

M269 Python, Logic,ADTs

M269 Python, ADTs Prsntn2024J

Contents

1	Agenda	2
2	Adobe Connect	4
2.1	Interface	4
2.2	Settings	4
2.3	Sharing Screen & Applications	6
2.4	Ending a Meeting	6
2.5	Invite Attendees	6
2.6	Layouts	7
2.7	Chat Pods	8
2.8	Web Graphics	8
2.9	Recordings	9
3	Programming	9
3.1	Computational Components	9
3.2	Computation, Programming, Programming Languages	10
3.3	Example Algorithm Design	10
3.4	Binary Search — Exercise	11
3.5	Binary Search — Comparison	11
3.6	Writing Programs & Thinking	12
4	Python	13
4.1	Learning Python	13
4.2	Basic Python	13
4.3	Python Workflows	14
5	Python Checking Tools	16
5.1	allowed Code Checker	16
5.2	Allowed Methods	17
6	Complexity	19
6.1	Complexity Example	21
6.2	Complexity & Python Data Types	23
6.3	Definitions and Rules for Complexity	24
6.3.1	Big-O and Big-Theta Definitions	24
6.3.2	Big-O and Big-Theta Rules	25
6.3.3	Big-Theta Rules — Example	25
6.4	List Comprehensions	25
Activity 1	List Comprehension Exercises	26
6.4.1	Complexity of List Comprehensions	28
6.5	Master Theorem for Divide-and-Conquer Recurrences	32
6.5.1	Master Theorem Example Usage	33
7	Logarithms	35

7.1	Exponentials and Logarithms — Definitions	35
7.2	Rules of Indices	35
7.3	Logarithms — Motivation	36
7.4	Exponentials and Logarithms — Graphs	36
7.5	Laws of Logarithms	37
7.6	Arithmetic and Inverses	37
7.7	Change of Base	39
8	Before Calculators	39
8.1	Log Tables	39
8.2	Slide Rules	42
8.3	Calculators	43
8.4	Example Calculation	45
9	Logic Introduction	45
9.1	Boolean Expressions and Truth Tables	45
9.2	Conditional Expressions and Validity	48
9.3	Boolean Expressions Exercise	48
9.4	Propositional Calculus	51
9.5	Truth Function	52
10	ADTs	55
10.1	Abstract Data Types — Overview	55
10.1.1	Haskell Code — Commentary	57
10.2	Abstract Data Type — Queue	59
10.3	ADT Lists in Lists	63
11	Future Work	69
12	Haskell Example	70
12.1	Binary Search — Haskell — version 1	70
12.2	Binary Search — Haskell — version 2	72
12.3	Binary Search — Haskell — Comparison	73
13	References	73
	References	74

1 Agenda

- Introductions
- Programming — Paradigms and Step-by-Step Guide
- Programming and Python
- Complexity and Big-O/Big-Theta Notation
- ...with a little classical logic
- Abstract Data Type examples
- Implementing Lists in Lists
- A look towards the next topics

- Beware: some topics are included for interest only and are not part of M269
- These notes use recursion, explained at the time
- *Adobe Connect* — if you or I get cut off, wait till we reconnect (or send you an email)
- Time: about 1 hour
- Do ask questions or raise points.
- Slides/Notes [M269Tutorial20241124ProgPythonADTPrsntn2024J](#)

Introductions — Phil

- *Name* Phil Molyneux
- *Background*
 - Undergraduate: Physics and Maths (Sussex)
 - Postgraduate: Physics (Sussex), Operational Research (Brunel), Computer Science (University College, London)
 - Worked in Operational Research, Business IT, Web technologies, Functional Programming
- *First programming languages* [Fortran](#), [BASIC](#), [Pascal](#)
- *Favourite Software*
 - [Haskell](#) — pure functional programming language
 - Text editors [TextMate](#), [Sublime Text](#) — previously [Emacs](#)
 - Word processing in [L^AT_EX](#) — all these slides and notes
 - [Mac OS X](#)
- *Learning style* — I read the manual before using the software

Introductions — You

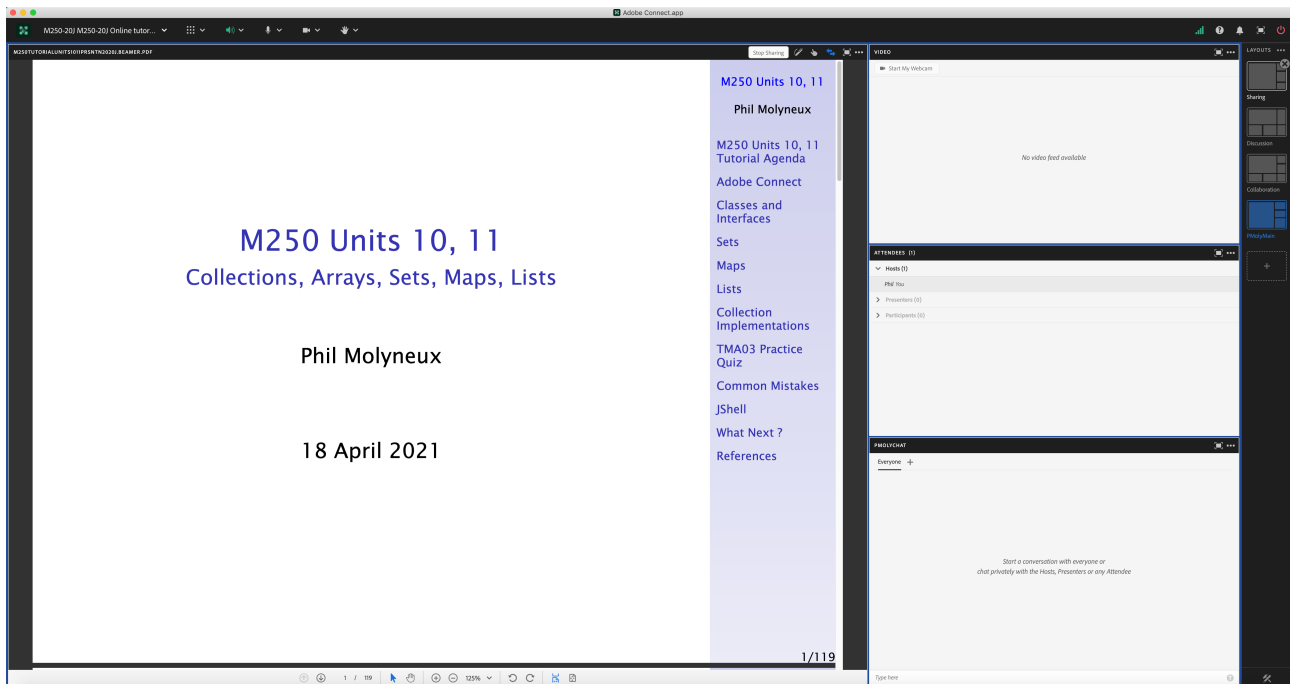
- *Name ?*
- *Favourite software/Programming language ?*
- *Favourite [text editor](#) or [integrated development environment \(IDE\)](#)*
- [List of text editors](#), [Comparison of text editors](#) and [Comparison of integrated development environments](#)
- *Other OU courses ?*
- *Anything else ?*

[Go to Table of Contents](#)

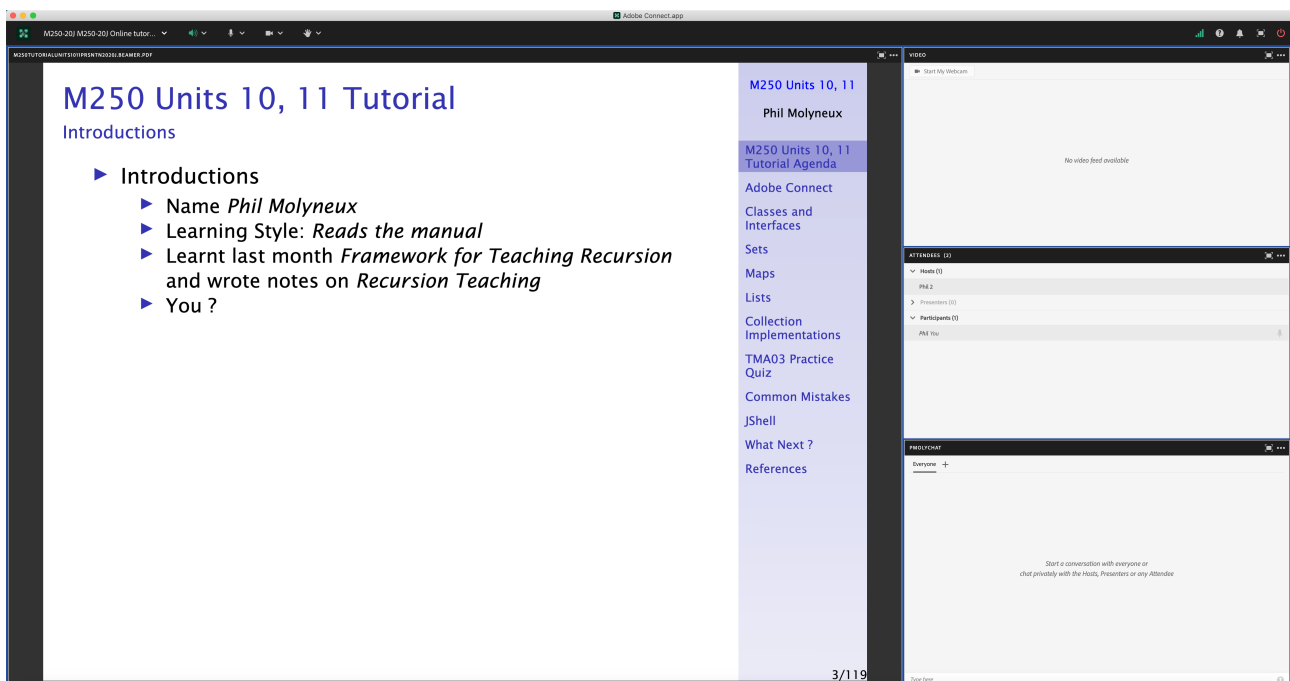
2 Adobe Connect Interface and Settings

2.1 Adobe Connect Interface

Adobe Connect Interface — Host View



Adobe Connect Interface — Participant View



2.2 Adobe Connect Settings

Adobe Connect — Settings

- Everybody **Menu bar** **Meeting** **Speaker & Microphone Setup**
- **Menu bar** **Microphone** **Allow Participants to Use Microphone** ✓

- Check Participants see the entire slide including slide numbers bottom right **Workaround**
 - *Disable Draw* **Share pod** **Menu bar** **Draw icon**
 - *Fit Width* **Share pod** **Bottom bar** **Fit Width icon** ✓
- **Meeting** **Preferences** **General** **Host Cursor** **Show to all attendees**
- **Menu bar** **Video** **Enable Webcam for Participants** ✓
- Do not *Enable single speaker mode*
- Cancel hand tool
- Do not enable green pointer
- **Recording** **Meeting** **Record Session** ✓
- **Documents** Upload PDF with drag and drop to share pod
- Delete **Meeting** **Manage Meeting Information** **Uploaded Content** and **check filename** **click on delete**

Adobe Connect — Access

• Tutor Access

TutorHome **M269 Website** **Tutorials**

Cluster Tutorials **M269 Online tutorial room**

Tutor Groups **M269 Online tutor group room**

Module-wide Tutorials **M269 Online module-wide room**

• Attendance

TutorHome **Students** **View your tutorial timetables**

• Beamer Slide Scaling 440% (422 x 563 mm)

• Clear Everyone's Status

Attendee Pod **Menu** **Clear Everyone's Status**

• Grant Access and send link via email

Meeting **Manage Access & Entry** **Invite Participants...**

• Presenter Only Area

Meeting **Enable/Disable Presenter Only Area**

Adobe Connect — Keystroke Shortcuts

- [Keyboard shortcuts in Adobe Connect](#)
- **Toggle Mic** **⌘** + **M** (Mac), **Ctrl** + **M** (Win) (On/Disconnect)
- **Toggle Raise-Hand status** **⌘** + **E**
- **Close dialog box** **⌘** (Mac), **Esc** (Win)
- **End meeting** **⌘** + ****

2.3 Adobe Connect — Sharing Screen & Applications

- **Share My Screen** > **Application tab** > **Terminal** for **Terminal**
- **Share menu** > **Change View** > **Zoom in** for mismatch of screen size/resolution (Participants)
- (Presenter) Change to 75% and back to 100% (solves participants with smaller screen image overlap)
- Leave the application on the original display
- Beware blue hatched rectangles — from other (hidden) windows or contextual menus
- Presenter screen pointer affects viewer display — beware of moving the pointer away from the application
- First time: **System Preferences** > **Security & Privacy** > **Privacy** > **Accessibility**

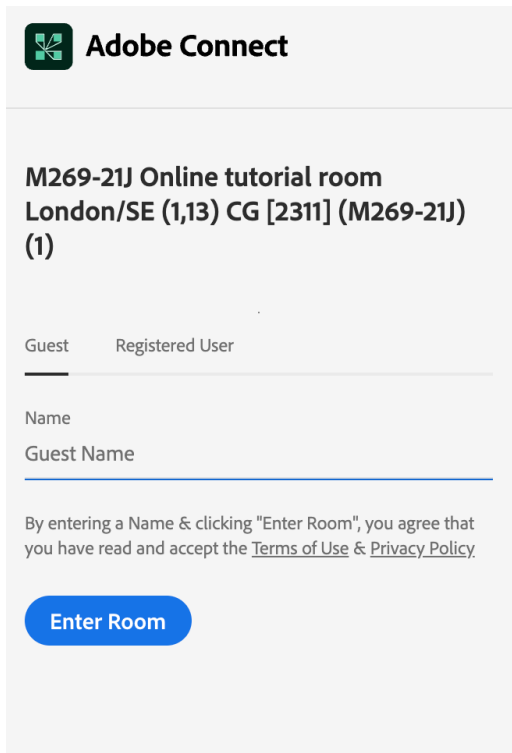
2.4 Adobe Connect — Ending a Meeting

- *Notes for the tutor only*
- **Student:** **Meeting** > **Exit Adobe Connect**
- **Tutor:**
- **Recording** **Meeting** > **Stop Recording** ✓
- **Remove Participants** **Meeting** > **End Meeting...** ✓
 - Dialog box allows for message with default message:
 - *The host has ended this meeting. Thank you for attending.*
- **Recording availability** In course Web site for joining the room, click on the eye icon in the list of recordings under your recording — edit description and name
- **Meeting Information** **Meeting** > **Manage Meeting Information** — can access a range of information in Web page.
- **Delete File Upload** **Meeting** > **Manage Meeting Information** > **Uploaded Content tab** select file(s) and click **Delete**
- **Attendance Report** see course Web site for joining room

2.5 Adobe Connect — Invite Attendees

- **Provide Meeting URL** **Menu** > **Meeting** > **Manage Access & Entry** > **Invite Participants...**
- **Allow Access without Dialog** **Menu** > **Meeting** > **Manage Meeting Information** provides new browser window with **Meeting Information** **Tab bar** > **Edit Information**
- Check *Anyone who has the URL for the meeting can enter the room*
- Default *Only registered users and accepted guests may enter the room*
- **Reverts to default next session but URL is fixed**
- Guests have blue icon top, registered participants have yellow icon top — same icon if URL is open

- See [Start, attend, and manage Adobe Connect meetings and sessions](#)
- Click on the link sent in email from the Host
- Get the following on a Web page
- As *Guest* enter your name and click on **Enter Room**



Adobe Connect

M269-21J Online tutorial room
London/SE (1,13) CG [2311] (M269-21J)
(1)

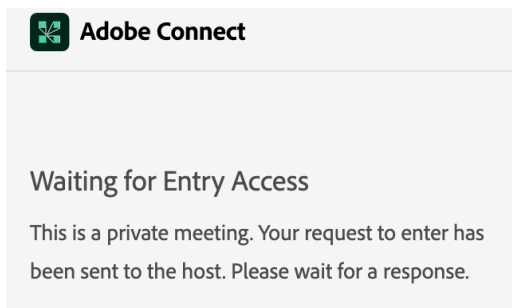
Guest Registered User

Name
 Guest Name

By entering a Name & clicking "Enter Room", you agree that you have read and accept the [Terms of Use](#) & [Privacy Policy](#).

Enter Room

- See the *Waiting for Entry Access* for *Host* to give permission

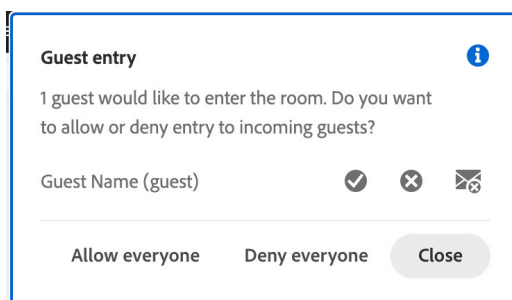


Adobe Connect

Waiting for Entry Access

This is a private meeting. Your request to enter has been sent to the host. Please wait for a response.

- *Host* sees the following dialog in *Adobe Connect* and grants access



Guest entry

1 guest would like to enter the room. Do you want to allow or deny entry to incoming guests?

Guest Name (guest) ✓ ✗ ✉

Allow everyone Deny everyone Close

2.6 Layouts

- **Creating new layouts** example *Sharing* layout

- **Menu** > **Layouts** > **Create New Layout...** > **Create a New Layout dialog** > **Create a new blank layout** and name it *PMolyMain*
- New layout has no Pods but does have Layouts Bar open (see Layouts menu)
- **Pods**
- **Menu** > **Pods** > **Share** > **Add New Share** and resize/position — initial name is *Share n* — rename *PMolyShare*
- **Rename Pod** **Menu** > **Pods** > **Manage Pods...** > **Manage Pods** > **Select** > **Rename** or **Double-click & rename**
- Add Video pod and resize/reposition
- Add Attendance pod and resize/reposition
- Add Chat pod — rename it *PMolyChat* — and resize/reposition
- Dimensions of **Sharing** layout (on 27-inch iMac)
 - Width of Video, Attendees, Chat column 14 cm
 - Height of Video pod 9 cm
 - Height of Attendees pod 12 cm
 - Height of Chat pod 8 cm
- **Duplicating Layouts** does *not* give new instances of the Pods and is probably not a good idea (apart from local use to avoid delay in reloading Pods)
- **Auxiliary Layouts** name *PMolyAuxOn*
 - Create new Share pod
 - Use existing Chat pod
 - Use same Video and Attendance pods

2.7 Chat Pods

- **Format Chat text**
- **Chat Pod** > **menu icon** > **My Chat Color**
- Choices: Red, Orange, Green, Brown, Purple, Pink, Blue, Black
- Note: Color reverts to Black if you switch layouts
- **Chat Pod** > **menu icon** > **Show Timestamps**

2.8 Graphics Conversion for Web

- Conversion of the screen snaps for the installation of Anaconda on 1 May 2020
- Using GraphicConverter 11
- **File** > **Convert & Modify** > **Conversion** > **Convert**
- Select files to convert and destination folder
- Click on **Start selected Function** or  + 

2.9 Adobe Connect Recordings

- **Menu bar** > **Meeting** > **Preferences** > **Video**
- **Aspect ratio** > **Standard (4:3)** (not Wide screen (16:9) default)
- **Video quality** > **Full HD** (1080p not High default 480p)
- **Recording** **Menu bar** > **Meeting** > **Record Session** ✓
- **Export Recording**
- **Menu bar** > **Meeting** > **Manage Meeting Information**
- **New window** > **Recordings** > **check Tutorial** > **Access Type button**
- **check Public** > **check Allow viewers to download**
- **Download Recording**
- **New window** > **Recordings** > **check Tutorial** > **Actions** > **Download File**

[ToC](#)

3 Programming — Computational Components

3.1 Computational Components

Computational Components — Imperative

Imperative or procedural programming has statements which can manipulate global memory, have explicit **control flow** and can be **organised** into procedures (or functions)

- **Sequence** of statements

```
stmtnt ; stmtnt
```

- **Iteration** to repeat statements

```
while expr :
    suite

for targetList in exprList :
    suite
```

- **Selection** choosing between statements

```
if expr : suite
elif expr : suite
else : suite
```

Functional programming treats computation as the evaluation of expressions and the definition of functions (in the mathematical sense)

- **Function composition** to combine the application of two or more functions — like sequence but from right to left (notation accident of history)

```
(f . g) x = f (g x)
```

- **Recursion** — function definition defined in terms of calls to itself (with *smaller* arguments) and base case(s) which do not call itself.

- **Conditional expressions** choosing between alternatives expressions

```
if expr then expr else expr
```

[ToC](#)

3.2 Computation, Programming, Programming Languages

- M269 is not a programming course but ...
- The course uses Python to illustrate various algorithms and data structures
- The final unit addresses the question:
- *What is an algorithm?* What is programming? What is a programming language?
- So it *is* a programming course (sort of)

[ToC](#)

3.3 Example Algorithm Design

Searching

- Given an ordered list (*xs*) and a value (*val*), return
 - Position of *val* in *xs* or
 - Some indication if *val* is not present
- *Simple strategy*: check each value in the list in turn
- *Better strategy*: use the ordered property of the list to reduce the range of the list to be searched each turn
 - Set a range of the list
 - If *val* equals the mid point of the list, return the mid point
 - Otherwise half the range to search
 - If the range becomes negative, report not present (return some distinguished value)

Binary Search Iterative

```

1 def binarySearchIter(xs, val):
2     lo = 0
3     hi = len(xs) - 1
4
5     while lo <= hi:
6         mid = (lo + hi) // 2
7         guess = xs[mid]
8
9         if val == guess:
10            return mid
11        elif val < guess:
12            hi = mid - 1
13        else:
14            lo = mid + 1
15
16    return None

```

Binary Search Recursive

```

17 def binarySearchRec(xs, val, lo=0, hi=-1):
18     if (hi == -1):
19         hi = len(xs) - 1
20
21     mid = (lo + hi) // 2
22
23     if hi < lo:
24         return None
25     else:
26         guess = xs[mid]
27         if val == guess:
28             return mid
29         elif val < guess:
30             return binarySearchRec(xs, val, lo, mid-1)
31         else:
32             return binarySearchRec(xs, val, mid+1, hi)

```

[ToC](#)

3.4 Binary Search — Exercise

Given the *Python* definition of `binarySearchRec` from above, trace an evaluation of `binarySearchRec(xs, 25)` where `xs` is

```
xs = [2, 5, 7, 15, 17, 19, 21, 24, 27, 31, 37, 48, 57, 87, 95]
```

Binary Search — Solution

```

xs = [2, 5, 7, 15, 17, 19, 21, 24, 27, 31, 37, 48, 57, 87, 95]
binarySearchRec(xs, 25)
xs = [2, 5, 7, 15, 17, 19, 21, 24, 27, 31, 37, 48, 57, 87, 95]
binarySearchRec(xs, 25, 8, 14) by line 31
xs = [2, 5, 7, 15, 17, 19, 21, 24, 27, 31, 37, 48, 57, 87, 95]
binarySearchRec(xs, 25, 8, 10) by line 31
xs = [2, 5, 7, 15, 17, 19, 21, 24, 27, 31, 37, 48, 57, 87, 95]
binarySearchRec(xs, 25, 8, 8) by line 29
xs = [2, 5, 7, 15, 17, 19, 21, 24, 27, 31, 37, 48, 57, 87, 95]
binarySearchRec(xs, 25, 8, 7) by line 29
Return value: None by line 23

```

[ToC](#)

3.5 Binary Search — Comparison

- Both forms compare the given value (`val`) to the mid-point value of the range of the list (`xs[mid]`)
- If not found, the range is adjusted via assignment in a `while` loop (iterative) or function call (recursive)
- The recursive version has *default parameter* values to initialise the function call (`evil`, should be a helper function)
- There are two base cases:
 - The value is found (`val == guess`)
 - The range becomes negative (`hi < lo`)

- The return value is either `mid` or `None`
- What is the *type* of the binary search function ?

Binary Search — Performance

- *Linear search* — number of comparisons
 - *Best case* 1 (first item in the list)
 - *Worst case* n (last item)
 - *Average case* $\frac{1}{2}n$
- *Binary search* — number of comparisons
 - *Best case* 1 (middle item in the list)
 - *Worst case* $\log_2 n$ (steps to see all)
 - *Average case* $\log_2 n - 1$ (steps to see half)

[ToC](#)

3.6 Writing Programs & Thinking

The Steps

1. Invent a *name* for the program (or function)
2. What is the *type* of the function ? What sort of *input* does it take and what sort of *output* does it produce ? In Python a type is implicit; in other languages such as Haskell a type signature can be explicit.
3. Invent *names* for the input(s) to the function (*formal parameters*) — this can involve thinking about possible *patterns* or *data structures*
4. What restrictions are there on the input — state the preconditions.
5. What must be true of the output — state the postconditions.
6. *Think* of the definition of the function body.

The *Think* Step

- **How to Think**

1. Think of an example or two — what should the program/function do ?
2. Break the inputs into separate cases.
3. Deal with simple cases.
4. Think about the result — try your examples again.

- **Thinking Strategies**

1. Don't think too much at one go — break the problem down. Top down design, step-wise refinement.
2. What are the inputs — describe all the cases.

3. Investigate choices. What data structures ? What algorithms ?
4. Use common tools — bottom up synthesis.
5. Spot common function application patterns — generalise & then specialise.
6. Look for good *glue* — to combine functions together.

[ToC](#)

4 Python

4.1 Learning Python

- [Python 3 Documentation](#)
- [Python Tutorial](#)
- [Python Language Reference](#)
- [Python Library Reference](#)
- [Hitchhiker's Guide to Python](#)
- [Stackoverflow on Python](#)
- [Martelli et al. \(2022\) *Python in a Nutshell*](#)
- [Lutz \(2025\) *Learning Python*](#)

[ToC](#)

4.2 Basic Python

Python Usage

- How do you enter an interactive Python shell ?
- How do you exit Python in *Terminal* (Mac) or *Command prompt* (Windows) ?
- How do you get help in a shell ?
- How do you exit the *interactive help utility* ?
- How do you enter an interactive Python shell ?
[Windows PythonWin Shell from Toolbox](#); [Mac python3 in Terminal](#)
- How do you exit Python in *Terminal* (Mac) or *Command prompt* (Windows) ?
[quit\(\)](#)
- How do you get help in a shell ?
[help\(\)](#)
- How do you exit the *interactive help utility* ?
[quit](#)

Sequences Indexing, Slices

- `xs[i:j:k]` is defined to be the sequence of items from index `i` to `(j-1)` with step `k`.
- If `k` is omitted or `None`, it is treated as `1`.
- If `i` or `j` are negative then they are relative to the end.
- If `i` is omitted or `None` use `0`.
- If `j` is omitted or `None` use `len(xs)`

Python Quiz — Lists

Given the following definitions

```
xs = [10.9,25,"Phil",3.14,42,1985]
ys = [[5]] * 3
```

Evaluate

```
1 xs[1]
2 xs[0]
3 xs[5]
4 ys
5 xs[1:3]
6 xs[:2]
7 xs[1:-1]
8 xs[-3]
9 xs[:]
10 ys[0].append(4)
```

Python Quiz — Lists — Answers

Given the following definitions

```
xs = [10.9,25,"Phil",3.14,42,1985]
ys = [[5]] * 3
```

Evaluate

```
1 xs[1] == 25
2 xs[0] == 10.9
3 xs[5] == 1985
4 ys == [[5],[5],[5]]
5 xs[1:3] == [25, 'Phil']
6 xs[:2] == [10.9, 'Phil', 42]
7 xs[1:-1] == [25, 'Phil', 3.14, 42]
8 xs[-3] == 3.14
9 xs[:] == [10.9, 25, 'Phil', 3.14, 42, 1985]
10 ys[0].append(4) == [[5, 4], [5, 4], [5, 4]]
```

[ToC](#)

4.3 Python Workflows

Komodo Python Workflow

1. Create *someProgram.py* with assignment statements defining variables and other data along with function definitions.
2. There may be auxiliary files with other definitions (for example, *Python Activity 2.2* has *Stack.py* with the *Stack* class definition) — this uses the *import* statement in *someProgram.py*

```
from someOtherDefinitions import someIdentifier
```

3. Load *someProgram.py* into *Komodo Edit* and use the *Run Python File* macro from the *Toolbox*
4. For further results, edit the file in *Komodo Edit* and use the *Save and Run* macro from the *Toolbox*

Standalone Python Workflow

1. Create *someDefinitions.py* with assignment statements defining variables and function definitions.
2. In *Terminal* (Mac) or *Command Prompt* (Windows), navigate to *someDefinitions.py* and invoke the *Python 3* interpreter
3. Load *someDefinitions.py* into *Python 3* with one of

```
from someDefinitions import *
```

```
import someDefinitions as sdf
```

The `as sdf` gives a shorter qualifier for the namespace — names in the file are now `sdf.x`

Note that the commands are executed — any print statement will execute

4. At the *Python 3* interpreter prompt, evaluate expressions (may have side effects and alter definitions)

Standalone Python Workflow 2

1. For further results, edit the file in *Your Favourite Editor* and use one of the following commands:

```
reload(sdf)

import imp
imp.reload(sdf)
```

Note the use of the name `sdf` as opposed to the original name.

Read the following references about the dangers of reloading as compared to re-cycling *Python 3*

- [How to re import an updated package while in Python Interpreter?](#)
- [How do I unload \(reload\) a Python module?](#)
- [Reloading Python modules](#)
- [How to dynamically import and reimport a file containing definition of a global variable which may change anytime](#)

5 Python Checking Tools

- M269 provides some Python checking tools: [ruff](#) and [allowed](#) — a description is in the M269 Book section 5.3.2 (these notes are based on Jason Clarke's notes)
- M269 software installation is documented at dsa-ou.github.io/m269-installer/
- [allowed](#) is documented at dsa-ou.github.io/allowed/ — it checks for permitted code
- [ruff](#) Web site is docs.astral.sh/ruff/ — Ruff is an extremely fast Python linter and code formatter, written in Rust
- Both [allowed](#) and [ruff](#) are installed in the standard M269 24J software install — they are in the [venvs](#) folder, wherever that was installed (in my case in my home folder)
- The intention is that [allowed](#) checks you are only using Python contained in the M269 Book and [ruff](#) comments on Python style issues such as the extent to which your code complies with [PEP 8 — Style Guide for Python Code](#)

5.1 allowed Code Checker

- [allowed](#) uses a data file [m269-24j.json](#) to determine if the code has allowed features only
- [JSON \(JavaScript Object Notation\)](#) is a lightweight data-exchange format — effectively it is one up from CSV (Comma Separated Values)
- JSON has a Web site json.org/json-en.html which contains the definitive standard — JSON is not part of M269 and you do not need to read the documentation for the course but since JSON is so widely used you may be interested
- An appendix to Douglas Crockford's book has more on JSON (Crockford was one of the original movers behind JSON) — see [Crockford \(2008, Appendix E\) JavaScript: The Good Parts](#)
- Activate [allowed](#) (and [ruff](#)) by running the cell that appears early on in your Jupyter Notebook which contains some [IPython Magic Commands](#) (see examples below)
- When you run any Python cells, you will see any output from [allowed](#) and [ruff](#) as well as your own output
- If all your usage is allowed you will see no output from [allowed](#)
- If you have used something outside the permitted code you may see a message from [allowed](#)
- Note that the [allowed](#) tool is not perfect and there are some differences between Windows, macOS and Linux users (see examples below)
- All the allowed code is described or listed in the *Summary* sections at the end of each chapter in the M269 Book (or you could read the [m269-24j.json](#) file if you are very relaxed)
- The TMA Jupyter Notebooks seem to have several versions of the software that activates [allowed](#) and [ruff](#) so this section of these notes may be subject to change
- From Section 5.3.2 of the M269 Book there are cells that has the following code

```
%load_ext algaesup.magics
%ruff on
```

```
import platform # allowed

if platform.system() in ('Linux', 'Darwin'):
    %allowed on --config m269-24j --unit 5 --method
else:
    %allowed on --config m269-24j --unit 5
```

- TMA01 has the following

```
%load_ext algaesup.magics
%allowed on
%ruff on
%run -i m269_test
```

- The first version activates `allowed` for code in the first 5 chapters and methods are checked in Linux and macOS
- The second activates for code in all chapters

5.2 Allowed Methods

- **Allowed Methods** a *method* is of the form

`objectName.methodName(args)`

- Examples

```
Python3>>> myList = [1,2,3]
Python3>>> myList.append(5) # using list append method
Python3>>> print(myList)
[1, 2, 3, 5]
Python3>>> myText = "abc"
Python3>>> myText.upper() # using string upper method
'ABC'
Python3>>> print(myText)
abc
```

- Some methods return values, others have side effects
- Read the Python documentation for the details
- From `m269-25j.json` we have the following allowed methods
- **List** insert, append, pop, sort
- **Dict** items, pop
- **Set** add, discard, union, intersection, difference, pop
- Note that **no string methods** are allowed

```
# List
myList = [1,2,3]

print(myList.count(2))
myList.extend([3,4])
myList.remove(2)
print(myList.index(1))
myList.reverse()
myList.clear()

1
0
```

allowed found issues

- 5: `list.count()`
- 6: `list.extend()`
- 7: `list.remove()`
- 8: `list.index()`
- 9: `list.reverse()`
- 10: `list.clear()`

```
# Anything on strings...

myText = "Hello"
print(myText.upper())
print(myText.find("e"))
print(myText.endswith("o"))
print(myText.isdigit())
sep = " "
print(sep.join(["A", "B", "C"]))

HELLO
1
True
False
A,B,C
```

allowed found issues

- 4: `str.upper()`
- 5: `str.find()`
- 6: `str.endswith()`
- 7: `str.isdigit()`
- 9: `str.join()`
- The above examples may not produce error messages on Windows platforms
- Expressions with literal object or literal arguments may be not-allowed but may not generate error messages
- Missing type hints for function definitions may result in non-allowed being missed
- The following are non-allowed but produce no messages

```
print([1,2,3].index(1)) # method on a literal

print("abc".upper()) # method on a literal
print(",".join(["A", "B", "C"])) # literal argument

def test(txt):
    return txt.isdigit() # Missing type hint cso missed

def anotherTest(txt : str):
    print(txt[0].upper()) # type hint - argument is an expression

0
ABC
A,B,C
```

- **User Defined Methods**

- **allowed** will not generate error messages to these

- So you can write methods (or functions) to replace the the functionality of some other methods

6 Complexity and Big O Notation

- Measuring program complexity introduced in section 4 of M269 Unit 2
- See also Miller and Ranum chapter 2 [Big-O Notation](#)
- See also [Wikipedia: Big O notation](#)
- See also [Big-O Cheat Sheet](#)
- Complexity of algorithm measured by using some surrogate to get rough idea
- In M269 mainly using assignment statements
- For *exact* measure we would have to have cost of each operation, knowledge of the implementation of the programming language and the operating system it runs under.
- But mainly interested in the following questions:
- (1) Is algorithm A more efficient than algorithm B for large inputs ?
- (2) Is there a lower bound on any possible algorithm for calculating this particular function ?
- (3) Is it always possible to find a polynomial time (n^k) algorithm for any function that is computable
- — the later questions are addressed in Unit 7

Orders of Common Functions

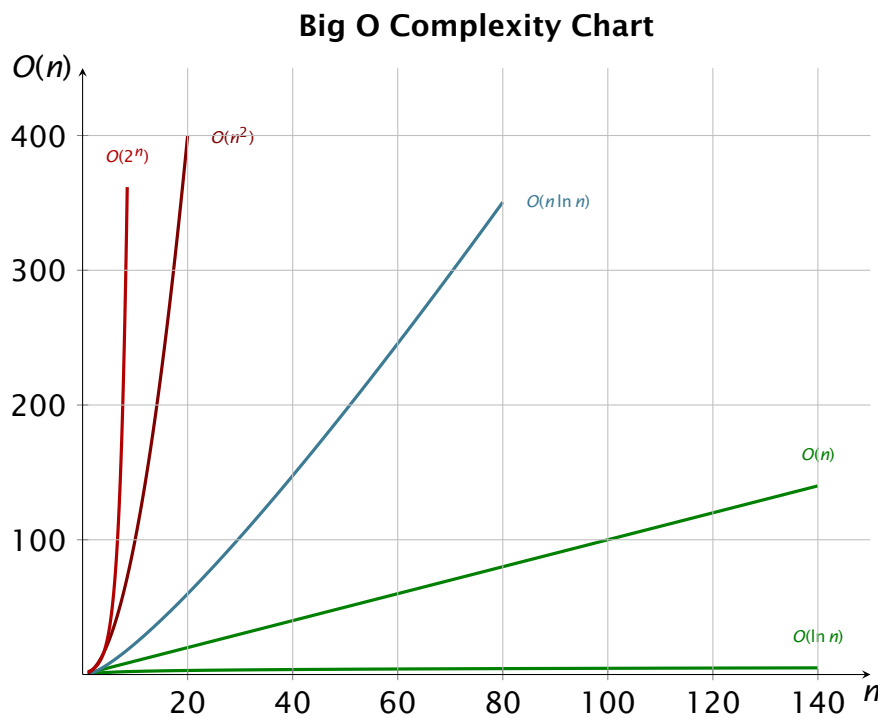
- $O(1)$ **constant** — [look-up table](#)
- $O(\log n)$ **logarithmic** — [binary search](#) of sorted array, binary search tree, binomial heap operations
- $O(n)$ **linear** — searching an unsorted list
- $O(n \log n)$ **loglinear** — [heapsort](#), [quicksort](#) (best and average), [merge sort](#)
- $O(n^2)$ **quadratic** — [bubble sort](#) (worst case or naive implementation), Shell sort, [quicksort](#) (worst case), [selection sort](#), [insertion sort](#)
- $O(n^c)$ **polynomial**
- $O(c^n)$ **exponential** — [travelling salesman problem](#) via [dynamic programming](#), determining if two logical statements are equivalent by brute force
- $O(n!)$ **factorial** — TSP via brute force.

Tyranny of Asymptotics

- Table from [Bentley \(1984, page 868\)](#)
- Cubic algorithm on Cray-1 supercomputer with $\text{FORTRAN3.0}n^3$ nanoseconds

- Linear algorithm on TRS-80 micro with BASIC $19.5n \times 10^6$ nanoseconds

N	Cray-1	TRS-80
10	3.0 microsecs	200 millisecs
100	3.0 millisecs	2.0 secs
1000	3.0 secs	20 secs
10000	49 mins	3.2 mins
100000	35 days	32 mins
1000000	95 yrs	5.4 hrs



Big O Notation

- Abuse of notation — we write $f(x) = O(g(x))$
- but $O(g(x))$ is the class of all functions $h(x)$ such that $|h(x)| \leq C|g(x)|$ for some constant C
- So we should write $f(x) \in O(g(x))$ (but we don't)
- We ought to use a notation that says that (informally) the function f is bounded both above and below by g asymptotically
- This would mean that for big enough x we have

$$k_1 g(x) \leq f(x) \leq k_2 g(x) \text{ for some } k_1, k_2$$
- This is Big Theta, $f(x) = \Theta(g(x))$
- But we use Big O to indicate an asymptotically tight bound where Big Theta might be more appropriate
- See [Wikipedia: Big O Notation](#)
- This could be Maths phobia generated confusion

6.1 Complexity Example

```

5 def someFunction(aList) :
6     n = len(aList)
7     best = 0
8     for i in range(n) :
9         for j in range(i + 1, n + 1) :
10             s = sum(aList[i:j])
11             best = max(best, s)
12     return best

```

- Example from M269 Unit 2 page 46
- Code in file [M269TutorialProgPythonADT.py](#)
- What does the code do ?
- (It was a *famous* problem from the late 1970s/early 1980s)
- Can we construct a more efficient algorithm for the same computational problem ?
- The code calculates the maximum subsegment of a list
- Described in [Bentley \(1984\)](#), [Bentley \(1986](#), column 7), [Bentley \(2000](#), column 8) Also in [Gries \(1989\)](#)
- These are all in a procedural programming style (as in C, Java, Python)
- Problem arose from medical image processing.
- A functional approach using Haskell is in [Bird \(1998](#), page 134), [Bird \(2014](#), page 127, 133) — a variant on this called the *Not the maximum segment sum* is given in [Bird \(2010](#), Page 73) — both of these *derive* a linear time program from the (n^3) initial specification
- See [Wikipedia: Maximum subarray problem](#)
- See [Rosetta Code: Greatest subsequential sum](#)
- Here is the same program but modified to allow lists that may only have negative numbers
- The complexity $T(n)$ function will be slightly different
- but the Big O complexity will be the same

```

14 def maxSubSeg01(xs) :
15     n = len(xs)
16     maxSoFar = xs[0]
17     for i in range(1,n) :
18         for j in range(i + 1, n + 1) :
19             s = sum(xs[i:j])
20             maxSoFar = max(maxSoFar, s)
21     return maxSoFar

```

- Complexity function $T(n)$ for [maxSubSeg01\(\)](#)
- Two initial assignments
- The outer loop will be executed $(n - 1)$ times,
- Hence the inner loop is executed

$$(n - 1) + (n - 2) + \dots + 2 + 1 = \frac{(n - 1)}{2} \times n$$
- Assume [sum\(\)](#) takes n assignments

- Hence $T(n) = 2 + (n+2) \times \left(\frac{(n-1)}{2} \times n \right)$

$$= 2 + (n+2) \times \left(\frac{n^2}{2} - \frac{n}{2} \right)$$

$$= 2 + \frac{1}{2}n^3 - \frac{1}{2}n^2 + n^2 - n$$

$$= \frac{1}{2}n^3 + \frac{1}{2}n^2 - n + 2$$

- Hence $O(n^3)$

- Developing a better algorithm**

- Assume we know the solution (`maxSoFar`) for `xs[0..(i - 1)]`
- We extend the solution to `xs[0..i]` as follows:
 - The maximum segment will be either `maxSoFar`
 - or the sum of a sublist ending at `i` (`maxToHere`) if it is bigger
- This reasoning is similar to divide and conquer in binary search or [Dynamic programming](#) (see Unit 5)
- Keep track of both `maxSoFar` and `maxToHere` — **the Eureka step**
- Developing a better algorithm `maxSubSeg02()`**

```

27 def maxSubSeg02(xs) :
28     maxToHere = xs[0]
29     maxSoFar = xs[0]
30     for x in xs[1:] :
31         # Invariant: maxToHere, maxSoFar OK for xs[0..(i-1)]
32         maxToHere = max(x, maxToHere + x)
33         maxSoFar = max(maxSoFar, maxToHere)
34     return maxSoFar

```

- Complexity function $T(n) = 2 + 2n$
- Hence $O(n)$
- What if we want more information ?
- Return the (or a) segment with max sum and position in list

```

38 def maxSubSeg03(xs) :
39     maxSoFar = maxToHere = xs[0]
40     startIdx, endIdx, startMaxToHere = 0, 0, 0
41     for i, x in enumerate(xs) :
42         if maxToHere + x < x :
43             maxToHere = x
44             startMaxToHere = i
45         else :
46             maxToHere = maxToHere + x
47
48         if maxSoFar < maxToHere :
49             maxSoFar = maxToHere
50             startIdx, endIdx = startMaxToHere, i
51
52     return (maxSoFar, xs[startIdx:endIdx+1], startIdx, endIdx)

```

- Developing a better algorithm `maxSubSeg03()`**
- Complexity function worst case $T(n) = 2 + 3 + (2 + 3)n$
- Hence still $O(n)$

- Note [Python assignments](#), `enumerate()` and `tuple`
- Sample data and output

```

56 egList = [-2,1,-3,4,-1,2,1,-5,4]
58 egList01 = [-1,-1,-1]
60 egList02 = [1,2,3]
62 assert maxSubSeg03(egList) == (6, [4, -1, 2, 1], 3, 6)
64 assert maxSubSeg03(egList01) == (-1, [-1], 0, 0)
66 assert maxSubSeg03(egList02) == (7, [1, 2, 3], 0, 2)

```

[ToC](#)

6.2 Complexity & Python Data Types

Lists

Operation	Notation	Average	Amortized Worst
Get item	<code>x = xs[i]</code>	$O(1)$	$O(1)$
Set item	<code>xs[i] = x</code>	$O(1)$	$O(1)$
Append	<code>xs = xs + xs</code>	$O(1)$	$O(1)$
Copy	<code>xs = xs[:]</code>	$O(n)$	$O(n)$
Pop last	<code>xs.pop()</code>	$O(1)$	$O(1)$
Pop other	<code>xs.pop(i)</code>	$O(k)$	$O(k)$
Insert(i,x)	<code>xs[i:i] = [x]</code>	$O(n)$	$O(n)$
Delete item	<code>del xs[i:i+1]</code>	$O(n)$	$O(n)$
Get slice	<code>xs = xs[i:j]</code>	$O(k)$	$O(k)$
Set slice	<code>xs[i:j] = xs</code>	$O(k + n)$	$O(k + n)$
Delete slice	<code>xs[i:j] = []</code>	$O(n)$	$O(n)$
Member	<code>x in xs</code>	$O(n)$	
Get length	<code>n = len(xs)</code>	$O(1)$	$O(1)$
Count(x)	<code>n = xs.count(x)</code>	$O(n)$	$O(n)$

- Source <https://wiki.python.org/moin/TimeComplexity>
- See <https://docs.python.org/3/library/stdtypes.html#sequence-types-list-tuple-range>

Bags

```

5 class Bag:
7     def __init__(self):
8         self.list = []
10
11     def add(self, item):
12         self.list.append(item)
13
14     def remove(self, item):
15         self.list.remove(item)
16
17     def contains(self, item):
18         return item in self.list
19
20     def count(self, item):
21         return self.list.count(item)
22
23     def size(self):
24         return len(self.list)

```

```

25 def __str__(self):
26     return str(self.list)

```

Information Retrieval Functions

- **Term Frequency**, `tf`, takes a string, `term`, and a Bag, `document`
returns occurrences of `term` divided by total strings in `document`
- **Inverse Document Frequency**, `idf`, takes a string, `term`, and a list of Bags, `documents`
returns $\log(\text{total}/(1 + \text{containing}))$ — `total` is total number of Bags, `containing` is the number of Bags containing `term`
- **tf-idf**, `tf_idf`, takes a string, `term`, and a list of Bags, `documents`
returns a sequence $[r_0, r_1, \dots, r_{n-1}]$ such that $r_i = \text{tf}(\text{term}, d_i) \times \text{idf}(\text{term}, \text{documents})$

[ToC](#)

6.3 Definitions and Rules for Complexity

6.3.1 Big-O and Big-Theta Definitions

- We compare the functions implementing algorithms by looking at the asymptotic behaviour of the functions for large inputs.
- If f and g are functions taking natural numbers as input (the problem size) and returning nonnegative results (the effort required in the calculations.)
- f is of *order* g and write $f = \Theta(g)$, if there are positive constants k_1 and k_2 and a natural number n_0 such that

$$k_1 g(n) \leq f(n) \leq k_2 g(n) \text{ for all } n > n_0$$

This means that some multipliers times $g(n)$ provide upper and lower bounds to $f(n)$

- If we only wanted an upper bound on the values of a function, then you can use Big- O notation.
- We say f is of *order at most* g and write $f = O(g)$, if there is a positive constant k and a natural number n_0 such that

$$f(n) \leq kg(n) \text{ for all } n > n_0$$

- Note that the notation is heavily abused:

Many authors use Big- O notation when they really mean Big- Θ notation

We really should define the Θ notation to say that $\Theta(g)$ denotes the set of all functions f with the stated property and write $f \in \Theta(g)$ — however the use of $f = \Theta(g)$ is traditional

- The next section gives some rules for manipulating the notation to calculate overall complexities of functions from their component parts — this also abuses the notation for equality

Based on [Bird and Gibbons \(2020, page 25\)](#) *Algorithm Design with Haskell* and [Graham et al. \(1994, page 450\)](#) *Concrete Mathematics: A Foundation for Computer Science*

[ToC](#)

6.3.2 Big-O and Big-Theta Rules

- $n^p = O(n^q)$ where $p \leq q$

This has some surprising consequences — $n = O(n)$ and $n = O(n^2)$ — remember Big-O just gives upper bounds.

- $O(f(n)) + O(g(n)) = O(|f(n)| + |g(n)|)$
- $\Theta(n^p) + \Theta(n^q) = \Theta(n^q)$ where $p \leq q$
- $f(n) = \Theta(f(n))$
- $c \cdot \Theta(f(n)) = \Theta(f(n))$ if c is constant
- $\Theta(\Theta(f(n))) = \Theta(f(n))$
- $\Theta(f(n))\Theta(g(n)) = \Theta(f(n)g(n))$
- $\Theta(f(n)g(n)) = f(n)\Theta(g(n))$

[ToC](#)

6.3.3 Big-Theta Rules — Example

```

1 def numVowels(txt : str) -> int ;
2     """Find the number of vowels in text
3
4     """
5
6     vowelCount = 0
7     vowels = "aeiouAEIOU"
8
9     for ch in txt :
10         if ch in vowels :
11             vowelCount = vowelCount + 1
12     return vowelCount

```

- The rules give

$$\Theta(1) + \Theta(1) + \Theta(n) \times \Theta(|\text{vowels}|) \times \Theta(1)$$
 where $n = |\text{txt}|$
- Since $|\text{vowels}| = 10$ the overall complexity is $\Theta(n)$

[ToC](#)
[ToC](#)

6.4 List Comprehensions

List Comprehensions — Python

- [List Comprehensions](#) (tutorial), [List Comprehensions](#) (reference) provide a concise way of performing calculations over lists (or other iterables)

- Example: Square the even numbers between 0 and 9

```
Python3>>> [x ** 2 for x in range(10) if x % 2 == 0]
[0, 4, 16, 36, 64]
```

- Example: List all pairs of integers (x, y) such that $x < 4$, $y < 4$ and x is divisible by 2 and y is divisible by 3

```
Python3>>> [(x,y) for x in range(4)
...           for y in range(4)
...           if x % 2 == 0
...           and y % 3 == 0]
[(0, 0), (0, 3), (2, 0), (2, 3)]
Python3>>>
```

- In general

```
[expr for target1 in iterable1 if cond1
     for target2 in iterable2 if cond2 ...
     for targetN in iterableN if condN ]
```

- Lots example usage in the algorithms below

List Comprehensions — Haskell

- **List Comprehensions** provide a concise way of performing calculations over lists
- Example: Square the even numbers between 0 and 9

```
GHCI> [x^2 | x <- [0..9], x `mod` 2 == 0]
[0,4,16,36,64]
GHCI>
```

- In general

```
[expr | qual1, qual2, ..., qualN]
```

- The qualifiers `qual` can be
 - Generators `pattern <- list`
 - Boolean guards — acting as filters
 - Local declarations with `let decls` for use in `expr` and later generators and boolean guards

Activity 1 (a) Stop Words Filter

- **Stop words** are the most common words that most search engines avoid: 'a', 'an', 'the', 'th
- Using list comprehensions, write a function `filterStopWords` that takes a list of words and filters out the stop words
- Here is the initial code

```
11 sentence \
12   = "the_quick_brown_fox_jumps_over_the_lazy_dog"
14 words = sentence.split()
16 wordsTest \
17   = (words == ['the', 'quick', 'brown'
18               , 'fox', 'jumps', 'over'
19               , 'the', 'lazy', 'dog'])
21 stopWords \
22   = ['a', 'an', 'the', 'that']
```

[Go to Answer](#)**Activity 1 (a) Stop Words Filter**

```

11 sentence \
12 = "the_quick_brown_fox_jumps_over_the_lazy_dog"
14 words = sentence.split()
16 wordsTest \
17 = (words == ['the', 'quick', 'brown'
18             , 'fox', 'jumps', 'over'
19             , 'the', 'lazy', 'dog'])
21 stopWords \
22 = ['a', 'an', 'the', 'that']

```

- Notice the [Python Explicit line joining](#) with (`\<n1>`) and [Python Implicit line joining](#) with (`(...)`)
- The [backslash](#) (`\`) must be followed by an [end of line character](#) (`<n1>`)
- The (`'_'`) symbol represents a space (see Unicode U+2423 Open Box)

[Go to Answer](#)**Activity 1 (b) Transpose Matrix**

- A matrix can be represented as a list of rows of numbers
- We *transpose* a matrix by swapping columns and rows
- Here is an example

```

38 matrixA \
39 = [[1, 2, 3, 4]
40    , [5, 6, 7, 8]
41    , [9, 10, 11, 12]]
43 matATr \
44 = [[1, 5, 9]
45    , [2, 6, 10]
46    , [3, 7, 11]
47    , [4, 8, 12]]

```

- Using list comprehensions, write a function `transMat`, to transpose a matrix

[Go to Answer](#)**Activity 1 (c) List Pairs in Fair Order**

- Write a function which takes a pair of positive integers and outputs a list of all possible pairs in those ranges
- If we do this in the simplest way we get a bias to one argument
- Here is an example of a bias to the second argument

```

68 yBiasLstTest \
69 = (yBiasListing(5,5)
70    == [(0, 0), (0, 1), (0, 2), (0, 3), (0, 4)
71        , (1, 0), (1, 1), (1, 2), (1, 3), (1, 4)
72        , (2, 0), (2, 1), (2, 2), (2, 3), (2, 4)
73        , (3, 0), (3, 1), (3, 2), (3, 3), (3, 4)
74        , (4, 0), (4, 1), (4, 2), (4, 3), (4, 4)])

```

[Go to Answer](#)**Activity 1 (c) List Pairs in Fair Order**

- Rewrite the function which takes a pair of positive integers and outputs a list of all possible pairs in those ranges
- The output should treat each argument *fairly* — any initial prefix should have roughly the same number of instances of each argument
- Here is an example output

```

81 fairLstTest \
82   = (fairListing(5,5)
83     == [(0, 0)
84         , (0, 1), (1, 0)
85         , (0, 2), (1, 1), (2, 0)
86         , (0, 3), (1, 2), (2, 1), (3, 0)
87         , (0, 4), (1, 3), (2, 2), (3, 1), (4, 0)])

```

[Go to Answer](#)

Activity 1 (c) List Pairs in Fair Order

- Rewrite the function which takes a pair of positive integers and outputs a list of lists of all possible pairs in those ranges
- The output should treat each argument *fairly* — any initial prefix should have roughly the same number of instances of each argument — further, the output should be segment by each initial prefix (see example below)
- Here is an example output

```

94 fairLstATest \
95   = (fairListingA(5,5)
96     == [[(0, 0)]
97         , [(0, 1), (1, 0)]
98         , [(0, 2), (1, 1), (2, 0)]
99         , [(0, 3), (1, 2), (2, 1), (3, 0)]
100        , [(0, 4), (1, 3), (2, 2), (3, 1), (4, 0)]]

```

[Go to Answer](#)

6.4.1 Complexity of List Comprehensions

- Note that list comprehensions are not in M269
- See [Complexity of a List Comprehension](#)

```
[f(e) for e in row for row in mat]
```

- Suppose $f = \Theta(g)$ with n elements in a row and m rows
- Then complexity is $\Theta(g(e)) \times \Theta(n) \times \Theta(m) = \Theta(m \times n \times g(e))$

```
[[e**2 for e in row] for row in mat]
```

- $\Theta(e * 2) = \Theta(1)$
- Suppose n is maximum length of a row and m rows
- Then complexity is $\Theta(1) \times \Theta(n) \times \Theta(m) = \Theta(n \times m)$

[ToC](#)

Answer 1 (a) Stop Words Filter

- Answer 1 (a) Stop Words Filter

- Write here:

Answer 1 (a) Stop Words Filter

- Answer 1 (a) Stop Words Filter

```

24 def filterStopWords(words) :
25     nonStopWords \
26     = [word for word in words
27         if word not in stopWords]
28     return nonStopWords

31 filterStopWordsTest \
32 = filterStopWords(words) \
33 == ['quick', 'brown', 'fox'
34     , 'jumps', 'over', 'lazy', 'dog']

```

[Go to Activity](#)

Answer 1 (b) Transpose Matrix

- Answer 1 (b) Transpose Matrix
- Write here:

Answer 1 (b) Transpose Matrix

- Answer 1 (b) Transpose Matrix

```

49 def transMat(mat) :
50     rowLen = len(mat[0])
51     matTr \
52     = [[row[i] for row in mat] for i in range(rowLen)]
53     return matTr

55 transMatTestA \
56 = (transMat(matrixA)
57    == matATr)

```

- Note that a list comprehension is a valid expression as a target expression in a list comprehension
- The code assumes every row is of the same length

[Go to Activity](#)

Answer 1 (b) Transpose Matrix

- Note the differences in the list comprehensions below

```

38 matrixA \
39 = [[1, 2, 3, 4]
40     , [5, 6, 7, 8]
41     , [9, 10, 11, 12]]

```

```

Python3>>> [[row[i] for row in matrixA]
...           for i in range(4)]
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
Python3>>> [row[i] for row in matrixA
...           for i in range(4)]
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]
Python3>>> [row[i] for i in range(4)
...           for row in matrixA]
[1, 5, 9, 2, 6, 10, 3, 7, 11, 4, 8, 12]
Python3>>> [[row[i] for i in range(4)]
...           for row in matrixA]
[[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]]

```

[Go to Activity](#)**Answer 1 (b) Transpose Matrix**

- Answer 1 (b) Transpose Matrix
- The Python [NumPy](#) package provides functions for N-dimensional array objects
- For transpose see [numpy.ndarray.transpose](#)

```

Python3>>> import numpy as np
Python3>>> ar = np.array([[1,2],[3,4]])
Python3>>> ar
array([[1, 2],
       [3, 4]])
Python3>>> arT = ar.transpose()
Python3>>> arT
array([[1, 3],
       [2, 4]])
Python3>>> ar
array([[1, 2],
       [3, 4]])
Python3>>> ar.shape
(2, 2)

```

[Go to Activity](#)**Answer 1 (c) List Pairs in Fair Order**

- Answer 1 (c) List Pairs in Fair Order — first version
- Write here

```

69 yBiasLstTest \
70   = (yBiasListing(5,5)
71       == [(0, 0), (0, 1), (0, 2), (0, 3), (0, 4)
72           , (1, 0), (1, 1), (1, 2), (1, 3), (1, 4)
73           , (2, 0), (2, 1), (2, 2), (2, 3), (2, 4)
74           , (3, 0), (3, 1), (3, 2), (3, 3), (3, 4)
75           , (4, 0), (4, 1), (4, 2), (4, 3), (4, 4)])

```

[Go to Activity](#)**Answer 1 (c) List Pairs in Fair Order**

- Answer 1 (c) List Pairs in Fair Order
- This is the *obvious* but biased version

```

63 def yBiasListing(xRng,yRng) :
64     yBiasLst \
65     = [(x,y) for x in range(xRng)
66         for y in range(yRng)]
67     return yBiasLst
69 yBiasLstTest \
70   = (yBiasListing(5,5)
71       == [(0, 0), (0, 1), (0, 2), (0, 3), (0, 4)
72           , (1, 0), (1, 1), (1, 2), (1, 3), (1, 4)
73           , (2, 0), (2, 1), (2, 2), (2, 3), (2, 4)
74           , (3, 0), (3, 1), (3, 2), (3, 3), (3, 4)
75           , (4, 0), (4, 1), (4, 2), (4, 3), (4, 4)])

```

[Go to Activity](#)**Answer 1 (c) List Pairs in Fair Order**

- Answer 1 (c) List Pairs in Fair Order — second version
- Write here

```

83 fairLstTest \
84   = (fairListing(5,5)
85     == [(0, 0)
86         , (0, 1), (1, 0)
87         , (0, 2), (1, 1), (2, 0)
88         , (0, 3), (1, 2), (2, 1), (3, 0)
89         , (0, 4), (1, 3), (2, 2), (3, 1), (4, 0)])

```

[Go to Activity](#)**Answer 1 (c) List Pairs in Fair Order**

- Answer 1 (c) List Pairs in Fair Order — second version
- This works by making the sum of the coordinates the same for each prefix

```

77 def fairListing(xRng,yRng) :
78     fairLst \
79     = [(x,d-x) for d in range(yRng)
80         for x in range(d+1)]
81     return fairLst
82
83 fairLstTest \
84   = (fairListing(5,5)
85     == [(0, 0)
86         , (0, 1), (1, 0)
87         , (0, 2), (1, 1), (2, 0)
88         , (0, 3), (1, 2), (2, 1), (3, 0)
89         , (0, 4), (1, 3), (2, 2), (3, 1), (4, 0)])

```

[Go to Activity](#)**Answer 1 (c) List Pairs in Fair Order**

- Answer 1 (c) List Pairs in Fair Order — third version
- Write here

```

97 fairLstATest \
98   = (fairListingA(5,5)
99     == [[(0, 0)]
100         , [(0, 1), (1, 0)]
101         , [(0, 2), (1, 1), (2, 0)]
102         , [(0, 3), (1, 2), (2, 1), (3, 0)]
103         , [(0, 4), (1, 3), (2, 2), (3, 1), (4, 0)]])

```

[Go to Activity](#)**Answer 1 (c) List Pairs in Fair Order**

- Answer 1 (c) List Pairs in Fair Order — third version
- The *inner loop* is placed into its own list comprehension

```

91 def fairListingA(xRng,yRng) :
92     fairLstA \
93     = [[(x,d-x) for x in range(d+1)]
94         for d in range(yRng)]
95     return fairLstA
96
97 fairLstATest \
98   = (fairListingA(5,5)
99     == [[(0, 0)]
100         , [(0, 1), (1, 0)]
101         , [(0, 2), (1, 1), (2, 0)]
102         , [(0, 3), (1, 2), (2, 1), (3, 0)]
103         , [(0, 4), (1, 3), (2, 2), (3, 1), (4, 0)]])

```

[Go to Activity](#)

6.5 Master Theorem for Divide-and-Conquer Recurrences

- **The Divide-and-Conquer Method**

Many useful algorithms are recursive in structure and often follow a *divide-and-conquer method*

They break the problem into several subproblems similar to the original problem

- The time analysis is represented by a *recurrence system*

- **References**

- [Big O notation](#)
- [Master theorem](#)
- [Cormen et al. \(2022, chp 4\) Algorithms](#)
- These notes are partly based on *M261 Mathematics in Computing* and *M263 Building Blocks of Software* and are *not* part of *M269 Algorithms, Data Structures and Computability*

- **Recurrence System**

$$T(1) = b \quad (1)$$

$$T(n) = bn^\beta + cT\left(\frac{n}{d}\right) \quad \{n = d^\alpha > 1\} \quad (2)$$

- **Typical Expansion**

n	T(n)
d^0	b
d^1	$bn^\beta + cb$
d^2	$bn^\beta + cb\left(\frac{n}{d}\right)^\beta + c^2b$

- **General Expansion**

$$\begin{aligned}
 T(n) &= bn^\beta + cT\left(\frac{n}{d}\right) \\
 &= bn^\beta + cb\left(\frac{n}{d}\right)^\beta + c^2T\left(\frac{n}{d^2}\right) \\
 &= bn^\beta \left(1 + \frac{c}{d^\beta} + \left(\frac{c}{d^\beta}\right)^2 + \cdots + \left(\frac{c}{d^\beta}\right)^\alpha\right) \\
 T(n) &= bn^\beta \sum_{i=0}^{\log_d n} \left(\frac{c}{d^\beta}\right)^i \quad (3)
 \end{aligned}$$

- **Proof of Closed Form Equation (3)**

- For $n = 1$ equation (3) gives

$$T(1) = b1^\beta \sum_{i=0}^0 \left(\frac{c}{d^\beta}\right)^i = b \text{ which is correct (same as (1))}$$

- Assume equation (3) holds for $n = d^\alpha$. Then for $n = d^{\alpha+1}$

$$T(d^{\alpha+1}) = cT(d^\alpha) + bn^\beta \quad \text{by equation (2)}$$

$$= cbd^{\alpha\beta} \sum_{i=0}^{\alpha} \left(\frac{c}{d^\beta}\right)^i + bd^{(\alpha+1)\beta} \quad \text{by assumption}$$

$$= \left(\frac{c}{d^\beta}\right) bd^{(\alpha+1)\beta} \sum_{i=0}^{\alpha} \left(\frac{c}{d^\beta}\right)^i + bd^{(\alpha+1)\beta}$$

$$= bd^{(\alpha+1)\beta} \left(\sum_{i=1}^{\alpha+1} \left(\frac{c}{d^\beta}\right)^i + 1 \right) \quad \text{by rearrangement}$$

$$= bd^{(\alpha+1)\beta} \sum_{i=0}^{\alpha+1} \left(\frac{c}{d^\beta}\right)^i \quad \text{by rearrangement}$$

- Hence equation (3) holds for all $n = d^\alpha$ where $\alpha \in \mathbb{N}$

- If $c < d^\beta$ then the sum converges and $T(n)$ is $\Theta(n^\beta)$
- If $c = d^\beta$ then each term in the sum is 1 and $T(n)$ is $\Theta(n^\beta \log_d n)$

- If $c > d^\beta$ then use $\sum_{i=0}^p x^i = \frac{x^{p+1} - 1}{x - 1}$

$$\begin{aligned} T(n) &= bn^\beta \left[\frac{\left(\frac{c}{d^\beta}\right)^{\log_d n+1} - 1}{\left(\frac{c}{d^\beta}\right) - 1} \right] \\ &= \Theta\left(n^\beta \left(\frac{c}{d^\beta}\right)^{\log_d n}\right) \\ &= \Theta\left(c^{\log_d n}\right) \\ &= \Theta\left(n^{\log_d c}\right) \text{ since } d^{\log_b x} = x^{\log_b d} \end{aligned}$$

6.5.1 Master Theorem Example Usage

- Algorithm**
- Find mid point and check
if not equal to target, recurse on half the data

- Timing equations**

$$T(1) \leq 1$$

$$T(n) = T\left(\frac{n}{2}\right) + 1$$

- Hence $c = 1$, $d = 2$, $\beta = 0 \rightarrow$ case (2)

$$T(n) = \Theta(\log_2 n)$$

- Algorithm**

- Best case: splitting on **median** of data
- Recursively sort each half
- **Timing equations**

$$T(1) \leq k$$

$$T(n) = 2T\left(\frac{n}{2}\right) + kn$$

- Hence $c = 2$, $d = 2$, $\beta = 1 \rightarrow$ case (2)

$$T(n) = \Theta(n \log_2 n)$$

- See [Averages/Median](#)

- **Matrix Multiplication**

- Let A, B be two square matrices over a **ring**, $\mathcal{R}$
- Informally, a *ring* is a set with two binary operations which look similar to addition and multiplication of integers
- The problem is to implement **matrix multiplication** to find the matrix product $C = AB$
- Without loss of generality, we may assume that A , and B have sizes which are powers of 2 — if A , and B were not of this size, they could be padded with rows or columns of zeroes
- The Strassen algorithm partitions A, B and C into equally sized blocks

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \quad B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} \quad C = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}$$

with $A_{ij}, B_{ij}, C_{ij} \in \text{Mat}_{2^{n-1} \times 2^{n-1}}(\mathcal{R})$

- The usual (naive, standard) algorithm gives

$$\begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = \begin{pmatrix} A_{11} \times B_{11} + A_{12} \times B_{21} & A_{11} \times B_{12} + A_{12} \times B_{22} \\ A_{21} \times B_{11} + A_{22} \times B_{21} & A_{21} \times B_{12} + A_{22} \times B_{22} \end{pmatrix}$$

- This as 8 multiplications and if we assume multiplication is more expensive than addition then the time complexity is $\Theta(n^3)$
- The Strassen algorithm rearranges the calculation

$$M_1 = (A_{11} + A_{22}) \times (B_{11} + B_{22})$$

$$M_2 = (A_{21} + A_{22}) \times B_{11}$$

$$M_3 = A_{11} \times (B_{12} - B_{22})$$

$$M_4 = A_{22} \times (B_{21} - B_{11})$$

$$M_5 = (A_{11} + A_{12}) \times B_{22}$$

$$M_6 = (A_{21} - A_{11}) \times (B_{11} + B_{12})$$

$$M_7 = (A_{12} - A_{22}) \times (B_{21} + B_{22})$$

- We now express the C_{ij} in terms of the M_k

$$\begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = \begin{pmatrix} M_1 + M_4 - M_5 + M_7 & M_3 + M_5 \\ M_2 + M_4 & M_1 - M_2 + M_3 + M_6 \end{pmatrix}$$

- **Strassen Matrix Multiplication Timing Equations**

$$T(n) = 7T\left(\frac{n}{2}\right) + \frac{18}{4}n^2$$

$$T(1) \leq \frac{18}{4}$$

- This is derived from the 7 multiplications and 18 additions or subtractions

- $c = 7, d = 2, \beta = 2 \rightarrow$ case (3)

$$T(n) = \Theta\left(n^{\log_2 7}\right) = \Theta\left(n^{2.8}\right)$$

[ToC](#)

7 Exponentials and Logarithms

7.1 Exponentials and Logarithms — Definitions

- **Exponential function** $y = a^x$ or $f(x) = a^x$
- $a^n = a \times a \times \dots \times a$ (n a terms)
- **Logarithm** reverses the operation of exponentiation
- $\log_a y = x$ means $a^x = y$
- $\log_a 1 = 0$
- $\log_a a = 1$
- Method of logarithms propounded by John Napier from 1614
- **Log Tables** from 1617 by Henry Briggs
- **Slide Rule** from about 1620–1630 by William Oughtred of Cambridge
- *Logarithm* from Greek *logos* ratio, and *arithmos* number [Chambers \(2014\)](#) *Chambers Dictionary*

[ToC](#)

7.2 Rules of Indices

1. $a^m \times a^n = a^{m+n}$
2. $a^m \div a^n = a^{m-n}$
3. $a^{-m} = \frac{1}{a^m}$
4. $a^{\frac{1}{m}} = \sqrt[m]{a}$
5. $(a^m)^n = a^{mn}$

6. $a^{\frac{n}{m}} = \sqrt[m]{a^n}$

7. $a^0 = 1$ where $a \neq 0$

- **Exercise** Justify the above rules
- What should 0^0 evaluate to ?
- See [Wikipedia: Exponentiation](#)
- The *justification* above probably only worked for whole or *rational* numbers — see later for exponents with *real* numbers (and the value of *logarithms*, *calculus*...)

ToC

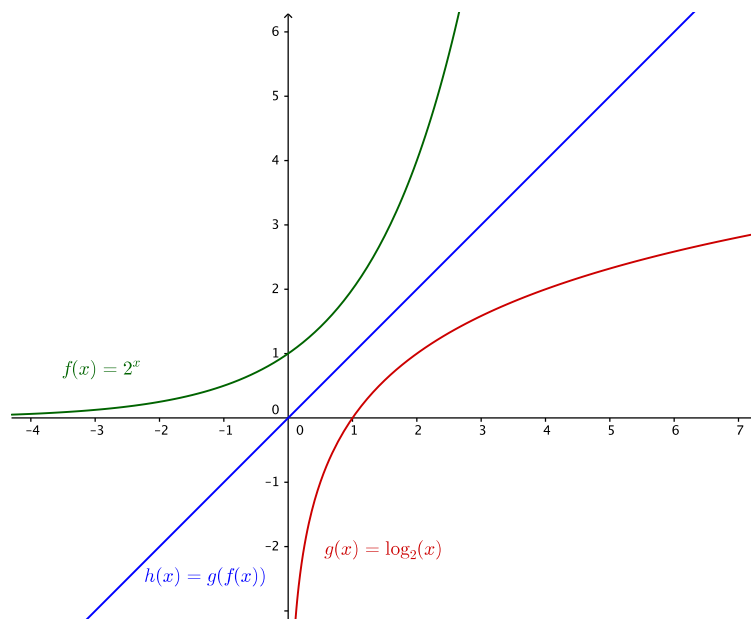
7.3 Logarithms — Motivation

- Make arithmetic easier — turns multiplication and division into addition and subtraction (see later)
- Complete the range of [elementary functions](#) for differentiation and integration
- An [elementary function](#) is a function of one variable which is the composition of a finite number of arithmetic operations ((+), (-), (×), (÷)), exponentials, logarithms, constants, and solutions of algebraic equations (a generalization of nth roots).
- The elementary functions include the trigonometric and hyperbolic functions and their inverses, as they are expressible with complex exponentials and logarithms.
- See A Level FP2 for Euler's relation $e^{i\theta} = \cos \theta + i \sin \theta$
- In A Level C3, C4 we get $\int \frac{1}{x} = \log_e |x| + C$
- e is [Euler's number](#) 2.71828...

ToC

7.4 Exponentials and Logarithms — Graphs

- See GeoGebra file [expLog.ggb](#)


[ToC](#)

7.5 Laws of Logarithms

- **Multiplication law** $\log_a xy = \log_a x + \log_a y$
- **Division law** $\log_a \left(\frac{x}{y}\right) = \log_a x - \log_a y$
- **Power law** $\log_a x^k = k \log_a x$
- **Proof of Multiplication Law**

$$x = a^{\log_a x}$$

$$y = a^{\log_a y}$$

$$xy = a^{\log_a x} \times a^{\log_a y}$$

$$= a^{\log_a x + \log_a y}$$

$$\text{Hence } \log_a xy = \log_a x + \log_a y$$

by definition of log

by laws of indices
by definition of log

[ToC](#)

7.6 Arithmetic and Inverses

- *Notation helps or maybe not ?*
- **Addition** $\text{add}(b, x) = x + b$
- **Subtraction** $\text{sub}(b, x) = x - b$
- Inverse $\text{sub}(b, \text{add}(b, x)) = (x + b) - b = x$
- **Multiplication** $\text{mul}(b, x) = x \times b$
- **Division** $\text{div}(b, x) = x \div b = \frac{x}{b} = x/b$

- Inverse $\text{div}(b, \text{mul}(b, x)) = (x \times b) \div b = \frac{(x \times b)}{b} = x$
- **Exponentiation** $\text{exp}(b, x) = b^x$
- **Logarithm** $\log(b, x) = \log_b x$
- Inverse $\log(b, \text{exp}(b, x)) = \log_b(b^x) = x$
- *What properties do the operations have that work (or not) with the notation ?*

Arithmetic Operations — Commutativity and Associativity

- **Commutativity** $x \otimes y = y \otimes x$
- **Associativity** $(x \otimes y) \otimes z = x \otimes (y \otimes z)$
- (+) and (×) are *semantically* commutative and associative — so we can leave the brackets out
- (−) and (÷) are not
- Evaluate $(3 - (2 - 1))$ and $((3 - 2) - 1)$
- Evaluate $(3/(2/2))$ and $((3/2)/2)$
- We have the *syntactic* ideas of left (and right) associativity
- We choose (−) and (÷) to be left associative
- $3 - 2 - 1$ means $((3 - 2) - 1)$
- $3/2/2$ means $((3/2)/2)$
- **Operator precedence** is also a choice (remember BIDMAS or BODMAS ?)
- If in doubt, put the brackets in

Exponentials and Logarithm — Associativity

- What should 2^{3^4} mean ?
- Let $b \wedge x \equiv b^x$
- Evaluate $(2 \wedge 3) \wedge 4$ and $2 \wedge (3 \wedge 4)$
- Evaluate $c = \log_b(\log_b((b \wedge b) \wedge x))$
- Evaluate $d = \log_b(\log_b(b \wedge (b \wedge x)))$
- Beware spreadsheets Excel and LibreOffice here
- $(2^3)^4 = 2^{12}$ and $2^{3^4} = 2^{81}$
- Exponentiation is not semantically associative
- We *choose* the syntactic left or right associativity to make the syntax nicer.
- Evaluate $c = \log_b(\log_b((b \wedge b) \wedge x))$
- $c = \log_b(x \log_b(b^b)) = \log_b(x \cdot (b \log_b b)) = \log_b(x \cdot b \cdot 1)$
- Hence $c = \log_b x + \log_b b = \log_b x + 1$
- Not symmetrical (unless b and x are both 2)

- Evaluate $d = \log_b(\log_b(b \wedge (b \wedge x)))$
- $d = \log_b((b \wedge x)(\log_b b)) = \log_b((b \wedge x) \times 1)$
- Hence $d = \log_b(b \wedge x) = x(\log_b b) = x$
- Which is what we want — so exponentiation is *chosen* to be right associative

[ToC](#)

7.7 Change of Base

- Change of base

$$\log_a x = \frac{\log_b x}{\log_b a}$$

Proof: Let $y = \log_a x$

$$a^y = x$$

$$\log_b a^y = \log_b x$$

$$y \log_b a = \log_b x$$

$$y = \frac{\log_b x}{\log_b a}$$

- Given x , $\log_b x$, find the base b

$$- b = x^{\frac{1}{\log_b x}}$$

- $\log_a b = \frac{1}{\log_b a}$

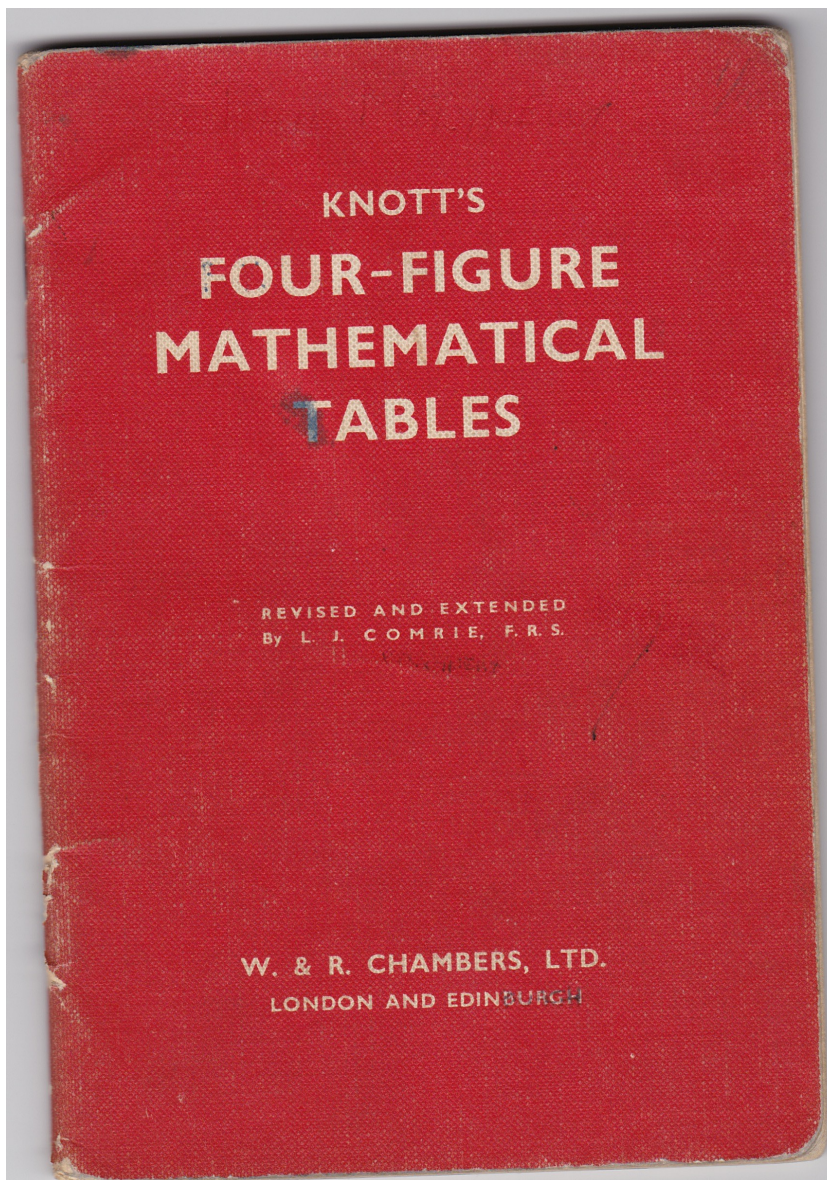
[ToC](#)

8 Before Calculators and Computers

- We had computers before 1950 — they were *humans* with pencil, paper and some further aids:
- **Slide rule** invented by [William Oughtred](#) in the 1620s — major calculating tool until [pocket calculators](#) in 1970s
- **Log tables** in use from early 1600s — method of logarithms propounded by [John Napier](#)
- **Logarithm** from Greek *logos* ratio, and *arithmos* number

8.1 Log Tables

Knott's Four-Figure Mathematical Tables



Logarithms of Numbers

2

LOGARITHMS OF NUMBERS

x	0 1 2 3 4 5 6 7 8 9										Δ	ADD									
												1 2 3 4 5 6 7 8 9									
10	-0000	0643	0086	0128	0170	0212	0254	0296	0338	0374	42	4	8	13	17	21	25	29	34		
11	-0414	0453	0492	0531	0569	0607	0645	0682	0719	0755	40	4	8	12	16	19	23	27	31		
12	-0792	0828	0864	0899	0934	0969	1004	1038	1072	1106	38	4	7	11	15	18	22	26	30		
13	-1139	1173	1206	1239	1271	1303	1335	1367	1399	1430	36	4	7	10	14	17	20	24	27		
14	-1461	1492	1523	1553	1584	1614	1644	1673	1703	1732	30	3	6	9	12	15	18	21	24		
15	-1761	1790	1818	1847	1875	1903	1931	1959	1987	2014	28	3	6	8	11	14	17	20	22		
16	-2041	2068	2095	2122	2148	2175	2201	2227	2253	2279	26	3	5	8	10	13	16	19	21		
17	-2304	2330	2355	2380	2405	2430	2455	2480	2504	2529	25	3	5	7	10	12	15	17	20		
18	-2553	2577	2601	2625	2648	2672	2695	2718	2742	2765	23	3	5	7	9	12	14	16	19		
19	-2788	2810	2833	2856	2878	2900	2923	2945	2967	2989	22	3	4	7	9	11	13	15	18		
20	-3010	3032	3054	3075	3096	3118	3139	3160	3181	3201	21	2	4	6	8	10	12	14	17		
21	-3222	3243	3263	3284	3304	3324	3345	3365	3385	3404	20	2	4	6	8	10	12	14	16		
22	-3424	3444	3464	3483	3502	3522	3541	3560	3579	3598	19	2	4	6	8	10	11	13	15		
23	-3617	3636	3655	3674	3692	3711	3729	3747	3766	3784	18	2	4	6	7	9	11	13	15		
24	-3802	3820	3838	3856	3874	3892	3909	3927	3945	3962	18	2	4	5	7	9	11	13	14		
25	-3979	3997	4014	4031	4048	4065	4082	4099	4116	4133	17	2	3	5	7	9	10	12	14		
26	-4150	4166	4183	4200	4216	4232	4249	4265	4281	4298	16	2	3	5	6	8	10	11	13		
27	-4314	4330	4346	4362	4378	4393	4409	4425	4440	4456	16	2	3	5	6	8	9	11	12		
28	-4472	4487	4502	4518	4533	4548	4564	4579	4594	4609	15	2	3	5	6	8	9	10	11		
29	-4624	4639	4654	4669	4683	4698	4713	4728	4742	4757	14	1	3	4	5	7	9	10	11		
30	-4771	4786	4800	4814	4829	4843	4857	4871	4886	4900	14	1	3	4	5	6	8	9	10		
31	-4914	4928	4942	4955	4969	4983	4997	5011	5024	5038	13	1	3	4	5	6	7	8	9		
32	-5051	5065	5079	5092	5105	5119	5132	5145	5159	5172	13	1	3	4	5	6	7	8	9		
33	-5185	5198	5211	5224	5237	5250	5263	5276	5289	5302	13	1	3	4	5	6	7	8	9		
34	-5315	5328	5340	5353	5366	5378	5391	5403	5416	5428	13	1	3	4	5	6	7	8	9		
35	-5441	5453	5465	5478	5490	5502	5514	5527	5539	5551	12	1	2	4	5	6	7	8	10		
36	-5563	5575	5587	5599	5611	5623	5635	5647	5659	5670	12	1	2	4	5	6	7	8	9		
37	-5682	5694	5705	5717	5729	5740	5752	5763	5775	5786	12	1	2	3	5	6	7	8	9		
38	-5798	5809	5821	5832	5843	5855	5866	5877	5888	5899	11	1	2	3	5	6	7	8	9		
39	-5911	5922	5933	5944	5955	5966	5977	5988	5999	6010	11	1	2	3	4	6	7	8	9		
40	-6021	6031	6042	6053	6064	6075	6085	6096	6107	6117	11	1	2	3	4	5	6	7	8		
41	-6128	6138	6149	6160	6170	6180	6191	6201	6212	6222	11	1	2	3	4	5	6	7	8		
42	-6232	6243	6253	6263	6274	6284	6294	6304	6314	6325	10	1	2	3	4	5	6	7	8		
43	-6335	6345	6355	6365	6375	6385	6395	6405	6415	6425	10	1	2	3	4	5	6	7	8		
44	-6435	6444	6454	6464	6474	6484	6493	6503	6513	6522	10	1	2	3	4	5	6	7	8		
45	-6532	6542	6551	6561	6571	6580	6590	6599	6609	6618	9	1	2	3	4	5	6	7	8		
46	-6628	6637	6646	6655	6665	6675	6684	6693	6702	6712	9	1	2	3	4	5	6	7	8		
47	-6721	6730	6739	6749	6758	6767	6776	6785	6794	6803	9	1	2	3	4	5	6	7	8		
48	-6812	6821	6830	6839	6848	6857	6866	6875	6884	6893	9	1	2	3	4	5	6	7	8		
49	-6902	6911	6920	6929	6938	6947	6955	6964	6972	6981	9	1	2	3	4	5	6	7	8		

USEFUL CONSTANTS WITH THEIR LOGARITHMS

No.	Log.	No.	Log.	No.	Log.
π 3-14159	-0-4971	$\frac{1}{2}\pi$	1-7581	e 2-71828	-0-4343
$\frac{1}{e}$ 0-3183	-7-9043	1 radian, $\frac{1}{360}^\circ$	3-5363	$\frac{1}{2}$ 0-4343	-6-378
$\frac{1}{\pi}$ 0-3183	-0-5029	$206265''$	5-3144	2 1-3026	-0-3622
$\sqrt{e}$ 1-6486	0-2193	$\log 1^\circ$	0-01 453 293	$\frac{1}{M}$ 2-0419	
$\sqrt{e}$ 1-7725	-0-2486	$\log 1'$	0-000 290 888	$\frac{1}{M}$ 2-4637	
$\frac{1}{4}$ 4-1888	-0-6221	$\log 1''$	0-000 004 848	$\frac{1}{M}$ 6-8556	

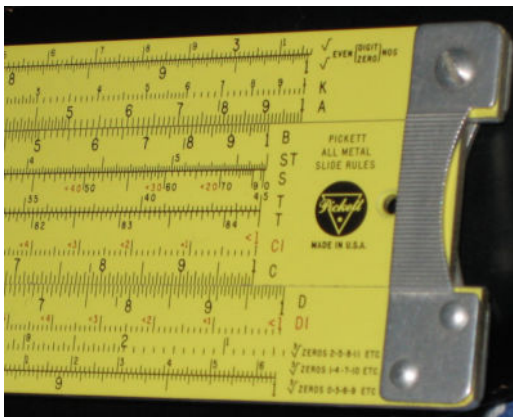
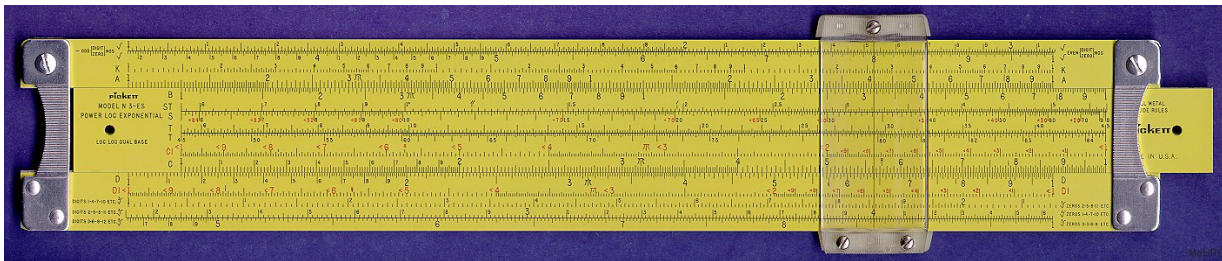
LOGARITHMS OF NUMBERS

3

x	0 1 2 3 4 5 6 7 8 9										Δ	ADD									
												1 2 3 4 5 6 7 8 9									
50	-6990	6998	7007	7016	7024	7033	7042	7050	7059	7067	8	1	2	3	4	5	6	7	8		
51	-7076	7084	7093	7101	7110	7118	7126	7135	7143	7152	8	1	2	3	4	5	6	7	8		
52	-7160	7168	7177	7185	7193	7202	7210	7218	7226	7235	8	1	2	3	4	5	6	7	8		
53	-7243	7251	7259	7267	7275	7284	7292	7300	7308	7316	8	1	2	3	4	5	6	7	8		
54	-7324	7332	7340	7348	7356	7364	7372	7380	7388	7396	8	1	2	3	4	5	6	7	8		
55	-7404	7412	7419	7427	7435	7443	7451	7459	7466	7474	8	1	2	3	4	5	6	7	8		
56	-7482	7490	7497	7505	7513	7520	7528	7536	7543	7551	8	1	2	3	4	5	6	7	8		
57	-7559	7566	7574	7582	7589	7597	7604	7612	7619	7627	8	1	2	3	4	5	6	7	8		
58	-7634	7642	7649	7657	7664	7672	7679	7686	7694	7701	8	1	2	3	4	5	6	7	8		
59	-7709	7716	7723	7731	7738	7745	7752	7760	7767	7774	8	1	2	3	4	5	6	7	8		
60	-7782	7789	7796	7803	7810	7818	7825	7832	7839	7846	7	1	1	2	3	4	4	5	6		
61	-7853	7860	7868	7875	7882	7889	7896	7903	7910	7917	7	1	1	2	3	4	4	5	6		
62	-7924	7931	7938	7945	7952	7959	7966	7973	7980	7987	7	1	1	2	3	4	4	5	6		
63	-7993	8000	8007	8014	8021	8028	8035	8041	8048	8055	7	1	1	2	3	4	4	5	6		
64	-8062	8069	8075	8082	8089	8096	8102	8109	8116	8122	7	1	1	2	3	4	4	5	6		
65	-8129	8136	8142	8149	8156	8162	8169	8176	8182	8189	7	1	1	2	3	4	4	5	6		
66	-8195	8202	8209	8215	8222	8228	8235	8241	8248	8254	7	1	1	2	3	4	4	5	6		
67	-8261	8267	8274	8280	8287	8293	8299	8306	8312	8319	7	1	1	2	3	4	4	5	6		
68	-8325	8331	8338	8344	8351	8357	8363	8370	8376	8382	7	1	1	2	3	4	4	5	6		
69	-8388	8395	8401	8407	8414	8420	8426	8432	8439	8445	7	1	1	2	3	4	4	5	6		
70	-8451	8457	8463	8470	8476	8482	8488	8494	8500	8506	6	1	1	2	2	3	4	4	5		
71	-8513	8519	8525	8531	8537	8543	8549	8555	8561	8567	6	1	1	2	2	3	4	4	5		
72	-8573	8579	8585	8591	8597	8603	8609	8615	8621	8627	6	1	1	2	2	3	4	4	5		
73	-8633	8639	8645	8651	8657	8663	8669	8675	8681	8686	6	1	1	2	2	3	4	4	5		
74	-8692	8698	8704	8710	8716	8722	8727	8733	8739	8745	6	1	1	2	2	3	4	4	5		
75	-8751	8756	8762	8768	8774	8779	8785	8791	8797	8802	6	1	1	2	2	3	4	4	5		
76	-8808	8814	8820	8825	8831	8837	8842	8848	8854	8859	6	1	1	2	2	3	4	4	5		
77	-8865	8871	8876	8882	8887	8893	8898	8904	8910	8915	6	1	1	2	2	3	4	4	5		
78	-8921	8927	8932	8938	8943	8949	8954	8960	8965	8971	6	1	1	2	2	3	4	4	5		
79	-8976	8982	8987	8993	8998	9004	9009	9015	9020	9025	6	1	1	2	2	3	4	4	5		
80	-9031	9036	9042																		

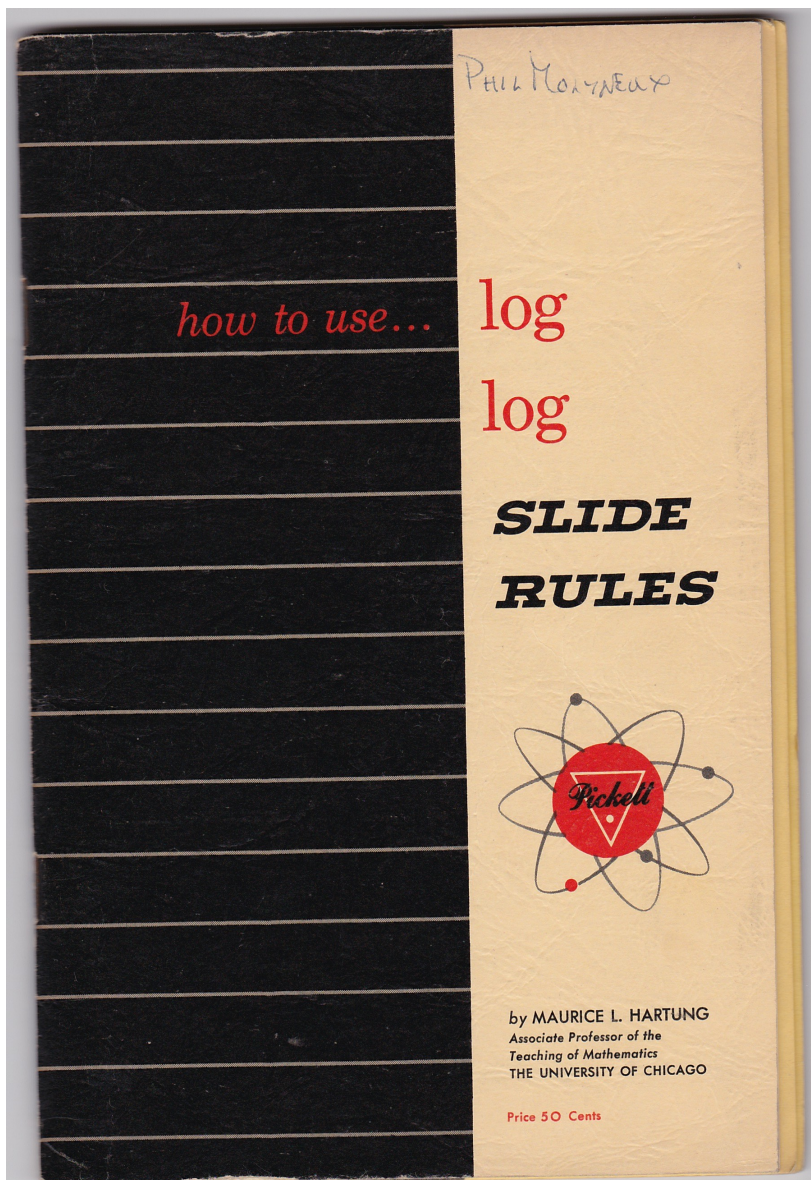
8.2 Slide Rules

Pickett N 3-ES from 1967



- See [Oughtred Society](#)
- [UKSRC](#)
- [Rod Lovett's Slide Rules](#)
- [Slide Rule Museum](#)

Pickett log log Slide Rules Manual 1953

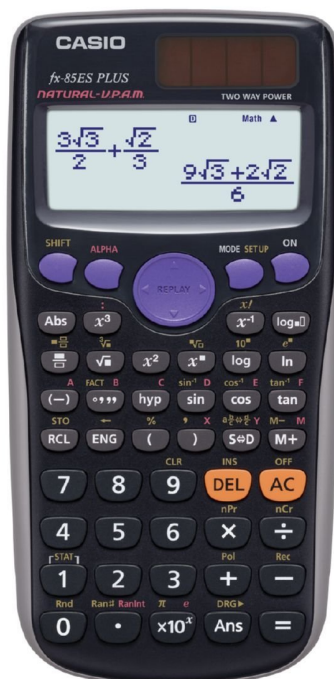
[ToC](#)

8.3 Calculators

HP HP-21 Calculator from 1975 £69



Casio fx-85GT PLUS Calculator from 2013 £10



Calculator links

- HP Calculator Museum <http://www.hpmuseum.org>

- **HP Calculator Emulators** <http://nonpareil.brouhaha.com>
- **HP Calculator Emulators for OS X** <http://www.bartosiak.org/nonpareil/>
- **Vintage Calculators Web Museum** <http://www.vintagecalculators.com>

[ToC](#)

8.4 Example Calculation

- Evaluate 89.7×597
- **Knott's Tables**
- $\log_{10} 89.7 = 1.9528$ and $\log_{10} 597 = 2.7760$
- Shows *mantissa* (decimal) & *characteristic* (integral)
- Add 4.7288, take *antilog* to get $5346 + 10 = 5.356 \times 10^4$
- **HP-21 Calculator** — set display to 4 decimal places
- $89.7 \log = 1.9528$ and $597 \log = 2.7760$
- $+$ displays 4.7288
- $10 \text{ ENTER}, x \Rightarrow y$ and y^x displays 53550.9000
- **Casio fx-85GT PLUS**
- $\log 89.7) = 1.952792443 + \log 597) = 2.775974331 =$
- $4.728766774 \text{ Ans} + 10^x$ gives 53550.9

[ToC](#)

9 Logic and Truth Tables

9.1 Boolean Expressions and Truth Tables

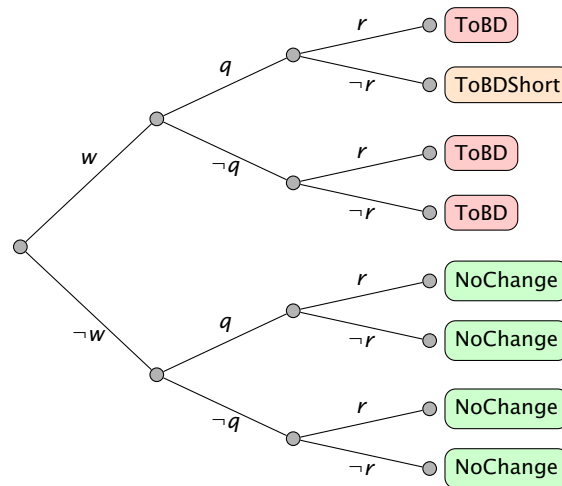
Traffic Lights Example

- Consider traffic light at the intersection of roads *AC* and *BD* with the following rules for the *AC* controller
- Vehicles should not wait on red on *BD* for too long.
- If there is a long queue on *AC* then *BD* is only given a green for a short interval.
- If both queues are long the usual flow times are used.
- We use the following propositions:
 - *w* Vehicles have been waiting on red on *BD* for too long
 - *q* Queue on *AC* is too long
 - *r* Queue on *BD* is too long
- Given the following events:
 - *ToBD* Change flow to *BD*

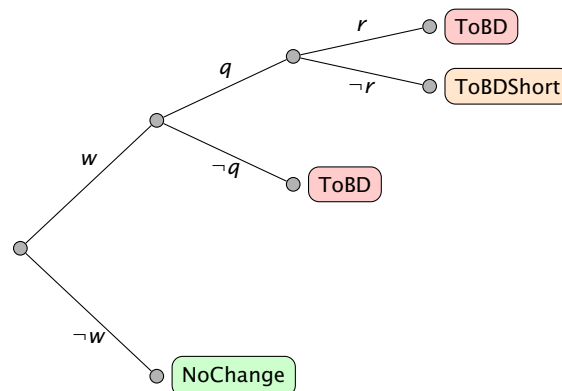
- **ToBDShort** Change flow to *BD* for short time
- **NoChange** No Change to lights
- Express above as truth table, outcome tree, boolean expression
- **Traffic Lights outcome table**

<i>w</i>	<i>q</i>	<i>r</i>	Event
T	T	T	ToBD
T	T	F	ToBDShort
T	F	T	ToBD
T	F	F	ToBD
F	T	T	NoChange
F	T	F	NoChange
F	F	T	NoChange
F	F	F	NoChange

- **Traffic lights outcome tree**



- **Traffic lights outcome tree simplified**



- **Traffic Lights code 01**
- See [M269TutorialProgPythonADT01.py](#)

```

3 def trafficLights01(w,q,r) :
4     """
5     Input 3 Booleans
6     Return Event string
7     """
8     if w :
```

```

9   if q :
10       if r :
11           evnt = "ToBD"
12       else :
13           evnt = "ToBDShort"
14       else :
15           evnt = "ToBD"
16   else :
17       evnt = "NoChange"
18   return evnt

```

• Traffic Lights test code 01

```

22 trafficLights01Evnts = [(w,q,r), trafficLights01(w,q,r))
23                         for w in [True,False]
24                         for q in [True,False]
25                         for r in [True,False]]
26
27 assert trafficLights01Evnts \
28 == [((True, True, True), 'ToBD')
29      ,((True, True, False), 'ToBDShort')
30      ,((True, False, True), 'ToBD')
31      ,((True, False, False), 'ToBD')
32      ,((False, True, True), 'NoChange')
33      ,((False, True, False), 'NoChange')
34      ,((False, False, True), 'NoChange')
35      ,((False, False, False), 'NoChange')]

```

• Traffic Lights code 02 compound Boolean conditions

```

37 def trafficLights02(w,q,r) :
38     """
39     Input 3 Booleans
40     Return Event string
41     """
42     if ((w and q and r) or (w and not q)) :
43         evnt = "ToBD"
44     elif (w and q and not r) :
45         evnt = "ToBDShort"
46     else :
47         evnt = "NoChange"
48     return evnt

```

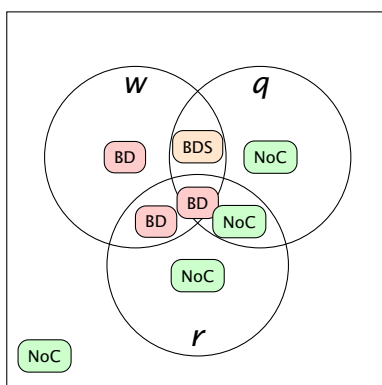
• What objectives do we have for our code ?

• Traffic Lights test code 02

```

52 trafficLights02Evnts = [(w,q,r), trafficLights02(w,q,r))
53                         for w in [True,False]
54                         for q in [True,False]
55                         for r in [True,False]]
56
57 assert trafficLights02Evnts == trafficLights01Evnts

```



• Traffic Lights Venn diagram

- OK using a fill colour would look better but didn't have the time to hack the package

[ToC](#)

9.2 Conditional Expressions and Validity

- **Validity** of Boolean expressions
- **Complete** every outcome returns an event (or error message, raises an exception)
- **Consistent** — we do not want two nested **if** statements or expressions resulting in different events
- We check this by ensuring that the events form a disjoint partition of the set of outcomes — see the Venn diagram
- We would quite like the programming language processor to warn us otherwise — not always possible

[ToC](#)

9.3 Boolean Expressions Exercise

Rail Ticket Exercise

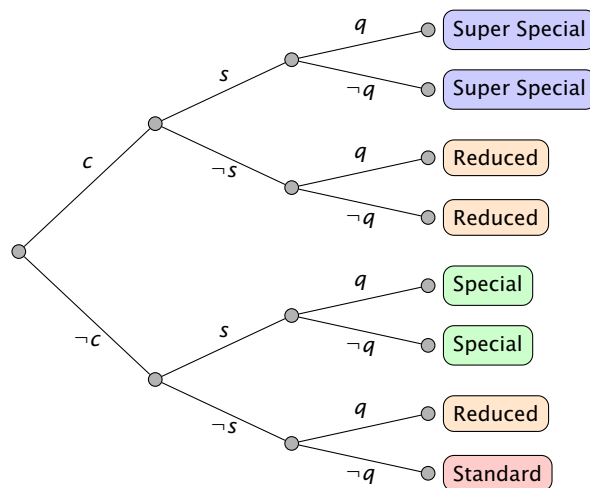
- Rail ticket discounts for:
 - c Rail card
 - q Off-peak time
 - s Special offer
- 4 fares: Standard, Reduced, Special, Super Special
- Rules:
 1. Reduced fare if rail card or at off-peak time
 2. Without rail card no reduction for both special offer and off-peak.
 3. Rail card always has reduced fare but cannot get off-peak discount as well.
 4. Rail card gets super special discount for journey with special offer
- Draw up truth table, outcome tree, Venn diagram and conditional statement (or expression) for this
- Rail ticket outcome table

c	q	s	Event
T	T	T	Super Special
T	T	F	Reduced
T	F	T	Super Special
T	F	F	Reduced
F	T	T	Special
F	T	F	Reduced
F	F	T	Special
F	F	F	Standard

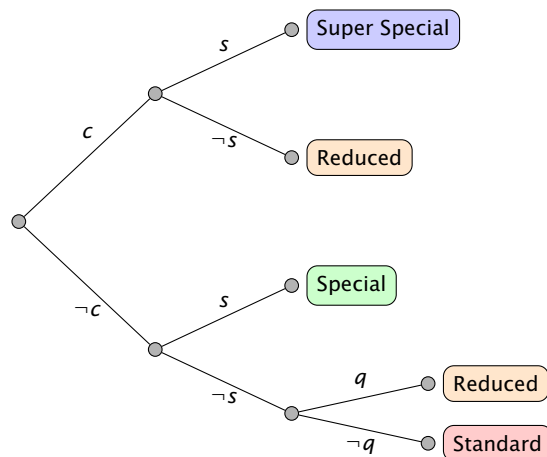
- Rail ticket outcome table
- Note that it may be more convenient to change columns

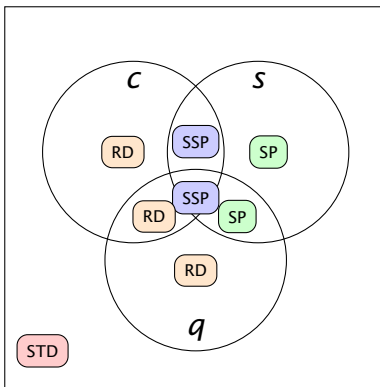
c	s	q	Event
T	T	T	Super Special
T	T	F	Super Special
T	F	T	Reduced
T	F	F	Reduced
F	T	T	Special
F	T	F	Special
F	F	T	Reduced
F	F	F	Standard

- Real fares are a little more complex — see brfares.com
- Rail Ticket outcome tree



- Rail Ticket outcome tree simplified





- Rail Ticket Venn diagram

- Rail Ticket code 01

```

61 def railTicket01(c,s,q) :
62     """
63     Input 3 Booleans
64     Return Event string
65     """
66     if c :
67         if s :
68             evnt = "SSP"
69         else :
70             evnt = "RD"
71     else :
72         if s :
73             evnt = "SP"
74         else :
75             if q :
76                 evnt = "RD"
77             else :
78                 evnt = "STD"
79     return evnt

```

- Rail Ticket test code 01

```

83 railTicket01Evnts = [((c,s,q), railTicket01(c,s,q))
84                       for c in [True,False]
85                       for s in [True,False]
86                       for q in [True,False]]
87
88 assert railTicket01Evnts \
89 == [((True, True, True), 'SSP')
90      ,((True, True, False), 'SSP')
91      ,((True, False, True), 'RD')
92      ,((True, False, False), 'RD')
93      ,((False, True, True), 'SP')
94      ,((False, True, False), 'SP')
95      ,((False, False, True), 'RD')
96      ,((False, False, False), 'STD')]

```

- Rail Ticket code 02 compound Boolean expressions

```

98 def railTicket02(c,s,q) :
99     """
100     Input 3 Booleans
101     Return Event string
102     """
103     if (c and s) :
104         evnt = "SSP"
105     elif ((c and not s) or (not c and not s and q)) :
106         evnt = "RD"
107     elif (not c and s) :
108         evnt = "SP"
109     else :
110         evnt = "STD"
111     return evnt

```

• Rail Ticket test code 02

```

115 railTicket02Evnts = [((c,s,q), railTicket02(c,s,q))
116                       for c in [True,False]
117                       for s in [True,False]
118                       for q in [True,False]]
120 assert railTicket02Evnts == railTicket01Evnts

```

[ToC](#)

9.4 Propositional Calculus

- Unit 2 section 3.2 *A taste of formal logic* introduces [Propositional calculus](#)
- A language for calculating about Booleans — *truth values*
- Gives operators (connectives) [conjunction \(\$\wedge\$ \) AND](#), [disjunction \(\$\vee\$ \) OR](#), [negation \(\$\neg\$ \) NOT](#), [implication \(\$\Rightarrow\$ \) IF](#)
- There are 16 possible functions $(\mathbb{B}, \mathbb{B}) \rightarrow \mathbb{B}$ — see below — defined by their truth tables
- **Discussion** Did you find the truth table for implication weird or surprising?
- **Implication** has a **negative** definition — we accept its truth unless we have contrary evidence
- $T \Rightarrow T == T$ and $T \Rightarrow F == F$
- Hence 4 possibilities for truth table

		$\uparrow$		$\leftrightarrow$	
p	q	p	q	p	q
T	T	T	T	T	T
T	F	F	F	F	F
F	T	T	T	F	F
F	F	T	F	T	F

- $(\Rightarrow)$ must have the entry shown — the others are taken
- Do not think of p *causing* q
- **Functionally complete** set of connectives is one which can be used to express all possible connectives
- $p \Rightarrow q \equiv \neg p \vee q$ so we could just use $\{\neg, \wedge, \vee\}$
- **Boolean programming** — we have to have a functionally complete set but choose more to make the programming easier
- *Expressiveness* is an issue in programming language design
- **NAND** $p \overline{\wedge} q$, $p \uparrow q$, Sheffer stroke
- **NOR** $p \overline{\vee} q$, $p \downarrow q$, Pierce's arrow
- See truth tables below — both $\{\uparrow\}, \{\downarrow\}$ are functionally complete
- **Exercise** verify

- $\neg p \equiv p \uparrow p$
- $p \wedge q \equiv \neg(p \uparrow q) = (p \uparrow q) \uparrow (p \uparrow q)$
- $p \vee q \equiv (p \uparrow p) \uparrow (q \uparrow q)$
- $\neg p \equiv p \downarrow p$
- $p \wedge q \equiv (p \downarrow p) \downarrow (q \downarrow q)$
- $p \vee q \equiv \neg(p \downarrow q) = (p \downarrow q) \downarrow (p \downarrow q)$

- Not a novelty — the [Apollo Guidance Computer](#) was implemented in [NOR](#) gates alone.

[ToC](#)

9.5 Truth Function

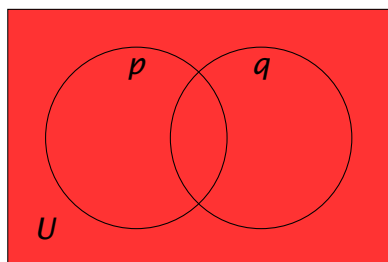
- The following appendix notes illustrate the 16 binary functions of two Boolean variables
- See [Truth function](#)
- See [Functional completeness](#)
- See [Sheffer stroke](#)
- See [Logical NOR](#)

Table of Binary Truth Functions

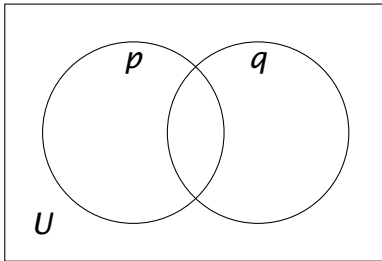
p	q	$\top$	$p \vee q$	$p \Leftrightarrow q$	p	$p \Uparrow q$	q	$p \Leftrightarrow q$	$p \wedge q$
T	T	T	T	T	T	T	T	T	T
T	F	T	T	F	T	F	F	F	F
F	T	T	T	F	F	T	T	F	F
F	F	T	F	T	F	F	F	T	F

p	q	$\perp$	$p \nabla q$	$p \nLeftrightarrow q$	$\neg p$	$p \nUparrow q$	$\neg q$	$p \nLeftrightarrow q$	$p \nabla q$
T	T	F	F	F	F	F	F	F	F
T	F	F	F	F	F	T	T	T	T
F	T	F	F	T	T	F	F	T	T
F	F	F	T	F	T	T	T	F	T

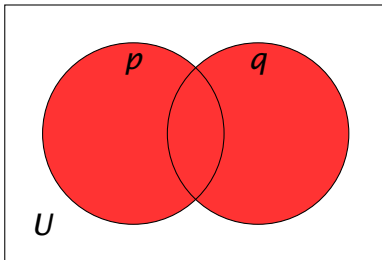
- **Tautology** True, $\top$, *Top*



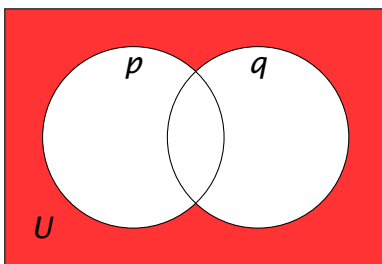
- **Contradiction** False, $\perp$, *Bottom*



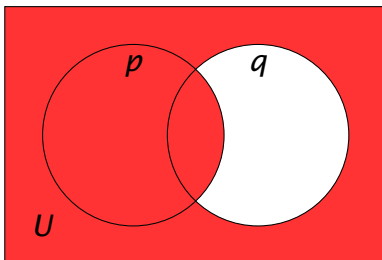
- **Disjunction** OR, $p \vee q$



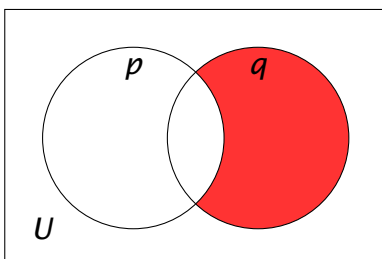
- **Joint Denial** NOR, $p \overline{\vee} q$, $p \downarrow q$, *Pierce's arrow*



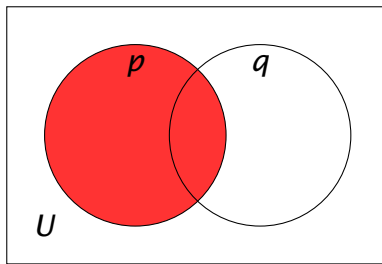
- **Converse Implication** $p \Leftarrow q$



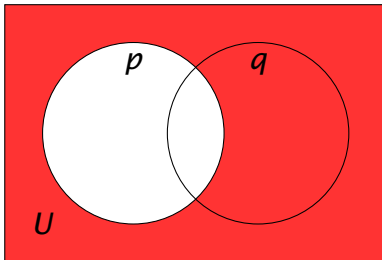
- **Converse Nonimplication** $p \nLeftarrow q$



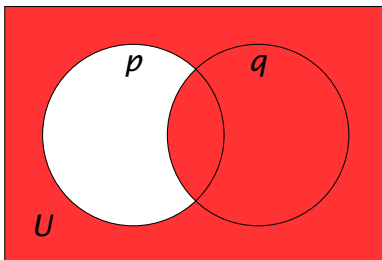
- **Proposition** p



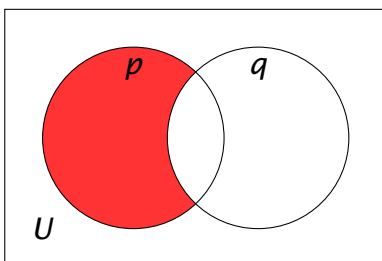
- Negation of p



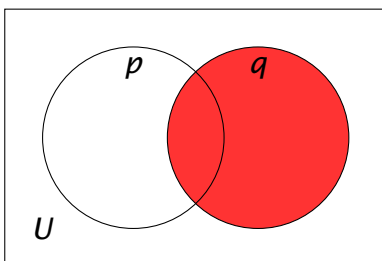
- Material Implication $p \Rightarrow q$



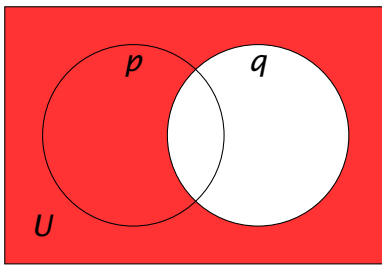
- Material Nonimplication $p \nRightarrow q$



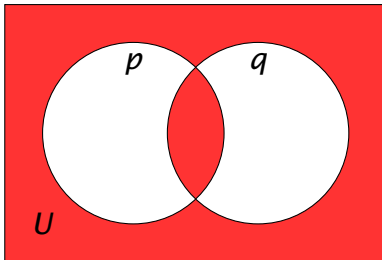
- Proposition $q \setminus p$



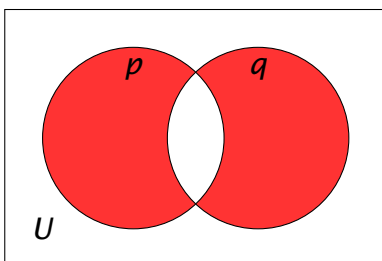
- Negation of q $\neg q$



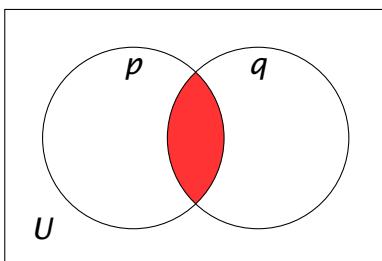
- **Biconditional** If and only if, IFF, $p \Leftrightarrow q$



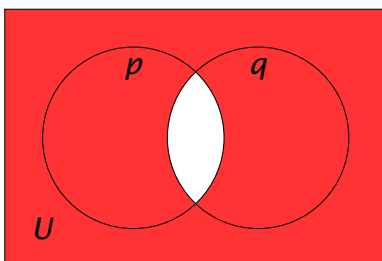
- **Exclusive disjunction** XOR, $p \oplus q$



- **Conjunction** AND, $p \wedge q$



- **Alternative denial** NAND, $p \bar{\wedge} q$, $p \uparrow q$, *Sheffer stroke*



10 Abstract Data Types

10.1 Abstract Data Types — Overview

- **Abstract data type** is a type with associated operations, but whose representation

is hidden (or not accessible)

- Common examples in most programming languages are Integer and Characters and other built in types such as tuples and lists
- [Abstract data types](#) are modeled on [Algebraic structures](#)
 - A set of values
 - Collection of operations on the values
 - Axioms for the operations may be specified as equations or pre and post conditions
- **Health Warning** different languages provide different ways of doing data abstraction with similar names that may mean subtly different things
- **Abstract Data Types and Object-Oriented Programming**
- Example: [Shape](#) with [Circles](#), [Squares](#), ... and operations [draw](#), [moveTo](#), ...
- ADT approach centres on the data type — that tells you what shapes exist
- For each operation on shapes, you describe what they do for different shapes.
- OO you declare that to be a shape, you have to have some operations ([draw](#), [moveTo](#))
- For each kind of shape you provide an implementation of the operations
- OO easier to answer *What is a circle?* and add new shapes
- ADT easier to answer *How do you draw a shape?* and add new operations
- **Health Warning and Optional Material** Discussions about the merits of [Functional programming](#) and [Object-oriented programming](#) tend to look like the disputes between [Lilliput](#) and [Blefuscu](#)
- [Abstract data type](#) article contrasts ADT and OO as algebra compared to co-algebra
- [What does coalgebra mean in the context of programming?](#) is a fairly technical but accessible article.
- [What does the forall keyword in Haskell do?](#) — is an accessible article on *Existential Quantification*
- [Bart Jacobs Coalgebra](#)
- [nLab Coalgebra](#)
- Beware the distinction between *concepts* and *features* in programming languages — see [OOP Disaster](#)
- **Not for this session** — this slide is here just in case

Further Links on ADT/OOP

- [Object Oriented Programming in Haskell](#) (15 October 2018)
- [Object-Oriented Haskell](#) (15 October 2018)

```

1 data Shape
2   = Circle Point Radius
3   | Square Point Size

```

```

5 draw :: Shape -> Pict
6 draw (Circle p r) = drawCircle p r
7 draw (Square p s) = drawRectangle p s s

9 moveTo :: Point -> Shape -> Shape
10 moveTo p2 (Circle p1 r) = Circle p2 r
11 moveTo p2 (Square p1 s) = Square p2 s

13 shapes :: [Shape]
14 shapes = [Circle (0,0) 1, Square (1,1) 2]

16 shapes01 :: [Shape]
17 shapes01 = map (moveTo (2,2)) shapes

```

- Example based on Lennart Augustsson email of 23 June 2005 on Haskell list

```

1 class IsShape shape where
2   draw :: shape -> Pict
3   moveTo :: Point -> shape -> shape

5 data Shape = forall aTVar . (IsShape aTVar) => Shape aTVar

7 data Circle = Circle Point Radius
8 instance IsShape Circle where
9   draw (Circle p r) = drawCircle p r
10  moveTo p2 (Circle p1 r) = Circle p2 r

12 data Square = Square Point Size
13 instance IsShape Square where
14   draw (Square p s) = drawRectangle p s s
15   moveTo p2 (Square p1 s) = Square p2 s

17 shapes :: [Shape]
18 shapes = [Shape (Circle (0,0) 10), Shape (Square (1,1) 2)]

20 shapes01 :: [Shape]
21 shapes01 = map (moveShapeTo (2,2)) shapes
22           where
23             moveShapeTo p (Shape s) = Shape (moveTo p s)

```

10.1.1 Haskell Code — Commentary

- The following is a *very* brief commentary on the Haskell code

```

1 data Shape
2   = Circle Point Radius
3   | Square Point Size

```

- **data** defines an **algebraic datatype** with two constructors (**Circle**, **Square**) which each take two arguments of types assumed to be defined elsewhere (*Point*, *Radius*, *Size*)

```

5 draw :: Shape -> Pict
6 draw (Circle p r) = drawCircle p r
7 draw (Square p s) = drawRectangle p s s

9 moveTo :: Point -> Shape -> Shape
10 moveTo p2 (Circle p1 r) = Circle p2 r
11 moveTo p2 (Square p1 s) = Square p2 s

```

- The lines starting **draw** :: and **moveTo** :: are *type signatures* which specify types for the functions **draw** and **moveTo**
- In each case the next couple of lines define the function
- Note that function and constructor application is denoted by juxtaposition, is left associative and is more binding than (almost) anything else

- $(f \ x \ y)$ means $((f \ x) \ y)$

```
1 class IsShape shape where
2   draw :: shape -> Pict
3   moveTo :: Point -> shape -> shape
```

- The above declares the type class **IsShape** which includes the type signatures of the functions which must be defined in any instance declaration

```
8 instance IsShape Circle where
9   draw (Circle p r) = drawCircle p r
10  moveTo p2 (Circle p1 r) = Circle p2 r
```

- The above is an instance declaration for the type **Circle** to be a member of the type class **IsShape**

```
5 data Shape = forall aTVar . (IsShape aTVar) => Shape aTVar
```

- The above declares **Shape** to have the constructor **Shape** which takes a type variable **aTVar** which is a member of the type class **IsShape**
- See [What does the forall keyword do ?](#)
- Understanding **forall** requires some knowledge of [first-order logic](#) so initially may appear a bit subtle.
- Norman Ramsey in the above StackOverflow article recommends [Launchbury and Peyton Jones \(1994\) Lazy Functional State Threads](#)
- Also see [HaskellWiki: Existential type](#)
- Note that in Haskell reserved words and variable names (including functions) and type variables start with a lower case letter while names for particular values or types start with an upper case letter (with a few exceptions)

- **Default: Language Main Constructs**

data map class where forall instance (Haskell)

- **Language Builtin other**

upper find count (Python)

- **User Defined**

Shape Circle Square IsShape draw moveTo (Haskell)

- **Meta**

decls, decl1, decl2, declK, expr, alts

- **Special**

GHCi> (Haskell GHCi)

- **Meta Builtin**

shape aTVar type variables (Haskell)

- **Meta User Defined**

Pict Point Radius drawCircle drawRectangle Haskell

The Expression Problem

- The *Expression Problem* describes a dual problem that neither Object Oriented Programming nor Functional Programming fully addresses.
- If you want to add a new thing, Object Oriented Programming makes it easy (since you can simply create a new class) but Functional Programming makes it harder (since you have to edit every function that accepts a thing of that type)
- If you want to add a new function, Functional Programming makes it easy (simply add a new function) while Object Oriented Programming makes it harder (since you have to edit every class to add the function)
- [Wikipedia: Expression problem](#)
- [Bendersky: The Expression Problem](#) and [More thoughts](#)
- [C2 Wiki: Expression Problem](#)
- [What is the 'expression problem'?](#)
- [Philip Wadler: The Expression Problem](#)
- [Debidda: Expression Problem](#)

[ToC](#)

10.2 Abstract Data Type — Queue

- [Queue Abstract Data Type](#) — operations
- [makeEmptyQ](#) returns empty queue
- [isEmptyQ](#) takes queue, returns Boolean
- [addToQ](#) takes queue, item, returns queue with item added at back
- [headOfQ](#) takes queue, returns item at front
- [tailOfQ](#) takes queue, returns queue without front item
- Other operations
- [removeFrontQ](#) takes queue, returns pair of item on the front and queue with item removed
- [sizeQ](#) to save calculating it
- [isFullQ](#) for a bounded queue
- **Pre, Post Conditions, Axioms** should be **complete**
- They define all permissible inputs to the functions (or methods)
- They define the outcome of all applications of the functions
- Composition of the functions constructs all possible members of the ADT set
- **Pre-conditions, Post-conditions, Axioms**
- [makeEmptyQ\(\)](#)
- [Pre True](#)

- **Post** Return value `q` is an empty queue
- **Axiom** `makeEmptyQ() == EmptyQ`
- `isEmptyQ()`
- **Pre** `True`
- **Post** Returns `True` if `q` is empty, otherwise `False`
- **Axiom** `isEmptyQ(makeEmptyQ()) == True`
- `isEmptyQ(addToQ(q,x)) == False`
- **Exercise** complete this for the other operations
- **Pre-conditions, Post-conditions, Axioms**
- `addToQ()`
- **Pre** `True`
- **Post** Returns queue with `x` at back, front part is input queue
- `headOfQ()`
- **Pre** Argument `q` is non-empty
- **Post** Return value is item at the front (queue is unchanged)
- **Axioms** `headOfQ(makeEmptyQ()) == error`
- `headOfQ(addToQ(makeEmptyQ(),x)) == x`
- `headOfQ(addToQ(q,x)) == headOfQ(q)`
- **Pre-conditions, Post-conditions, Axioms**
- `tailOfQ()`
- **Pre** `True`
- **Post** Returns queue without first item
- **Axioms** `tailOfQ(makeEmptyQ()) == error`
- `tailOfQ(addToQ(makeEmptyQ(),x)) == EmptyQ`
- `tailOfQ(addToQ(q,x)) == addToQ(tailOfQ(q),x)`
- **Queue Implementation**
- **Using Lists as Queues** section 5.1.2 of the *Tutorial*
- **Quote:** It is also possible to use a list as a queue, where the first element added is the first element retrieved (first-in, first-out); however, lists are not efficient for this purpose. While appends and pops from the end of list are fast, doing inserts or pops from the beginning of a list is slow (because all of the other elements have to be shifted by one).
- Could use `collections.deque` but we will use a pair of lists — See (Okasaki, 1998, page 42)
- **Queue Implementation 1**
- Using a `namedtuple()`
- A factory function for creating tuple subclasses with named fields

```

5 from collections import namedtuple
7 Qp1 = namedtuple('Qp1', ['frs', 'rbks'])

```

• Queue Implementation 1 main operations

```

9 def makeEmptyQp1():
10     return Qp1([], [])
12 def isEmptyQp1(q):
13     return q.frs == []
15 def addToQp1(q, x):
16     return checkQp1(q.frs, [x] + q.rbks[:])
18 def headOfQp1(q):
19     if q.frs == [] :
20         RuntimeError("headOfQp1_applied_to_empty_queue")
21     else:
22         return q.frs[0]
24 def tailOfQp1(q):
25     if q.frs == [] :
26         RuntimeError("tailOfQp1_applied_to_empty_queue")
27     else:
28         return checkQp1(q.frs[1:], q.rbks[:])

```

• Queue Implementation 1 checkQp1()

```

30 def checkQp1(frs, rbks):
31     if frs == [] :
32         bks = rbks[:]
33         bks.reverse()
34         return Qp1(bks, [])
35     else :
36         return Qp1(frs, rbks)

```

- Note copying of arguments — see below for reason
- Note also in `stringQp1Items` below at line 47 on page 62
- `implicit line joining` using `()` (why is this needed ??)
- Note use of recursion

Python Argument Passing

- **Functions, Immutable and Mutable Arguments**
- Immutable arguments are passed by value
- Mutable arguments are passed by reference
- Immutable: numbers, strings, tuples
- Mutable: Lists, dictionaries, sets, and most objects in user classes

```

>>> def changer (a,b) :
...     a = 2
...     b[0] = 'spam'
...
>>> n = 1
>>> xs = [1,2]
>>> changer(n, xs)
>>> (n,xs)
(1, ['spam', 2])

```

• Queue Implementation 1 conversion operations

```

38 def stringQp1(q) :
39     return ("<" + stringQp1Items(q) + ">")
41
42 def stringQp1Items(q) :
43     if isEmptyQp1(q) :
44         return ""
45     elif isEmptyQp1(tailOfQp1(q)) :
46         return str(headOfQp1(q))
47     else :
48         return ( str(headOfQp1(q))
49                 + ",_" + stringQp1Items(tailOfQp1(q)) )
50
51 def buildQp1(xs,q) :
52     if xs == [] :
53         return q
54     else :
55         return buildQp1(xs[1:],addToQp1(q,xs[0]))
56
57 def listToQp1(xs) :
58     return buildQp1(xs, makeEmptyQp1())

```

• Queue Implementation 1 test code

```

61 q11 = listToQp1([1,2,3,1])
62
63 q12 = tailOfQp1(q11)
64
65 assert q11 == Qp1(frs=[1], rbks=[1, 3, 2])
66
67 assert stringQp1(q11) == '<1,_,3,_,1>'
68
69 assert q12 == Qp1(frs=[2, 3, 1], rbks=[])
70
71 assert stringQp1(q12) == '<2,_,3,_,1>'

```

• Queue Implementation 2

- Modify to add **size**
- Store in tuple to save calculating each time

```

75 Qp2 = namedtuple('Qp2',['frs','rbks','sz'])

```

• Exercise Add **size()** operation and other modifications

• Queue Implementation 2 main operations

```

77 def makeEmptyQp2():
78     return Qp2([],[], 0)
79
80 def isEmptyQp2(q):
81     return q.frs == []
82
83 def addToQp2(q,x):
84     return checkQp2(q.frs, [x] + q.rbks[:], q.sz + 1)
85
86 def headOfQp2(q):
87     if q.frs == [] :
88         RuntimeError("headOfQp2_applied_to_empty_queue")
89     else:
90         return q.frs[0]
91
92 def tailOfQp2(q):
93     if q.frs == [] :
94         RuntimeError("tailOfQp2_applied_to_empty_queue")
95     else:
96         return checkQp2(q.frs[1:], q.rbks[:], q.sz - 1)

```

• Queue Implementation 2 **sizeQp2()**, **checkOp1()**

```

98 def sizeOfQp2(q) :
99     return q.sz

101 def checkQp2(frs, rbks, sz):
102     if frs == [] :
103         bks = rbks[:]
104         bks.reverse()
105         return Qp2(bks, [], sz)
106     else :
107         return Qp2(frs, rbks, sz)

```

- Note also in `stringQp2Items` below at line 118 on page 63
- `implicit line joining` using `()` (why is this needed ??)
- Note use of recursion
- **Queue Implementation 2** conversion operations

```

109 def stringQp2(q) :
110     return "<" + stringQp2Items(q) + ">"

112 def stringQp2Items(q) :
113     if isEmptyQp2(q) :
114         return ""
115     elif isEmptyQp2(tailOfQp2(q)) :
116         return str(headOfQp2(q))
117     else :
118         return ( str(headOfQp2(q))
119                 + ",_" + stringQp2Items(tailOfQp2(q)) )

121 def buildQp2(xs,q) :
122     if xs == [] :
123         return q
124     else :
125         return buildQp2(xs[1:],addToQp2(q,xs[0]))

127 def listToQp2(xs) :
128     return buildQp2(xs, makeEmptyQp2())

```

- **Queue Implementation 2** test code

```

132 q21 = listToQp2([1,2,3,1])
134 q22 = tailOfQp2(q21)
136 assert q21 == Qp2(frs=[1], rbks=[1, 3, 2], sz=4)
138 assert stringQp2(q21) == '<1,_2,_3,_1>'
140 assert q22 == Qp2(frs=[2, 3, 1], rbks=[], sz=3)
142 assert stringQp2(q22) == '<2,_3,_1>'

```

[ToC](#)

10.3 ADT Lists in Lists

- Lists implemented naively as linked lists have some operations that take constant time and some that are linear in the length of the list
- Adding an element to the front of a list takes constant time while adding an element to the rear takes linear time
- This section reimplements lists using a pair of lists that overcomes this asymmetry in efficiency giving constant time for all operations.

- The basic idea is quite simple: break the list in two and reverse the second half
- This means that the last element is the first element of the second list
- A problem arises when one attempts to remove an element — in some cases the list has to be reorganised into two halves
- The criteria for reorganising gives the clue in how to write the code
- This implementation is based on [Bird and Gibbons \(2020, chp 3\)](#)
- The idea is attributed to [Gries \(1981, page 250\)](#) and [Hood and Melville \(1980\)](#)
- See also [Hoogerwoord \(1992\)](#)
- We give the code in Python from [SymmetricLists.py](#) with Haskell type specifications and declarations given as comments
- Here is the type alias declaration as a comment along with [fromSL](#) which converts back from symmetric lists to standard lists — this is known as the *abstraction function*

```

12 # type SymList a = ([a],[a])
14 # Abstraction function
16 # fromSL :: SymList a -> [a]
18 def fromSL (pr) :
19     xs = pr[0]
20     ys = pr[1]
21     return xs + reverseF (ys)
23 def reverseF (xs) :
24     ys = xs[:]
25     ys.reverse()
26     return ys

```

- The [abstraction function](#) captures the relationship between the implementation of an operation on the representing type and its abstract type with an equation
- The *Eureka* bit of the implementation is spotting the *representation invariant* that our definitions both exploit and maintain

```

28 # repInvSL :: SymList a -> Bool
30 def repInvSL (pr) :
31     xs = pr[0]
32     ys = pr[1]
33     xsTest = ((not isEmpty (xs))
34              or (isEmpty (ys) or singleton (ys)))
35     ysTest = ((not isEmpty (ys))
36              or (isEmpty (xs) or singleton (xs)))
37     return (xsTest and ysTest)

```

- This says if one list is empty then the other must be either empty or a singleton
- This tells us when we need to reorganise the lists
- Here are the service operations for empty lists and singletons

```

39 # isEmpty :: [a] -> Bool
41 def isEmpty (xs) :
42     return (xs == [])
44 # isEmptySL :: SymList a -> Bool
46 def isEmptySL (pr) :

```

```

47 xs = pr[0]
48 ys = pr[1]
49 return (isEmpty (xs) and isEmpty (ys))

51 def makeEmptySL() :
52     return ([],[])

54 # singleton :: [a] -> Bool

56 def singleton (xs) :
57     return (len(xs) == 1)

59 # singletonSL :: SymList a -> Bool

61 def singletonSL (pr) :
62     xs = pr[0]
63     ys = pr[1]
64     return ((isEmpty (xs) and singleton (ys))
65             or (isEmpty (ys) and singleton (xs)))

```

- Constructor operations
- Both of these definitions make use of the representation invariant

```

67 # Constructor functions

69 # consSL :: a -> SymList a -> SymList a

71 def consSL (x, pr) :
72     xs = pr[0]
73     ys = pr[1]
74     if isEmpty (ys) :
75         return ([x],xs)
76     else :
77         return ([x] + xs, ys)

79 # snocSL :: a -> SymList a -> SymList a

81 def snocSL (x, pr) :
82     xs = pr[0]
83     ys = pr[1]
84     if isEmpty (xs) :
85         return (ys,[x])
86     else :
87         return (xs, [x] + ys)

```

- Inspectors

```

91 # headSL :: SymList a -> a

93 def headSL (pr) :
94     xs = pr[0]
95     ys = pr[1]
96     if isEmpty (xs) :
97         if isEmpty (ys) :
98             raise RuntimeError("headSL_([],[])")
99         else :
100             return ys[0]
101     else :
102         return xs[0]

104 # lastSL :: SymList a -> a

106 def lastSL (pr) :
107     xs = pr[0]
108     ys = pr[1]
109     if isEmpty (ys) :
110         if isEmpty (xs) :
111             raise RuntimeError("tailSL_([],[])")
112         else :
113             return xs[0]
114     else :
115         return ys[0]

```

- `tailSL`
- Notice how the representation invariant is maintained

```

118 # tailSL :: SymList a -> SymList a
120 def tailSL (pr) :
121     xs = pr[0]
122     ys = pr[1]
123     if isEmpty (xs) :
124         if isEmpty (ys):
125             raise RuntimeError("tailSL_([],[])")
126         else:
127             return ([],[])
128     elif singleton (xs) :
129         splitPt = len(ys) // 2
130         (us,vs) = (ys[:splitPt],ys[splitPt:])
131         return (reverseF (vs), us)
132     else :
133         return (xs[1:],ys)

```

- `initSL`

```

135 # initSL :: SymList a -> SymList a
137 def initSL (pr) :
138     xs = pr[0]
139     ys = pr[1]
140     if isEmpty (ys) :
141         if isEmpty (xs):
142             raise RuntimeError("initSL_([],[])")
143         else:
144             return ([],[])
145     elif singleton (ys) :
146         splitPt = len(xs) // 2
147         (us,vs) = (xs[:splitPt],xs[splitPt:])
148         return (us, reverseF (vs))
149     else :
150         return (xs,ys[1:])

```

- The implementations are designed to satisfy the six equations:
- The equations are expressed here in Haskell notation

```

1  # -- The implementation satisfies the following
2  # --
3  # -- (cons x . fromSL) ps == (fromSL . consSL x) ps
4  # -- (snoc x . fromSL) ps == (fromSL . snocSL x) ps
5  # -- (tail . fromSL) ps == (fromSL . tailSL) ps
6  # -- (init . fromSL) ps == (fromSL . initSL) ps
7  # -- (head . fromSL) ps == headSL ps
8  # -- (last . fromSL) ps == lastSL ps

```

- Each of the operations apart from `tailSL` and `initSL` take constant time
- `tailSL` and `initSL` can take linear time in the worst case but they take amortised constant time — see the references for derivation
- Note that Haskell `Data.Sequence` uses 2-3 Finger Trees for better performance
- Ex (1) Write down all the ways "abcd" can be represented as a symmetric list.
Give examples to show how each of these representations can be generated.
- Ex (2) Define `lengthSL`
- Ex (3) Implement `dropWhileSL` so that

```
# dropWhile . fromSL = fromSL . dropWhileSL
```

- **Ex (4)** Define `initsSL` with the type

```
# initsSL :: SymList a -> SymList (SymList a)
```

Write down the equation which expresses the relationship between `fromSL`, `initsSL`, and `inits`.

- Note Exs (3) and (4) use Python beyond M269
- **Ans (1)** There are three ways:

```
("a", "dcb"), ("ab", "dc"), ("abc", "d")
```

```
Python3>>> prs1 = consSL('a', ([], []))
Python3>>> prs1
(['a'], [])
Python3>>> prs2 = snocSL('b', prs1)
Python3>>> prs2
(['a'], ['b'])
Python3>>> prs3 = snocSL('c', prs2)
Python3>>> prs3
(['a'], ['c', 'b'])
Python3>>> prs4 = snocSL('d', prs3)
Python3>>> prs4
(['a'], ['d', 'c', 'b'])

Python3>>> prs1a = snocSL('a', ([], []))
Python3>>> prs1a
([], ['a'])
Python3>>> prs2a = snocSL('b', prs1a)
Python3>>> prs2a
(['a'], ['b'])
```

- **Ans (1)** There are three ways:

```
("a", "dcb"), ("ab", "dc"), ("abc", "d")
```

```
Python3>>> prs1 = consSL('d', ([], []))
Python3>>> prs1
(['d'], [])
Python3>>> prs2 = consSL('c', prs1)
Python3>>> prs2
(['c'], ['d'])
Python3>>> prs3 = consSL('b', prs2)
Python3>>> prs3
(['b', 'c'], ['d'])
Python3>>> prs4 = consSL('a', prs3)
Python3>>> prs4
(['a', 'b', 'c'], ['d'])
```

- Functional programmers will spot that the first is an instance of a `foldl` while the third is an instance of a `foldr`
- **Ans (2)** Define `lengthSL`

```
# lengthSL :: SymList -> Int

def lengthSL (pr) :
    xs = pr[0]
    ys = pr[1]
    return len(xs) + len(ys)
```

- Note that we have made `lengthSL` a function in the SymmetricList ADT that has access to the underlying implementation
- We could have done that without giving it access but that would have a cost
- Since `lengthSL` is used a lot we give it access

- **Ans (3)** Implement `dropWhileSL`

```
def dropWhileSL(pred, xs) :
    if isEmptySL(xs) :
        return makeEmptySL
    elif pred(headSL(xs)) :
        return dropWhileSL(pred, (tailSL(xs)))
    else :
        return xs
```

- Note `pred` is a function object, defining a function that takes one argument and returns a Boolean
- `dropWhileSL` is an example of a *higher order function* — a function that can take or receive a function as an argument

- **Ans (3)** Test code

```
def greaterThan5 (x) :
    return x > 5

testList3 = [25, 35, 4, 45]
testList3SL = toSL(testList3)

test3A = (testList3SL == ([25, 35], [45, 4]))
test3B = (fromSL(dropWhileSL(greaterThan5, testList3SL)) == [4, 45])
```

- **Ans (4)** Implement `initsSL(xs)`

```
def initsSL(xs) :
    if isEmptySL(xs) :
        return (snocSL(xs, makeEmptySL()))
    else :
        return (snocSL(xs, (initsSL(initSL(xs)))))
```

- **Ans (4)** Test code

```
test4inits = initsSL(testList3SL)
test4initsFromSL = fromSL (test4inits)

def fromSLs(prs) :
    if prs == [] :
        return []
    else :
        return [fromSL(prs[0])] + fromSLs(prs[1:])

def displayInitsSL(xs) :
    return fromSLs(fromSL(initsSL(xs)))

test4Display = displayInitsSL(testList3SL)
```

```
Python3>>> testList3
[25, 35, 4, 45]
Python3>>> testList3SL
([25, 35], [45, 4])
Python3>>> test4Display
[[], [25], [25, 35], [25, 35, 4], [25, 35, 4, 45]]
```

- **Ans (4)** Relationship between `fromSL`, `initsSL`, and `inits`.
- In Haskell first, followed by Python
- Why that way round ? Haskell makes it more obvious what is going on

```
(inits . fromSL) prs = (map fromSL . fromSL . initsSL) prs
```

- `(.)` is the Haskell function composition operator

```
inits(fromSL(prs)) = map(fromSL, fromSL(initsSL(prs)))
```

- `map` in Python does (roughly) the same as Haskell `map`
- However, in Python `map` returns an `iterator`, which represents a stream of data
- This means we need some extra code to print the result — maybe using the type constructor `list(iterable)`
or the alternative `fromSLs` and `displayInitsSL`
- **Ans (4)** Using a list comprehension instead of an explicit `map`

```
def displayInitsSL01(symList) :  
    return [fromSL(pr) for pr in fromSL(initsSL(symList))]
```

[ToC](#)

11 Future Work

Programming, Debugging, Psychology

Although programming techniques have improved immensely since the early days, the process of finding and correcting errors in programming — known graphically if inelegantly as *debugging* — still remains a most difficult, confused and unsatisfactory operation. The chief impact of this state of affairs is psychological. Although we are happy to pay lip-service to the adage that to err is human, most of us like to make a small private reservation about our own performance on special occasions when we really try. It is somewhat deflating to be shown publicly and incontrovertibly by a machine that even when we do try, we in fact make just as many mistakes as other people. If your pride cannot recover from this blow, you will never make a programmer.

Christopher Strachey, Scientific American 1966 vol 215 (3) September pp112-124

- To err is human, to really foul things up requires a computer.
- Attributed to [Paul R. Ehrlich](#) in [101 Great Programming Quotes](#)
- Attributed to [Bill Vaughn](#) in [Quote Investigator](#)
- Derived from [Alexander Pope](#) (1711, [An Essay on Criticism](#))
- *To Err is Humane; to Forgive, Divine*
- This also contains
*A little learning is a dangerous thing;
Drink deep, or taste not the [Pierian Spring](#)*
- In programming, this means you have to *read the fabulous manual* ([RTFM](#))

Sorting, Searching, Binary Trees

- Recursive function definitions
- Inductive data type definitions
 - A *list* is either an empty list or a first item followed by the rest of the list

- A *binary tree* is either an empty tree or a node with an item and two sub-trees
- Recursive definitions often easier to find than iterative
- Sorting
- Searching
- Both use binary tree structure
- TMA01 Thursday 19 December 2024 TMA01
- Sunday 5 January 2025 Tutorial Online Sorting
- Sunday 12 January 2025 Tutorial Online Binary Trees
- Sunday 9 February 2025 Tutorial Online Binary Trees
- TMA02 Thursday 6 March 2025 TMA02

[ToC](#)

12 Example Algorithm Design — Haskell

Binary Search — Haskell

- The notes following give two implementations of *Binary Search* in [Haskell](#)
- **Note:** these are not part of M269 and are purely for comparison for those interested
- The first is a direct translation of the recursive Python version
- The second is derived from http://rosettacode.org/wiki/Binary_search and is more idiomatic Haskell
- The code for both implementations is in the file [M269BinarySearch.hs](#) (which should be near the file of these slides)

12.1 Binary Search — Haskell — version 1

Binary Search — Haskell — 1 (a)

```
1 module M269BinarySearch where
3 import Data.Array
4 import Data.List
```

- A Haskell script starts with a module header which starts with the reserved identifier, `module` followed by the module name, `M269BinarySearch`
- The module name must start with an upper case letter and is the same as the file name (without its extension of `.hs` or `.lhs`)
- Haskell uses *layout* (or the *off-side rule*) to determine scope of definitions, similar to Python
- The body of the module follows the reserved identifier `where` and starts with `import` declarations
- This imports the libraries `Data.List`, `Data.Array`

Binary Search — Haskell — 1 (b)

```

6  binarySearch :: Ord a => [a] -> a -> Maybe Int
8  binarySearch xs val
9    = binarySearch01 xs val (lo,hi)
10     where
11       lo = 0
12       hi = Tlength xs - 1

```

- Line 8 is the definition of `binarySearch`
- The preceding line, 6, is the *type signature*
- `binarySearch` takes a list and a value of type `a` (in the class `Ord` for ordering) and returns a `Maybe Int` — `a` is a *type variable*
- The `Maybe a` type is an *algebraic data type* which is the union of the *data constructors* `Nothing` and `Just a`

```
data Maybe a = Nothing | Just a
```

Code Description 1

- `f :: t` is a *type signature* for variable `f` that reads *f is of type t*
- `f :: t1 -> t2` means that `f` has the type of a function that takes elements of type `t1` and returns elements of type `t2`
- The *function type arrow* `->` associates to the right
 - `f :: t1 -> t2 -> t3` means
 - `f :: t1 -> (t2 -> t3)`
- `f x` — function application is denoted by juxtaposition and is more binding than (almost) any other operation.
- *Function application* is left associative
 - `f x y` means
 - `(f x) y`

Binary Search — Haskell — 1 (c)

```

14  binarySearch01 :: Ord a
15    => [a] -> a -> (Int, Int) -> Maybe Int
17  binarySearch01 xs val (lo,hi)
18    = if hi < lo then Nothing
19      else
20        let mid = (lo + hi) `div` 2
21            guess = xs !! mid
22        in
23          if val == guess
24            then Just mid
25          else if val < guess
26            then binarySearch01 xs val (lo,mid-1)
27          else binarySearch01 xs val (mid + 1, hi)

```

Code Description 2

- A `let` expression has the form

```
let decls in expr
```

- `decls` is a number of *declarations*
- `expr` is an expression (which is the scope of the declarations)
- `div` is the integer division function
- In ``div``, the grave accents (```) make a function into an infix operator (OK, that is syntactic sugar I need not have introduced — and my formatting program has coerced the grave accent to a left single quotation mark Unicode U+2018, not the grave accent U+0060)
- `(!!)` is the list index operator — first item has index 0

12.2 Binary Search — Haskell — version 2

Binary Search — Haskell — 2 (a)

```

29 binarySearchGen :: Integral a
30   => (a -> Ordering) -> (a, a) -> Maybe a
31 binarySearchGen p (lo,hi)
32   | hi < lo = Nothing
33   | otherwise =
34     let mid = (lo + hi) `div` 2 in
35     case p mid of
36       LT -> binarySearchGen p (lo, mid - 1)
37       GT -> binarySearchGen p (mid + 1, hi)
38       EQ -> Just mid

```

Code Description 3

- A `case` expression has the form

```
case expr of alts
```

`expr` is evaluated and whichever alternative of `alts` matches is the result

- The lines starting with `(|)` are *guarded* definitions — if the boolean expression to the right is `True` then the following expression is used
- `otherwise` is a synonym for `True`
- A *conditional* expression has the form

```
if expr then expr else expr
```

The first `expr` must be of type `Bool`

- *Guards* and *conditionals* are alternative styles in programming

Binary Search — Haskell — 2 (b)

```

40 binarySearchArray :: (Ix i, Integral i, Ord a)
41   => Array i a -> a -> Maybe i
42 binarySearchArray ary x
43   = binarySearchGen p (bounds ary)
44   where
45     p m = x `compare` (ary ! m)

```

```

47 binarySearchList :: Ord a
48                 => [a] -> a -> Maybe Int
49 binarySearchList xs val
50   = binarySearchGen p (0, length xs - 1)
51   where
52     p m = val 'compare' (xs !! m)

```

Code Description 4

- `compare` is a method of the `Ord` class, for *ordering*, defined in the standard *Prelude*

```

class (Eq a) => Ord a where
  compare :: a -> a -> Ordering
  (<), (<=), (>=), (>) :: a -> a -> Bool
  max, min :: a -> a -> a

  compare x y
    | x == y    = EQ
    | x <= y    = LT
    | otherwise = GT

data Ordering = LT | EQ | GT
deriving (Eq, Ord, Enum, Read, Show, Bounded)

```

- Minimal type-specific definitions required are `compare` or `(==)` and `(<=)`
- `!` and `!!` are the array and list indexing operators

12.3 Binary Search — Haskell — Comparison

- The first version with `binarySearch` and `binarySearch01` is very similar to the *Python* recursive version `binarySearchRec`
- In the *Haskell* case an explicit helper function is used
- The second version is more general: `binarySearchGen` can be used with any type that is indexed by a data type in the `Integral` class
- `binarySearchArray` and `binarySearchList` specialise the function to arrays or lists.
- For the Haskell `Array` data type see the [Haskell Report](#)
- Idiomatic Haskell tends to be more general and make use of higher order functions, type classes and advanced features.

[ToC](#)

13 Web Links & References

- Python Online IDEs
 - **Repl.it** <https://repl.it/languages/python3> (Read-eval-print loop)
 - **TutorialsPoint CodingGround Python 3** https://www.tutorialspoint.com/execute_python3_online.php
 - **TutorialsPoint CodingGround Haskell ghci** https://www.tutorialspoint.com/compile_haskell_online.php

- The *offside rule* (using layout to determine the start and end of code blocks) comes originally from [Landin \(1966\)](#) — see [Off-side rule](#) for other programming languages that use this.
- The *step-by-step* approach to writing programs is described in [Glaser et al. \(2000\)](#)
- The difficulty in learning programs is described in many articles — see, for example, [Dehnadi and Bornat \(2006\)](#)
- **Inductive data type**
 - [Algebraic data type](#) composite type — possibly recursive [sum type](#) of [product types](#) — common in modern functional languages.
 - [Recursive data type](#) from [Type theory](#)

References

- Bentley, Jon (1984). Programming pearls: Algorithm design techniques. *Commun. ACM*, 27(9):865–873. ISSN 0001-0782. doi:10.1145/358234.381162. URL <http://doi.acm.org/10.1145/358234.381162>. 19, 21
- Bentley, Jon (1986). *Programming Pearls*. Addison Wesley. ISBN 0201103311. 21
- Bentley, Jon (2000). *Programming Pearls*. Addison Wesley, second edition. ISBN 0201657880. 21
- Bird, Richard (1998). *Introduction to Functional Programming using Haskell*. Prentice Hall, second edition. ISBN 0134843460. 21
- Bird, Richard (2010). *Pearls of Functional Algorithm Design*. Cambridge University Press. ISBN 0521513383. 21
- Bird, Richard (2014). *Thinking Functionally with Haskell*. Cambridge University Press. ISBN 1107452643. URL <https://www.cs.ox.ac.uk/publications/books/functional/>. 21
- Bird, Richard and Jeremy Gibbons (2020). *Algorithm Design with Haskell*. Cambridge University Press. ISBN 9781108869041. URL <https://www.cs.ox.ac.uk/publications/books/adwh/>. 25, 64
- Chambers (2014). *The Chambers Dictionary (13th Edition)*. Chambers. ISBN 1473602254. 35
- Cormen, Thomas H.; Charles E. Leiserson; Ronald L. Rivest; and Clifford Stein (2022). *Introduction to Algorithms*. MIT Press, fourth edition. ISBN 9780262046305. URL <https://mitpress.mit.edu/books/introduction-algorithms-fourth-edition>. 32
- Crockford, Douglas (2008). *JavaScript: The Good Parts*. O'Reilly. ISBN 0596517742. URL <http://javascript.crockford.com/>. 16
- Dehnadi, Saeed and Richard Bornat (2006). The camel has two humps. Web (Last checked 22 October 2015). URL <http://www.eis.mdx.ac.uk/research/PhDArea/saeed/paper1.pdf>. 74
- Gibbons, Jeremy (2008). Unfolding abstract datatypes. In *Mathematics of Program Construction*. Springer. doi:10.1007/978-3-540-70594-9_8. URL <http://www.comlab.ox.ac.uk/jeremy.gibbons/publications/adt.pdf>.

- Glaser, H; P J Hartel; and P W Garratt (2000). Programming by numbers: a programming method for complete novices. *The Computer Journal*, 43(4):252–265. A functional approach to learning programming. 74
- Goguen, J A; J W Thatcher; E G Wagner; and J B Wright (1977). Initial algebra semantics and continuous algebras. *Journal of the Association for Computing Machinery*, 24(1):68–95.
- Graham, Ronald L.; Donald E. Knuth; and Oren Patashnik (1994). *Concrete Mathematics: Foundation for Computer Science*. Addison Wesley, second edition. ISBN 0201558025. 25
- Gries, David (1981). *The Science of Programming*. Springer. ISBN 0387964800. URL <https://www.cs.cornell.edu/gries/July2016/The-Science-Of-Programming-Gries-038790641X.pdf>. 64
- Gries, David (1982). A note on a standard strategy for developing loop invariants and loops. *Science of Computer Programming*, 2(3):207–214.
- Gries, David (1989). The maximum-segment-sum problem. In *Formal development programs and proofs*, pages 33–36. Addison-Wesley Longman Publishing Co., Inc. 21
- Guttag, John (1977). Abstract data types and the development of data structures. *Communications of the ACM*, 20(6):396–404.
- Guttag, John (1980). Notes on type abstraction (version 2). *Software Engineering, IEEE Transactions on*, 6(1):13–23.
- Guttag, John V. and James J. Horning (1978). The algebraic specification of abstract data types. *Acta informatica*, 10(1):27–52.
- Guttag, John V; Ellis Horowitz; and David R Musser (1978). Abstract data types and software validation. *Communications of the ACM*, 21(12):1048–1064.
- Hood, Robert T and Robert C Melville (1980). Real time queue operations in pure Lisp. Technical report, Cornell University. 64
- Hoogerwoord, Rob R (1992). Functional pearls a symmetric set of efficient list operations. *Journal of Functional Programming*, 2(4):505–513. 64
- Jacobs, Bart and Jan Rutten (1997). A tutorial on (co) algebras and (co) induction. *Bulletin-European Association for Theoretical Computer Science*, 62:222–259.
- Landin, Peter J. (1966). The next 700 programming languages. *Communications of the Association for Computing Machinery*, 9:157–166. 74
- Launchbury, John and Simon L Peyton Jones (1994). Lazy functional state threads. *ACM SIGPLAN Notices*, 29(6):24–35. 58
- Liskov, Barbara and Stephen Zilles (1974). Programming with abstract data types. *ACM Sigplan Notices*, 9(4):50–59.
- Lutz, Mark (2011). *Programming Python*. O'Reilly, fourth edition. ISBN 0596158106. URL <http://learning-python.com/books/about-pp4e.html>.
- Lutz, Mark (2013). *Learning Python*. O'Reilly, fifth edition. ISBN 1449355730. URL <http://learning-python.com/books/about-1p5e.html>.
- Lutz, Mark (2025). *Learning Python*. O'Reilly Media, sixth edition. ISBN 1098171306. 13

Martelli, Alex; Anna Martelli Ravenscroft; Steve Holden; and Paul McGuire (2022). *Python in a Nutshell: A Desktop Quick Reference*. O'Reilly, fourth edition. ISBN 1098113551.

[13](#)

Mu, Shin-Cheng (2008). Maximum segment sum is back: deriving algorithms for two segment problems with bounded lengths. In *Proceedings of the 2008 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 31–39. ACM.

Okasaki, Chris (1995). Simple and efficient purely functional queues and dequeues. *Journal of functional programming*, 5(04):583–592.

Okasaki, Chris (1998). *Purely Functional Data Structures*. Cambridge University Press. ISBN 0-521-63124-6. [60](#)

Rutten, Jan JMM (2000). Universal coalgebra: a theory of systems. *Theoretical Computer Science*, 249(1):3–80.

van Rossum, Guido and Fred Drake (2003a). *An Introduction to Python*. Network Theory Limited. ISBN 0954161769.

van Rossum, Guido and Fred Drake (2003b). *The Python Language Reference Manual*. Network Theory Limited. ISBN 0954161785.

van Rossum, Guido and Fred Drake (2011a). *An Introduction to Python*. Network Theory Limited, revised edition. ISBN 1906966133.

van Rossum, Guido and Fred Drake (2011b). *The Python Language Reference Manual*. Network Theory Limited, revised edition. ISBN 1906966141.

[ToC](#)