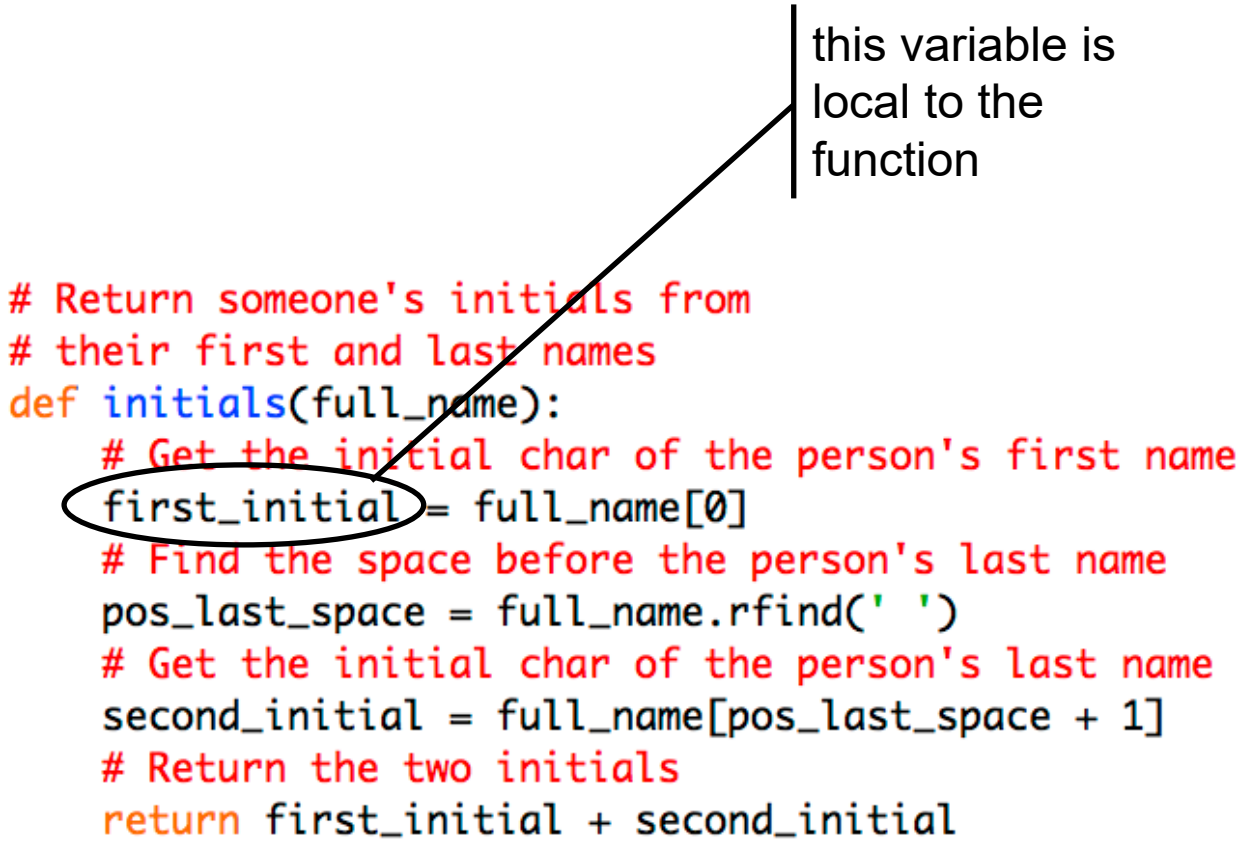
- To make functions customisable to different situations we give them *parameters*
- A function can have zero, one or many parameters
  - The parameter names used in the function definition are the *formal parameters*
  - The values or expressions used in the function call are the *actual parameters*, also called *arguments*
- Calling a function is like running the function's body with each occurrence of the parameters in the code replaced with the corresponding arguments

```
>>> tongue_twister = "Upon a slitted sheet I sit"
>>>
>>> # Return a chosen word in a phrase
>>> def word(index, phrase):
        words = phrase.split()
        return(words[index])

>>> # Find the fourth word (counting from zero)
>>> tongue_twister.split()[3]
'sheet'
>>>
>>> # Find the fourth word (counting from zero)
>>> word(3, tongue_twister)
'sheet'
```

Local variables and scope

- Sometimes a function may require a variable to store a temporary value as part of its job
- Introducing a new variable within a function's body creates a *local variable*
- Local variables and parameters are not accessible to statements outside the function body
  - The *scope* of the "locals" is limited to within the function
- To *assign* to a global variable from within a function we need to declare it as `global` (otherwise it would be treated as a new local variable)



Key programming concept: Relational operators

- Boolean expressions involving numeric values typically use *relational operators*
- Each of these expressions will return `True` or `False`
- Some of these operators work on strings too

Expression	Meaning
x > *y*	Is *x* greater than *y*?
x < *y*	Is *x* less than *y*?
x >= *y*	Is *x* greater than or equal to *y*?
x <= *y*	Is *x* less than or equal to *y*?
x == *y*	Is *x* equal to *y*?
x != *y*	Is *x* not equal to *y*?
x **in** *y*	Does *x* occur in *y*? (Where y is a compound type such as a string or list)

Key programming concept: Relational operators

- Boolean expressions involving numeric values typically use *relational operators*
- Each of these expressions will return `True` or `False`
- Some of these operators work on strings too

Expression	Meaning
x > *y*	Is *x* greater than *y*?
x < *y*	Is *x* less than *y*?
x >= *y*	Is *x* greater than or equal to *y*?
x <= *y*	Is *x* less than or equal to *y*?
x == *y*	Is *x* equal to *y*?
x != *y*	Is *x* not equal to *y*?
x **in** *y*	Does *x* occur in *y*? (Where y is a compound type such as a string or list)

Week 4 Quick Recap — Lists and Loops

Key programming concept: Two reasons for calling a function (or method)

- Most functions/methods accept some arguments and *return* a new value, which can be used in a larger expression, assigned to a variable, or printed to the screen
- However, some functions produce a *side-effect* on variables or the computing environment and return nothing
    - An attempt to access the value returned by a pure side-effecting function gets the special value `None`

```
>>> # import a constant from a module
>>> from math import pi
>>> # evaluate an expression and store the
>>> # value returned
>>> area = pi * (4 ** 2)
>>> area
50.26548245743669
>>>
>>> # define a list of values
>>> directions = ['up', 'down', 'left']
>>> # apply a function to the list that returns
>>> # a result
>>> len(directions)
3
>>> # apply a method to the list that has a
>>> # side-effect
>>> directions.remove('left')
>>> directions
['up', 'down']
```

Key programming concept: Mutable versus immutable variables

- String variables are immutable (unchangeable)
  - All string operations return new strings
  - To update a string variable we must assign a new value to it
- List variables are mutable (changeable)
  - Some list operations return new lists and leave the given list unchanged
  - Other list operations change the list 'in place' when the operation is applied
  - It's hard to remember which of these effects a particular string function has, so keep the *Python Standard Library* manual handy!

```
>>> name = 'Tweedledum' # create a string
>>> name[0:-2] + 'ee' # replace the last two letters
'Tweedledee'
>>> name # but the variable's value is unchanged
'Tweedledum'
>>> name = name[0:-2] + 'ee' # store the changed value
>>> name # now the variable has changed
'Tweedledee'
>>>
>>> letters = ['a', 'b', 'c'] # create a list
>>> letters + ['d'] # return a new list
['a', 'b', 'c', 'd']
>>> letters # but the variable is unchanged
['a', 'b', 'c']
>>> letters.append('d') # use an 'in place' function
>>> letters # now the variable's value has changed
['a', 'b', 'c', 'd']
>>>
```

Key programming concept: Lists of lists

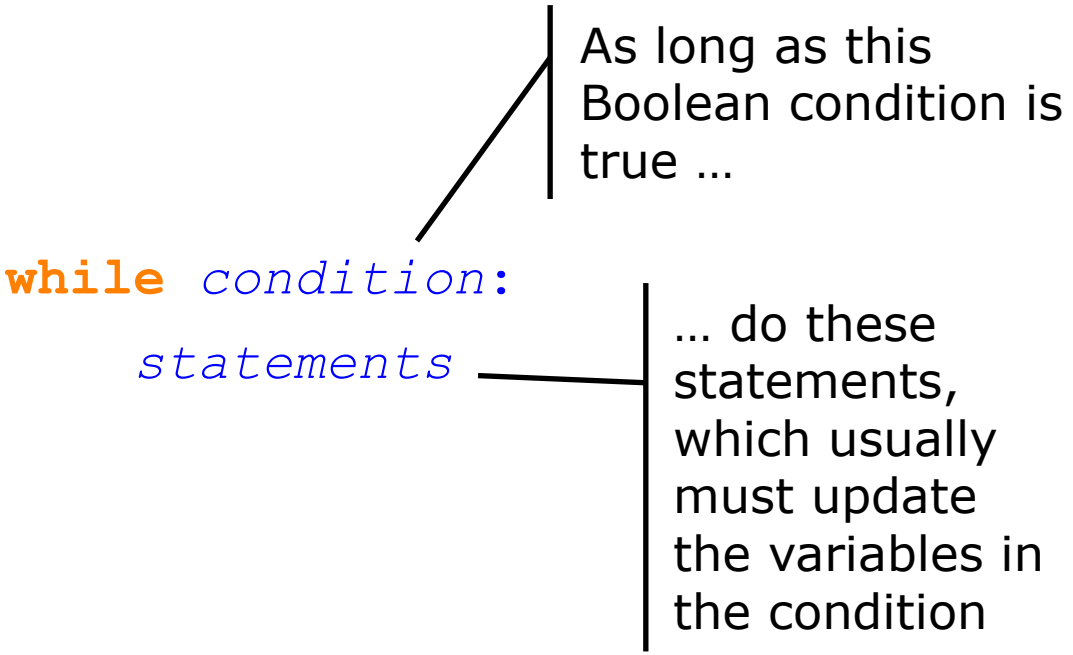
- For-each loops are the basic way of accessing each item in the list, one at a time
  - List items can be accessed either directly via their value or indirectly via their position
- When lists contain ‘sublists’ we can nest for-each loops accordingly

```
>>> # Some elements, grouped into families
>>> elements = [
        # Inert gases
        ['He', 'Ne', 'Ar', 'Kr', 'Xe'],
        # Halogens
        ['F', 'Cl', 'Br', 'I', 'At'],
        # Alkili earth metals
        ['Li', 'Na', 'K', 'Rb', 'Cs']]
>>>
>>> # Print elements with an 'a' in their symbol
>>> for family in elements:
        for symbol in family:
            if 'a' in symbol.lower():
                print(symbol)


Ar
At
Na
```

Key programming concept: Condition-controlled iteration via `while` loops

- In Python condition-controlled iteration is achieved using **`while`** loops:




```
# Set up a variable to control the loop
keep_going = 'yes'

# Calculate and print a series of sales commissions
while keep_going == 'yes':

    # Get salesperson's sales and commission rate
    sales = input('Enter number of sales: ')
    comm_rate = input('Enter commission rate: ')

    # Calculate and display the commission
    commission = sales * comm_rate
    print('Your commission is', commission)

    # See if the user wants to continue
    keep_going = input('More (yes/no)? ')
```

Key programming concept: Exiting loops early

- For efficiency it is sometimes convenient to exit a loop before all iterations have been completed, e.g., when the answer has already been found before the end
- Two ways of doing this in Python are as follows:
  - A **`return`** statement executed from anywhere within a function, including from within a loop, will immediately exit the entire function
  - A **`break`** statement executed in a loop will cause the loop to end immediately
- Doing this makes the number of iterations performed by a 'definite' loop indefinite!

```
>>> palindrome = 'Rats live on no evil star!'
>>>
>>> # Print letters until a space is encountered
>>> for letter in palindrome:
        if letter == ' ':
            break
        else:
            print(letter)


R
a
t
s
>>>
```

Week 5 Quick Recap — Unit Tests & RegEx

Modules

- In Python a *module* is any file containing one or more function definitions
- Given a file *m*`.py` containing function definitions *f*, *g*, *h*, etc we can import all or part of module *m* into our program as desired:

```
from m import f, h  # imports just specific functions f and h
from m import *  # imports all functions from module m
```

Automated testing in Python

- If we need to repeatedly test our program while it is being developed or improved, it is impractical to manually run lots of tests each time the code is changed
- Most modern programming languages provide a way of automatically running a whole suite of tests all at once
- In Python all the tests in a ‘doctest’ string can be run automatically by calling the `testmod` function

```
from doctest import testmod

testmod(...)
```

Example of automated testing

- Using the following code as the main program in a module containing some function definitions and a doctest string will cause the tests to be performed when the module is run
  - However the tests will *not* be performed if the module is imported into another program

```
# If this file is being run as the
# main program, automatically execute
# any tests it contains
if __name__ == '__main__':
    from doctest import testmod
    testmod(verbose = True)
```

Demonstration file: `distance_doctest.py`

Some regular expression primitives

`a`	The letter “a”
`ab`	The letter “a” followed by the letter “b”
`[abc]`	Any one of the letters “a”, “b” or “c”
`.`	Any single character (except newlines, unless in “dotall” mode)
`[b-m]`	Any one of the letters from “b” to “m”, inclusive
`[^xy]`	Any one character *except* the letters “x” or “y”
`^`	The beginning of the string (or of a line in “multi-line” mode)
`$`	The end of the string (or of a line in “multi-line” mode)
`\b`	The beginning or ending of a word
`\s`	Any whitespace character (space, tab, newline, etc)
`\n`	A newline character (in a multi-line string)

Some regular expression operators

$P*$	Pattern P repeated zero or more times
$P+$	Pattern P repeated one or more times
$P?$	Pattern P zero times or once (i.e., P is optional)
$P\{n\}$	Pattern P repeated exactly n times
$P \mid Q$	Pattern P or pattern Q
(...)	Several patterns treated as one group (see later)

Replacing patterns found in strings

- Having found an occurrence of a pattern in a string we often want to do something to it
- Python's `sub` function allows each instance of a pattern to be replaced with another string

```
>>> from re import sub
>>> text = '''
I love my iPad.
iPads are the best!
'''
>>> print(sub('iPad', 'Galaxy Tab', text))

I love my Galaxy Tab.
Galaxy Tabs are the best!
```

Week 6 Quick Recap — Tuples and Dictionaries

Tuples

- Tuples are similar to list in that they contain sequences of values.
  - Example: animals=(turtle, cat, camel)
- However, they are **Immutable**: Tuples cannot be modified after creation. Once created, their elements cannot be changed.
- Ideal for collections of items that should not change, ensuring data integrity.

Feature	List	Tuple
Mutability	Mutable	Immutable
Syntax	[]	()
Example	[1, 2, 3]	(1, 2, 3)
Use Case	Dynamic collections	Fixed collections

Some Function return tuples

- Findall() returns tuples if several parts of the regex are captured
  - Example:

```
>>> input_text = "unicorn 120 190 mouse 8 0.2 dragon 300 1000"
>>> elements = findall('([a-z]+) ([0-9\.]+) ([0-9\.]+)',input_text)
   [('unicorn', '120', '190'), ('mouse', '8', '0.2'), ('dragon', '300',
   '1000')]
```

- To convert a tuple into a list: list(*your_tuple*)

```
>>> list(elements[0])
      ['unicorn', '120', '190']
```

Dictionaries

- Also store collections of items, but
  - **Unordered**: Elements in a dictionary do not have a defined order. There is no index to retrieve items
  - **Key-Value Pairs**: Access elements by their keys, not by index.
  - They are also **mutable**: Can be modified after creation (e.g., adding, removing, or changing key-value pairs).
- They are defined using curly braces {}. Elements are retrieved using the key.

Feature	List	Dictionary
Structure	Elements	Key-Value Pairs
Access	By Index	By Key
Order	Ordered	Unordered (insertion order in Python 3.7+)
Syntax	[]	{}
Example	[1, 2, 3]	{'key': 'value'}

Dictionaries

- They are defined using curly braces {}. Elements are retrieved using the key.

```
>>> animal_dict = {'unicorn':[120, 190], 'mouse':[8, 0.2],
'dragon':[300, 1000]}

>>> animal_dict['unicorn']

      [120, 190]
```

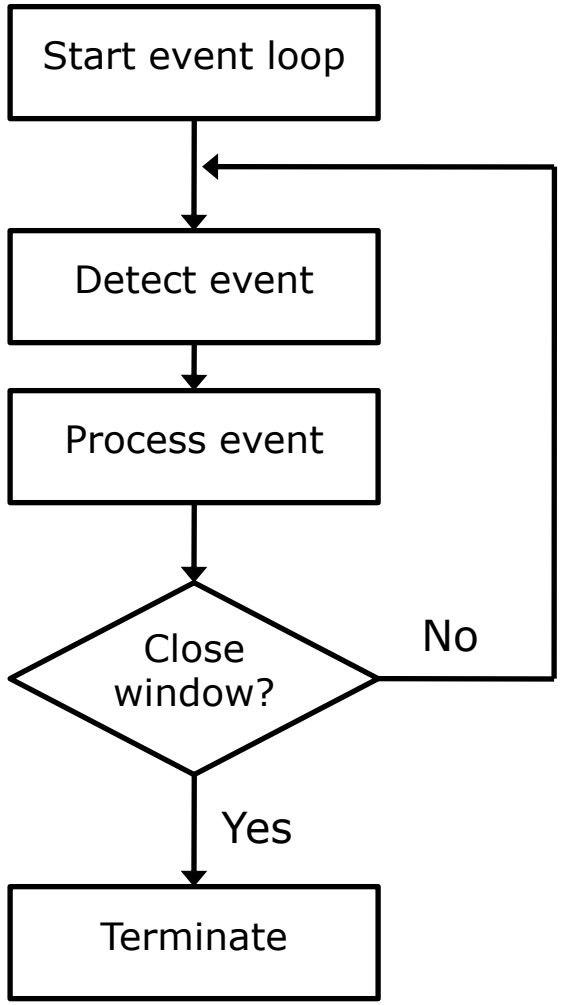
```
>>> for key in animal_dict:
        print(key)
    unicorn
    mouse
    dragon
```

```
>>> for value in animal_dict.values():
        print(value)
    [120, 190]
    [8, 0.2]
    [300, 1000]
```

Week 7 Quick Recap — Graphical User Interfaces

The event loop

- Unlike most of the programs we've developed so far, which run to completion when we start them, GUI programs are "reactive"
- They wait for the user to initiate some event via the interface before doing anything



Some standard Tkinter widgets

- **Button** (to execute a command when pressed)
- **Canvas** (for drawing)
- **Checkbutton** (buttons that toggle; allows several options to be chosen simultaneously)
- **Entry** (text entry field)
- **Frame** (a widget containing other widgets)
- **Label** (to display text or an image)
- **Listbox** (list of options for the user to select)
- **Menu** (pull-down or popup menu at top of window)
- **Menubutton** (pull-down menu inside window)
- **Message** (text display with automatic wrapping)
- **Radiobuttons** (buttons that set a shared variable when pressed; allows one of several options to be chosen)
- **Text** (formatted text display)

Some standard Tkinter widget properties

- **fg** (foreground colour)
- **bg** (background colour)
- **text** (text to display)
- **command** (function to call)
- **font** (text font as name-size pair)
- **justify** (text alignment left, center or right)
- **cursor** (choose the cursor style)
- **variable** (Python variable to set)
- **value** (value of the given variable)

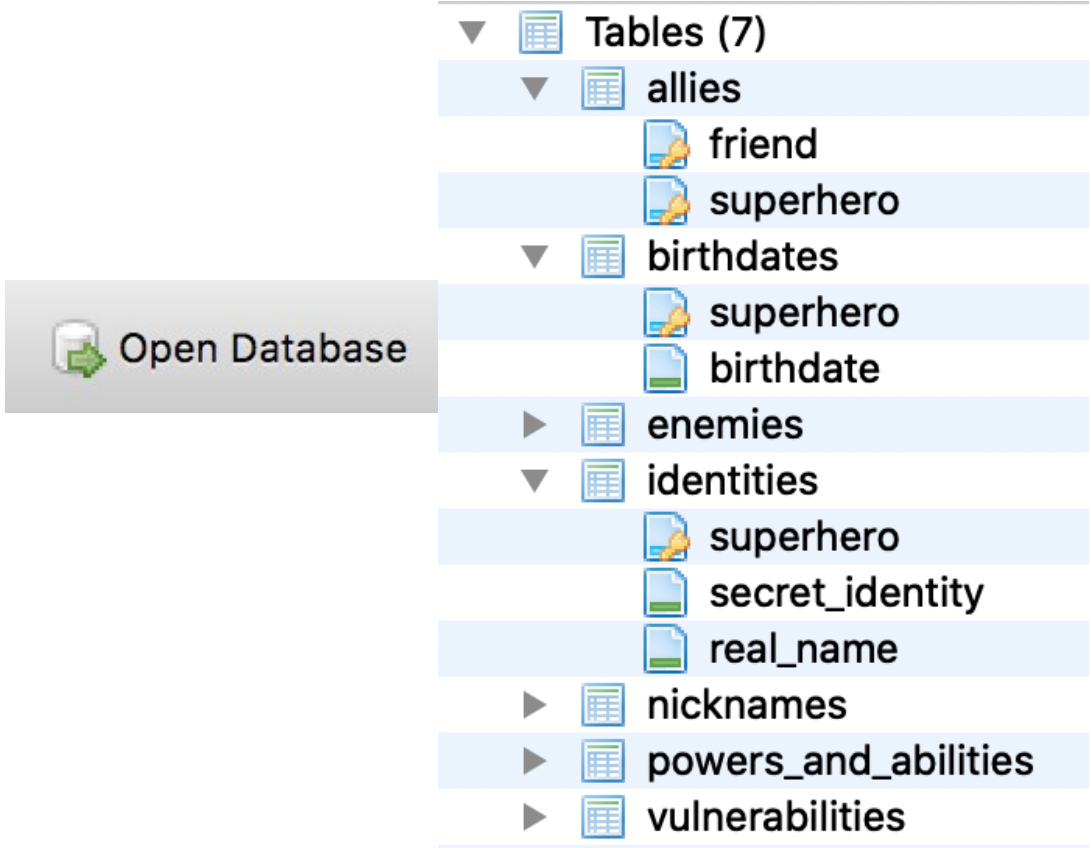
- To change a property of a widget after its creation:

  widget['*property*'] **=** *value*

Week 9 Quick Recap — Relational Databases

Relational databases

- All the data in a *relational database* is stored in a collection of two dimensional *tables* made up of *columns* and *rows*, where each column contains data of a specific *type*
- To open an existing database in the *DB Browser for SQLite* either:
  - Select a “db” file via the Open Database button; or
  - Execute a “dump” script via File > Import > Database from SQL file
  - (Don’t try to drag-and-drop the database into the application because it may challenge you for a password)



Retrieving data from a database

- By far the most common operation we perform on databases is “querying” their contents
- For this reason SQL provides a rich “query” statement, called **SELECT**

- General form of the select statement:

Mandatory

```
SELECT [DISTINCT | ALL] [* | column_list]
FROM table_list
```

Optional

```
[WHERE condition]
[GROUP BY column_list]
[ORDER BY column_list]
```

The order of the clauses cannot be changed

- **WHERE**: filters the rows subject to some condition
- **GROUP BY**: forms groups of rows with the same column value
- **ORDER BY**: specifies the order of the rows
- **DISTINCT** or **ALL**: specifies whether duplicate rows should be returned or not

The anatomy of a SELECT statement

SELECT [**DISTINCT** or **ALL**] [***** or *column_list*]

When you ask a database for information, you can use a **SELECT** statement. It has a few parts you can choose:

- **DISTINCT** or **ALL** (optional):
  - DISTINCT means “Only return unique rows”
  - ALL means “Return all matching rows, including duplicates” (default)
- * or **column_list:**
  - * means “show all columns”
  - A column_list means “only show these specific columns” (like StudentID, FirstName)

SELECT * -> show everything.

SELECT DISTINCT FirstName -> show each unique name only once.

Week 10 Quick Recap — Defensive Coding & Exception Handling

Checking validity of external inputs

- The biggest threat to a program's calculations is receiving incorrect inputs
  - External inputs are beyond the program's control
- A "defensive" program will always check the validity of values received before relying on or using them
  - Here we use the regular expression `match` function to confirm the input is a well-formed integer
- If an invalid value is received the program may:
  - Request another value
  - Use a default value instead

```
# Main program to decide when to fire the retros
while altitude != 0:

    # Read a valid integer from the barometer
    raw_air_pressure = input('Enter barometer reading: ')
    while match('^-?[0-9]+$', raw_air_pressure) == None:
        raw_air_pressure = input('Re-enter barometer reading: ')
    air_pressure = int(raw_air_pressure)

    # Calculate the lander's altitude
    altitude = altimeter(air_pressure)
    # Decide whether or not the retros should be firing
    if altitude == 0 or altitude > 25:
        retros_off()
    else:
        retros_on()
```

```
Enter barometer reading: blob
Re-enter barometer reading: glop
Re-enter barometer reading: 45
-- Retro rockets are off --
```

Checking validity of parameters

- When we introduce parameters to our function definitions we make assumptions about their types and possible values
  - This is especially true in a language like Python where we don't have to declare the types of parameters
- A good coding habit is to check that the parameters provided really have the types and values we expect and to respond sensibly if they don't

```
# Calculate altitude based on atmospheric
# pressure - higher pressure means lower altitude
def altimeter(barometer_reading):

    # Sanity check - air pressure can't be negative
    if barometer_reading < 0:
        # Alert the space agency
        print('** Warning: Barometer failure! **')
        # Return the last known altitude as the best guess!
        return altitude

    # Return the result (if we make it this far!)
    return pressure_at_surface - barometer_reading
```

```
Enter barometer reading: 60
-- Retro rockets are off --
Enter barometer reading: 78
-- Retro rockets are firing --
Enter barometer reading: -10
** Warning: Barometer failure! **
-- Retro rockets are firing --
Enter barometer reading: 90
-- Retro rockets are firing --
```

Allow for unexpected values

- Another defensive programming technique is to anticipate that external input values may not always be precise
  - We should allow for slight errors in external inputs, especially when using floating point numbers

```
Enter barometer reading: 0
-- Retro rockets are off --
Enter barometer reading: 25
-- Retro rockets are off --
Enter barometer reading: 70
-- Retro rockets are off --
Enter barometer reading: 82
-- Retro rockets are firing --
Enter barometer reading: 101
-- Retro rockets are off --
Houston, the Eagle has landed!
```

If our Mars lander descends into a crater this loop may never terminate!

```
# Main program to decide when
# to fire the retros
while altitude != 0:
    ...
```

But this one always will

```
# Main program to decide when
# to fire the retros
while altitude > 0:
    ...
```

This condition would cause the retros to fire if the given altitude was negative

```
# Decide whether or not the retros
# should be firing
if altitude == 0 or altitude > 25:
    retros_off()
else:
    retros_on()
```

But this one turns them off if we appear to have descended below ground level

```
# Decide whether or not the retros
# should be firing
if altitude <= 0 or altitude > 25:
    retros_off()
else:
    retros_on()
```

Raising exceptions

- When the run-time system detects an error during program execution it “raises” (or “throws”) an *exception*
  - This usually results in the program terminating and an error message being displayed
- We can deliberately `raise` an exception ourselves if we believe an unrecoverable problem has occurred in our program
- Doing so alerts the calling code to the problem and gives it the opportunity to deal with the fault

```
# Given a record's diameter in inches,
# determine its speed in revolutions per
# minute
def rpm(diameter):
    if diameter == 10: # inches
        return 78 # revs per min
    if diameter == 7:
        return 45
    if diameter == 12:
        return 33
    # Something's wrong if we get
    # to this point!
    raise ValueError('Unknown record size')

print('Play your record at', rpm(8), 'rpms')
```

```
Traceback (most recent call last):
    raise ValueError('Unknown record size')
ValueError: Unknown record size
```

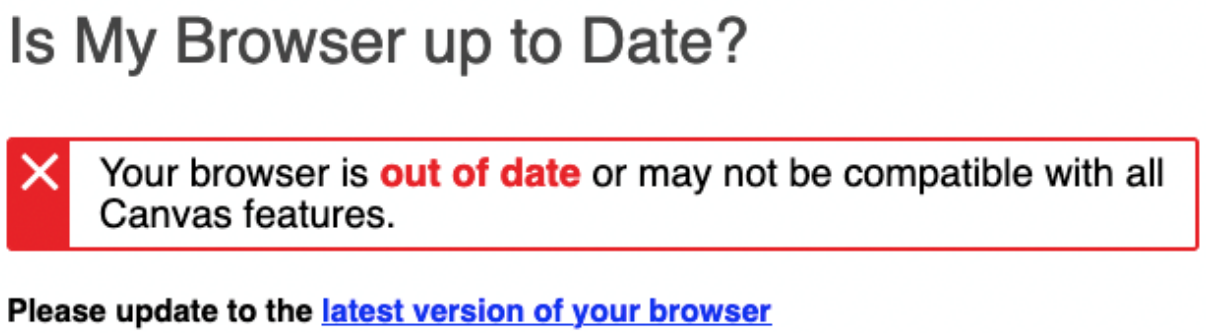
Part C - Practice Exam

Prepare with the practice exam

- The practice exam will be available on Canvas from Wednesday (28/05/25) for those who want to attempt it under realistic conditions
    - Suggested solutions will also be on Canvas but don't look at them until you've attempted the questions yourself
- NB: Unlike the real exam you can restart the practice exam as many times as you like; you can attempt the real exam only once
- NB: The real exam will be on Canvas under Assignments

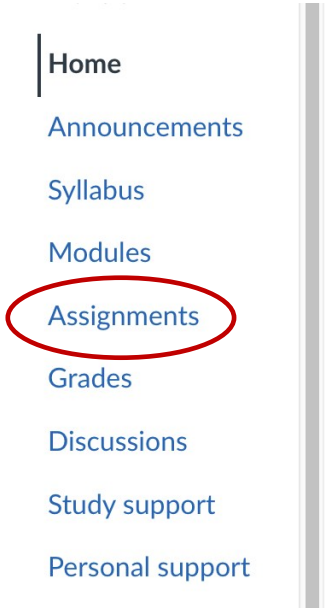
Make sure your web browser is up to date

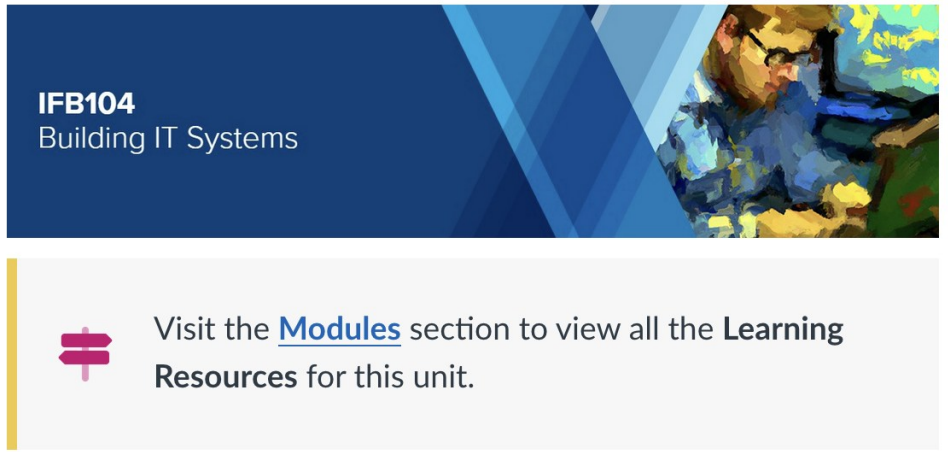
- The IFB104 Final Exam is hosted by the *Canvas* Learning Management System
- Canvas' suppliers say that it is optimised for recent versions of:
  - *Firefox* (excluding Extended Releases)
  - *Chrome*
  - *Edge*
  - *Safari* (Macintosh only)
- More detail about browser compatibility can be found in Canvas' Basics Guide



Accessing the real exam

- The link to the final exam will appear on Canvas at the specified time
  - Look under Assignments
  - The link will *not* be visible at any other time
- Make sure, using the practice exam, that your personal computing environment is adequate for the task
  - You will **not** have time to get technical help from *HiQ* or the teaching team during the exam period itself





Farewell!

- We hope this teaching unit has opened your eyes to the fun of programming
  - It's not magic – you can do it yourself!
- You'll see many other programming styles and techniques in later Information Technology units