M269 Python, Logic, ADTs

M269 Units 1, 2

Contents

1 M269 Tutorial: Python, Logic & ADTs

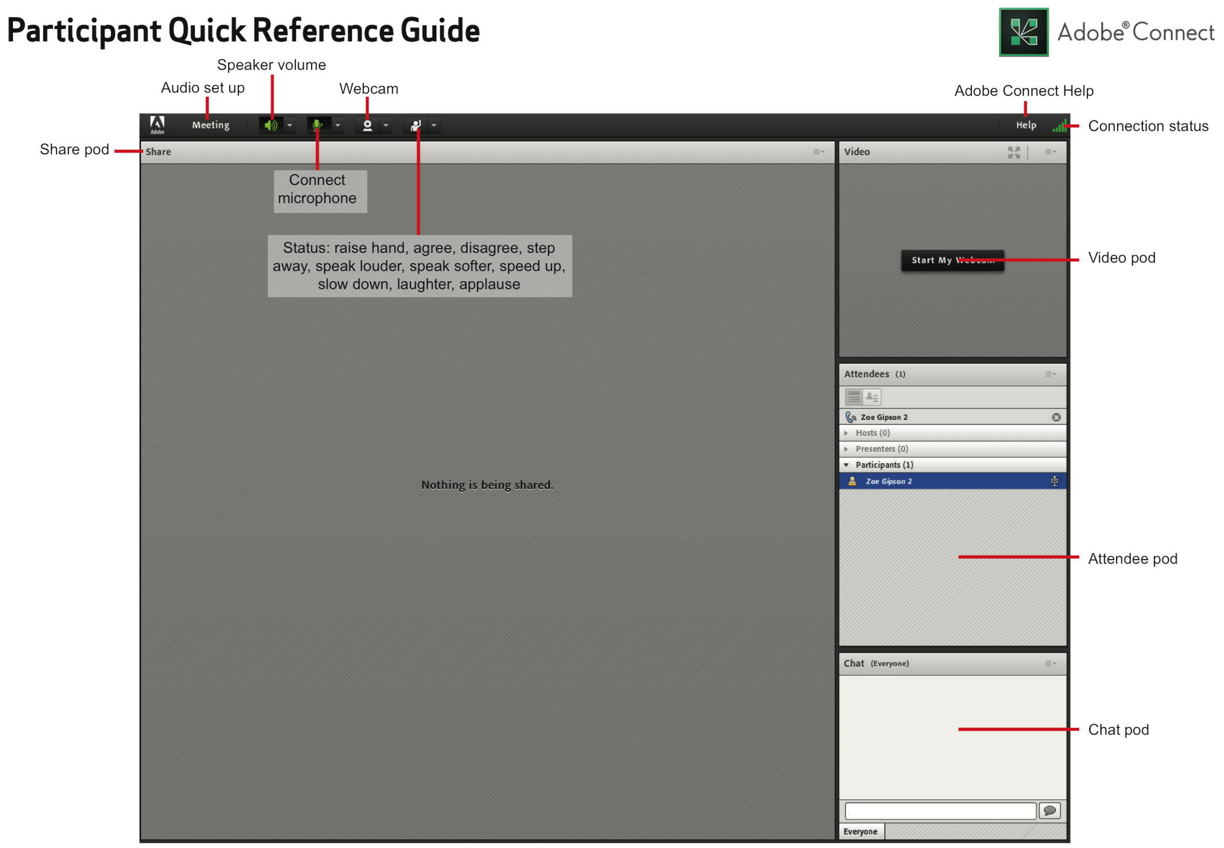
- Introductions
- Programming — Paradigms and Step-by-Step Guide
- Programming and Python
- Complexity and Big O Notation
- . . . with a little classical logic
- Abstract Data Type examples
- A look towards the next Units
  - Recursive function definitions
  - Inductive data type definitions
- Introductions
  - Name *Phil Molyneux*
  - Learning Style *Reads the manual*
  - Learnt last month *Framework for ADT vs OO* and wrote notes on *Monads to encapsulate programming with effects*
  - Read Stack Overflow Annual Developer Survey
  - Read Programming Languages Survey
  - You ?
- Do ask questions or raise points

- More material here that we can cover, some reference
- pmolyneux.co.uk/OU/M269FolderSync/M269TutorialNotes/M269TutorialProgPythonADT/
- Complexity example: M269TutorialProgPythonADT.py
- Rail Ticket example: M269TutorialProgPythonADT01.py
- Queue ADT example: M269TutorialProgPythonADT02Queue.py

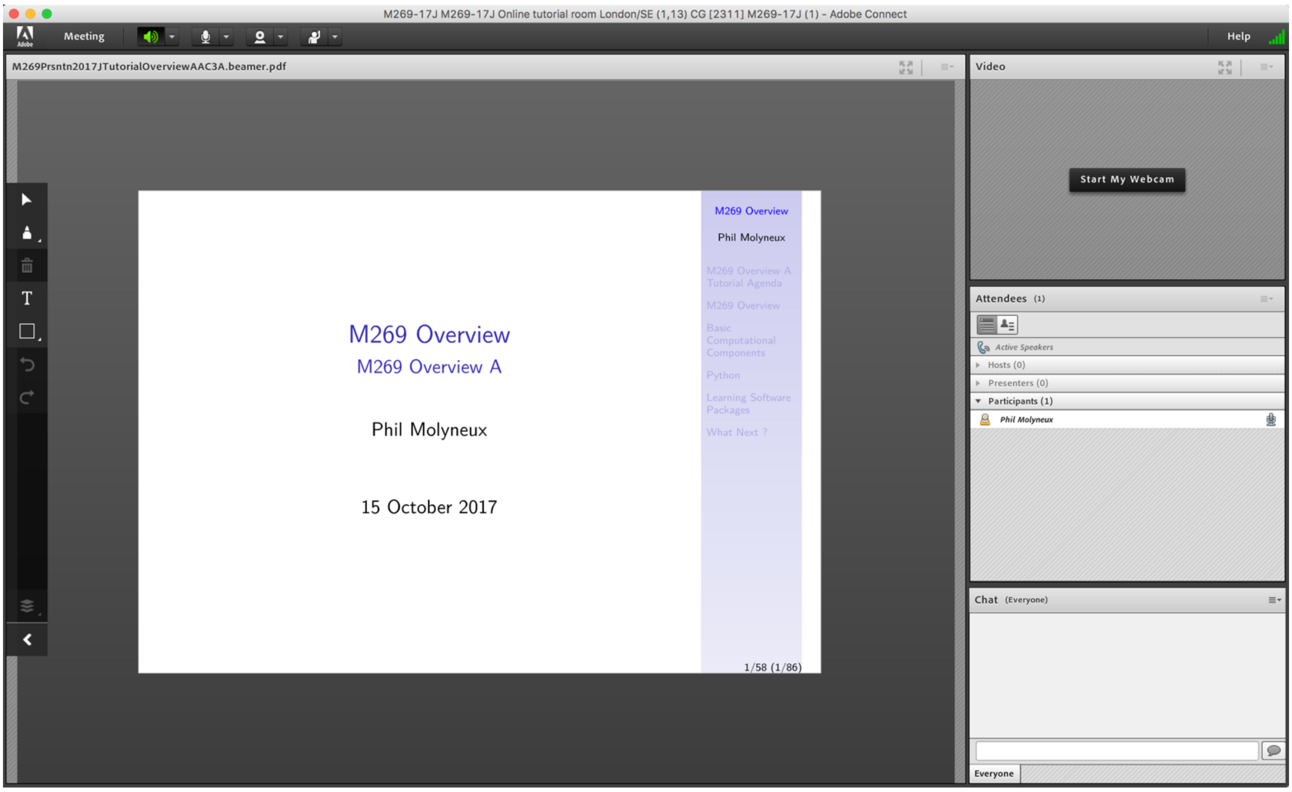
2 Adobe Connect Interface and Settings

2.1 Adobe Connect Interface — Student View

Adobe Connect Interface — Student Quick Reference



Adobe Connect Interface — Student View



2.2 Adobe Connect Settings

Adobe Connect Settings

- **Everybody: Audio Settings** Meeting › Audio Setup Wizard...
- **Audio** Menu bar › Audio › Microphone rights for Participants ✔
- Do not *Enable single speaker mode*
- **Drawing Tools** Share pod menu bar › Draw (1 slide/screen)
- Share pod menu bar › Menu icon › Enable Participants to draw ✔ gray
- Meeting › Preferences › Whiteboard › Enable Participants to draw ✔
- Cancel hand tool
- Do not enable green pointer...
- Meeting › Preferences › Attendees Pod Disable *Raise Hand notification*
- **Cursor** Meeting › Preferences › General tab › Host Cursors › Show to all attendees ✔ (default *Off*)
- Meeting › Preferences › Screen Share › Cursor › Show Application Cursor
- **Webcam** Menu bar › Webcam › Enable Webcam for Participants ✔
- **Recording** Meeting › Record Meeting... ✔

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**

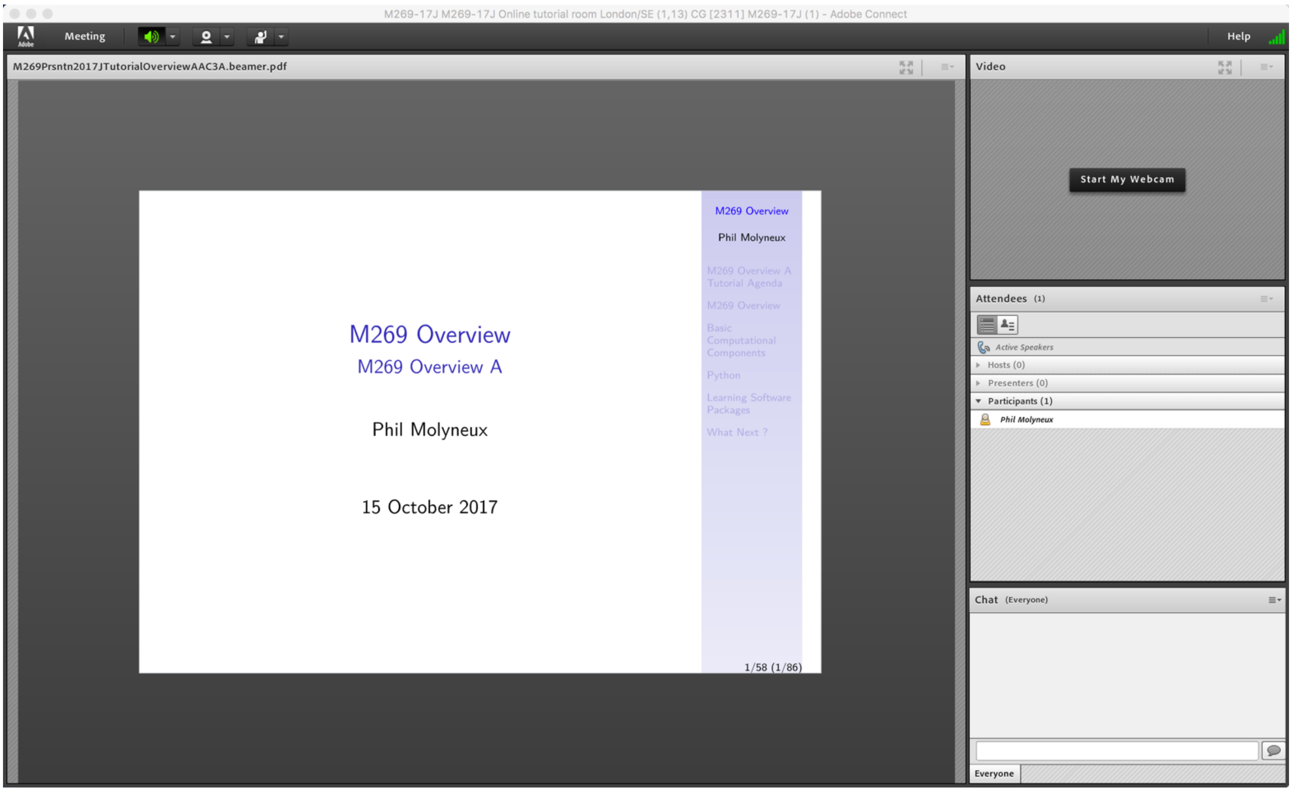
  Meeting > Enable/Disable Presenter Only Area

Adobe Connect — Keystroke Shortcuts

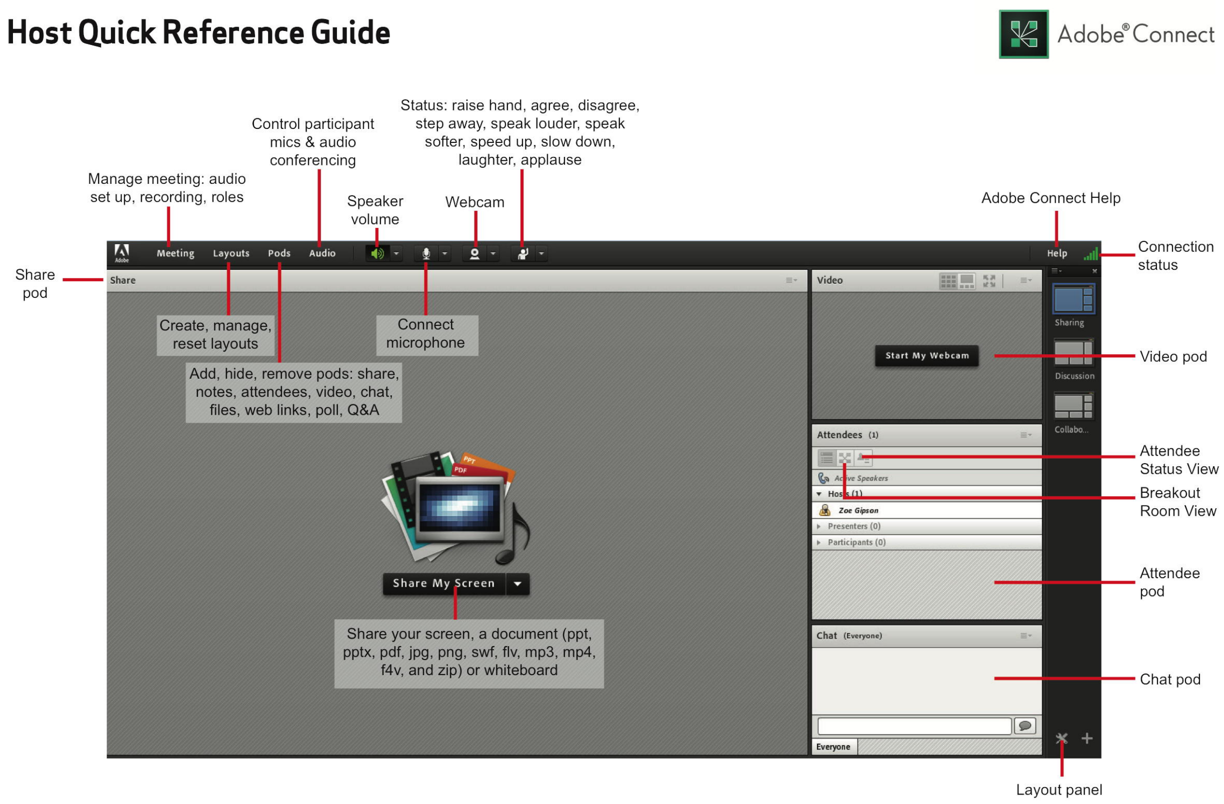
- Keyboard shortcuts in Adobe Connect
- **Toggle Mic** ⌘+M (Mac), Ctrl+M (Win) (On/Disconnect)
- **Toggle Raise-Hand status** ⌘+E
- **Close dialog box** ⎋ (Mac), Esc (Win)
- **End meeting** ⌘+\

2.3 Adobe Connect Interface — Student & Tutor Views

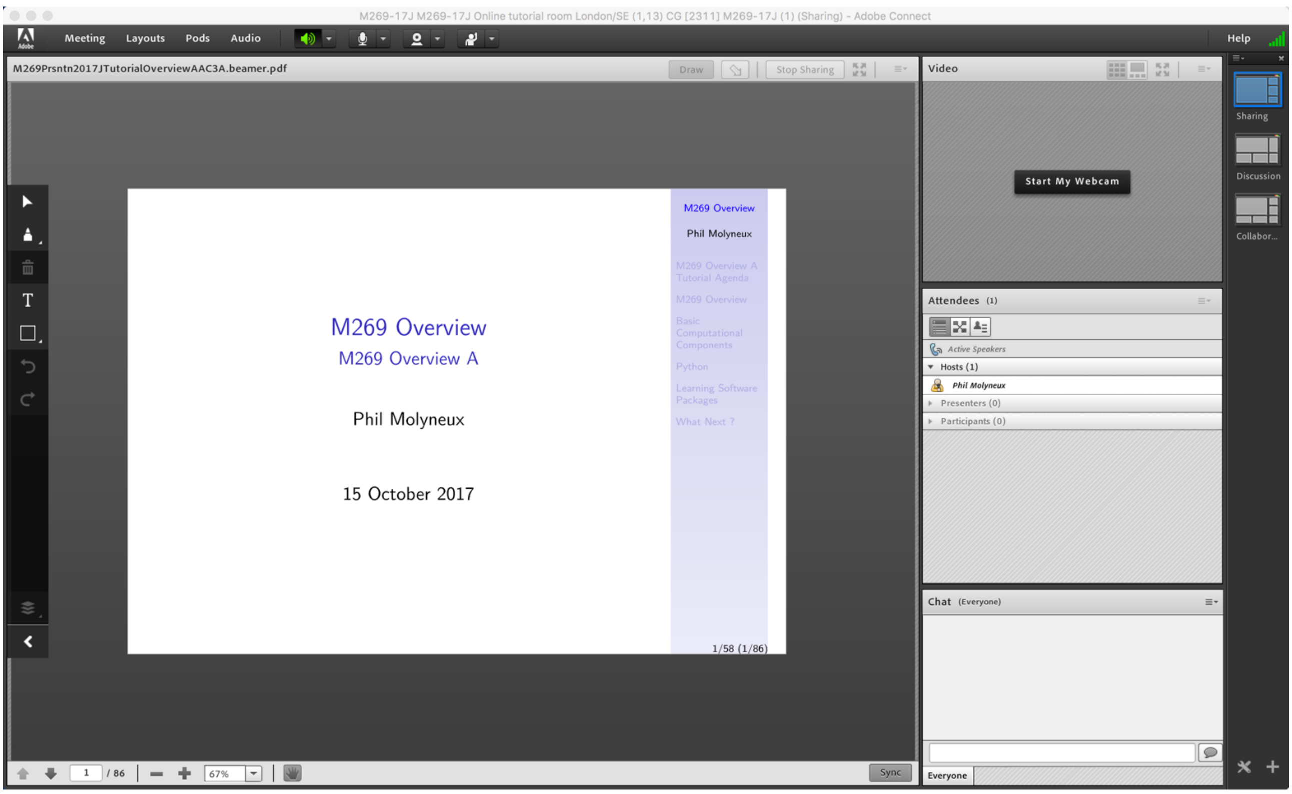
Adobe Connect Interface — Student View (default)



Adobe Connect Interface — Tutor Quick Reference



Adobe Connect Interface — Tutor View



2.4 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 blued 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

2.5 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.
- **Attendance Report** see course Web site for joining room

2.6 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

2.7 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 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 — name 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)

2.8 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

Go to Table of Contents

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

```
stmnt ; stmnt
```

- **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
```

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)

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

```python
def binarySearchIter(xs,val):
  lo = 0
  hi = len(xs) - 1

  while lo <= hi:
    mid = (lo + hi) // 2
    guess = xs[mid]

    if val == guess:
      return mid
    elif val < guess:
      hi = mid - 1
    else:
      lo = mid + 1

  return None
```

Binary Search Recursive

```python
def binarySearchRec(xs,val,lo=0,hi=-1):
  if (hi == -1):
    hi = len(xs) - 1

  mid = (lo + hi) // 2

  if hi < lo:
    return None
  else:
    guess = xs[mid]
    if val == guess:
      return mid
    elif val < guess:
      return binarySearchRec(xs,val,lo,mid-1)
    else:
      return binarySearchRec(xs,val,mid+1,hi)
```

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
```

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)

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.

4 Python

4.1 Learning Python

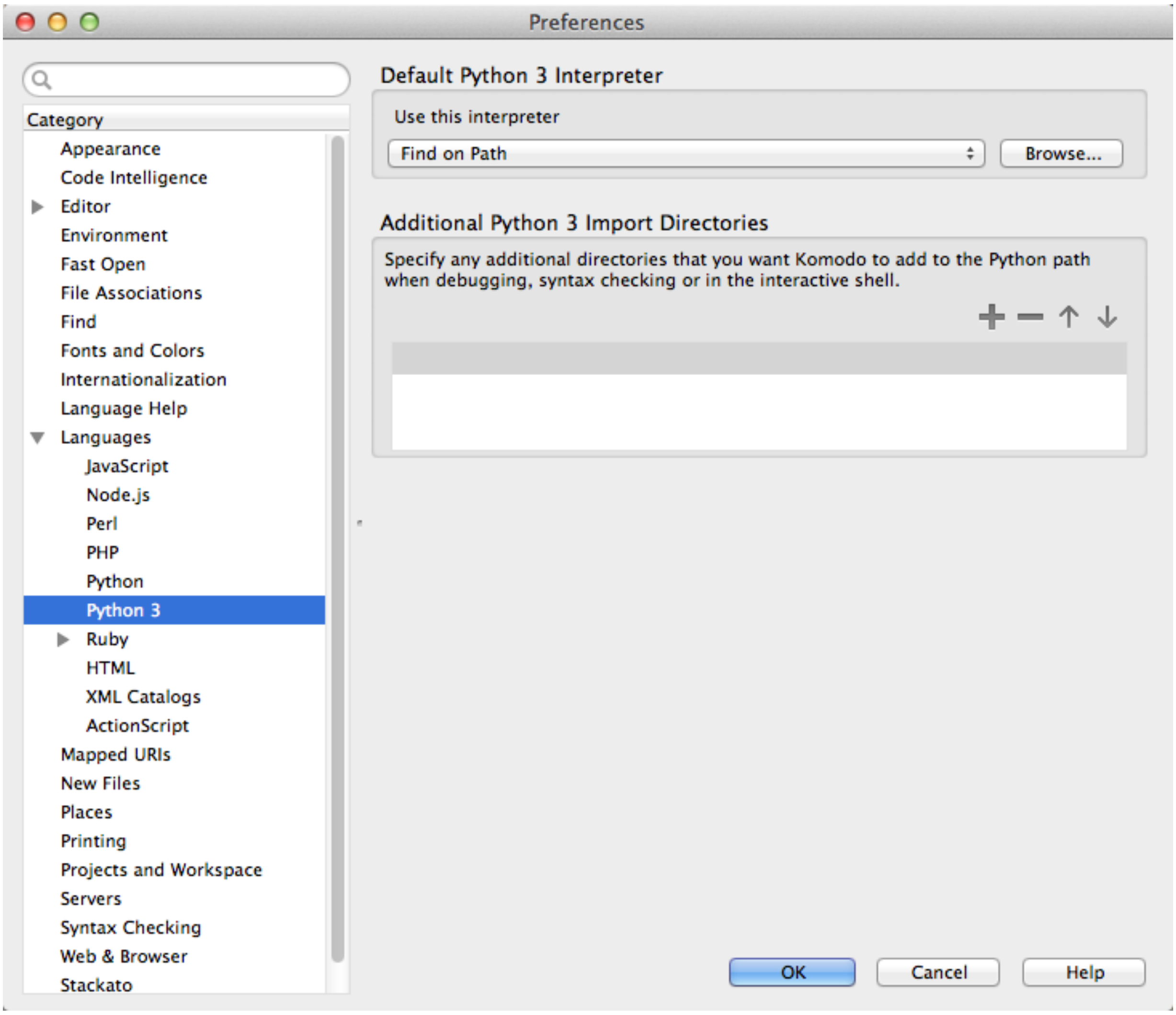
- Miller & Ranum Problem Solving with Algorithms and Data Structures using Python

- Python 3 Documentation
- Python Tutorial
- Python Language Reference
- Python Library Reference
- Hitchhiker's Guide to Python
- Stackoverflow on Python
- Dive into Python 3

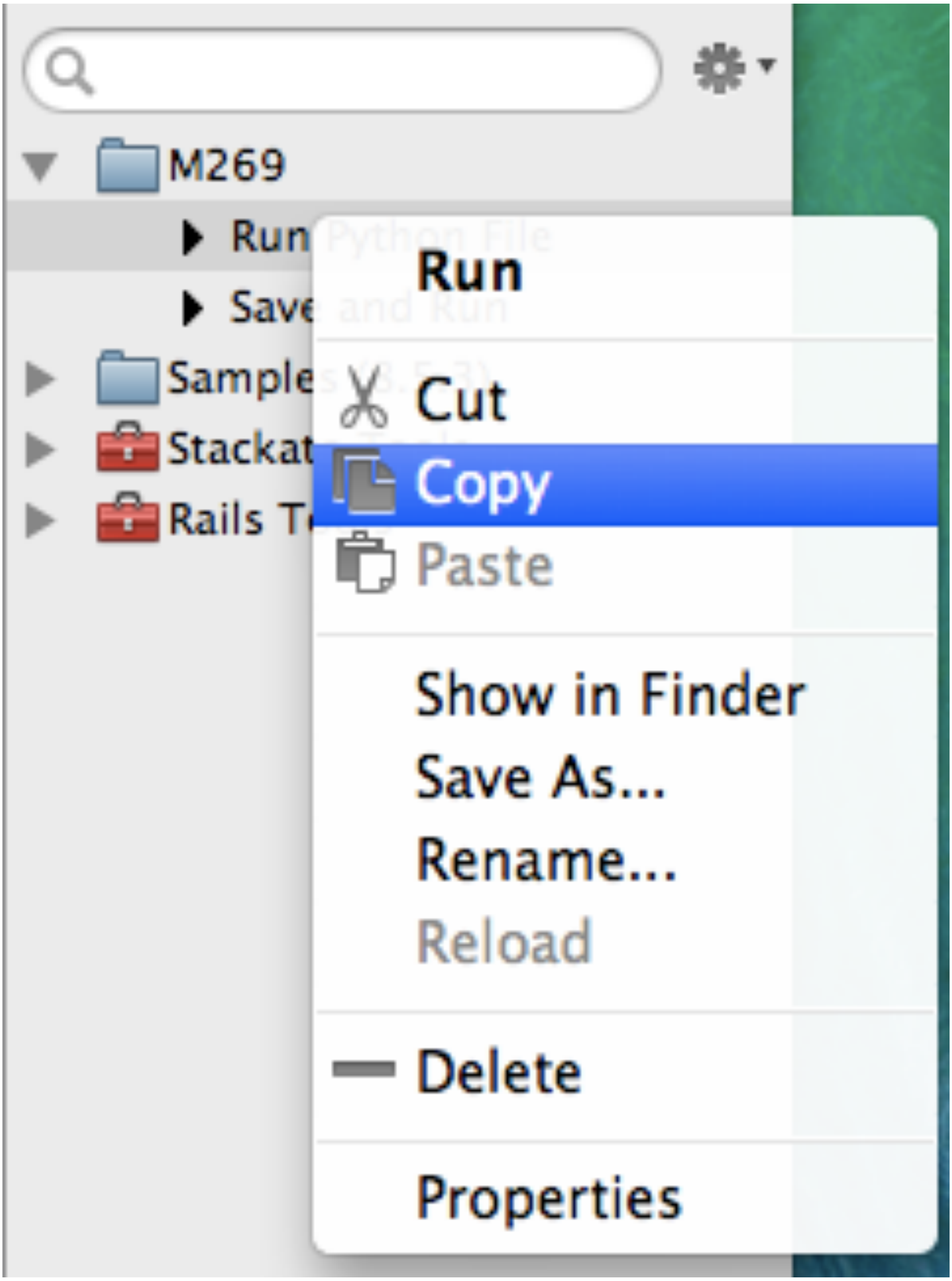
4.2 Setting up Python with Komodo

- Install *ActivePython* version 3.x from `http://www.activestate.com/activepython/downloads`
- *Mac OS X* Python 3 is at `/usr/local/bin/python3.3` which is a *symbolic link* to `/Library/Frameworks/Python.framework/Versions/3.3/bin/python3.3`
- *Mac OS X* idle 3 is at `/usr/local/bin/idle3.3` (exact versions will depend on install date)
- *Windows* install location %SystemDrive%\Python33 and in *Start* menu (if Windows 7)
- Documentation at `http://docs.activestate.com/activepython/3.3/`
- *Mac OS X* may need to install correct version of *Tcl/tk* for *IDLE* — `https://www.python.org/download/mac/tcltk`
- Install the *M269 Komodo macros*
  - See *M269 Software Installation*
  - Make sure the *Toolbox* and *Command output* tabs are visible — View > Tabs & Sidebars
  - Right-Click in *Toolbox* and select Add > New Folder... to create *M269* folder in *Toolbox*
  - Select *M269* folder, right-click and select Import/Export > Import Files from File System and select both files from the M269 macro download.
- Ensure Komodo is using Python 3
  - Preferences... > Languages Category > Python 3 and select your Python 3
  - In the *Toolbox* right-click *Run Python File* and select *Properties*

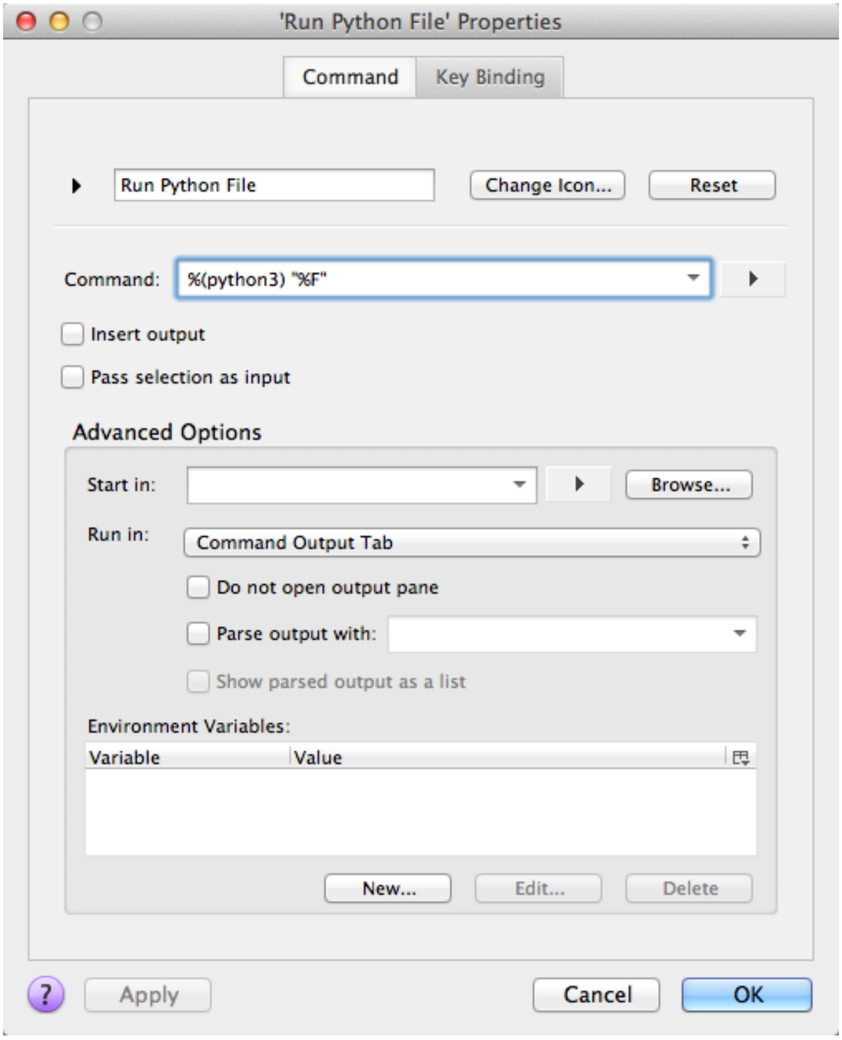
Komodo Preferences: Languages Python 3



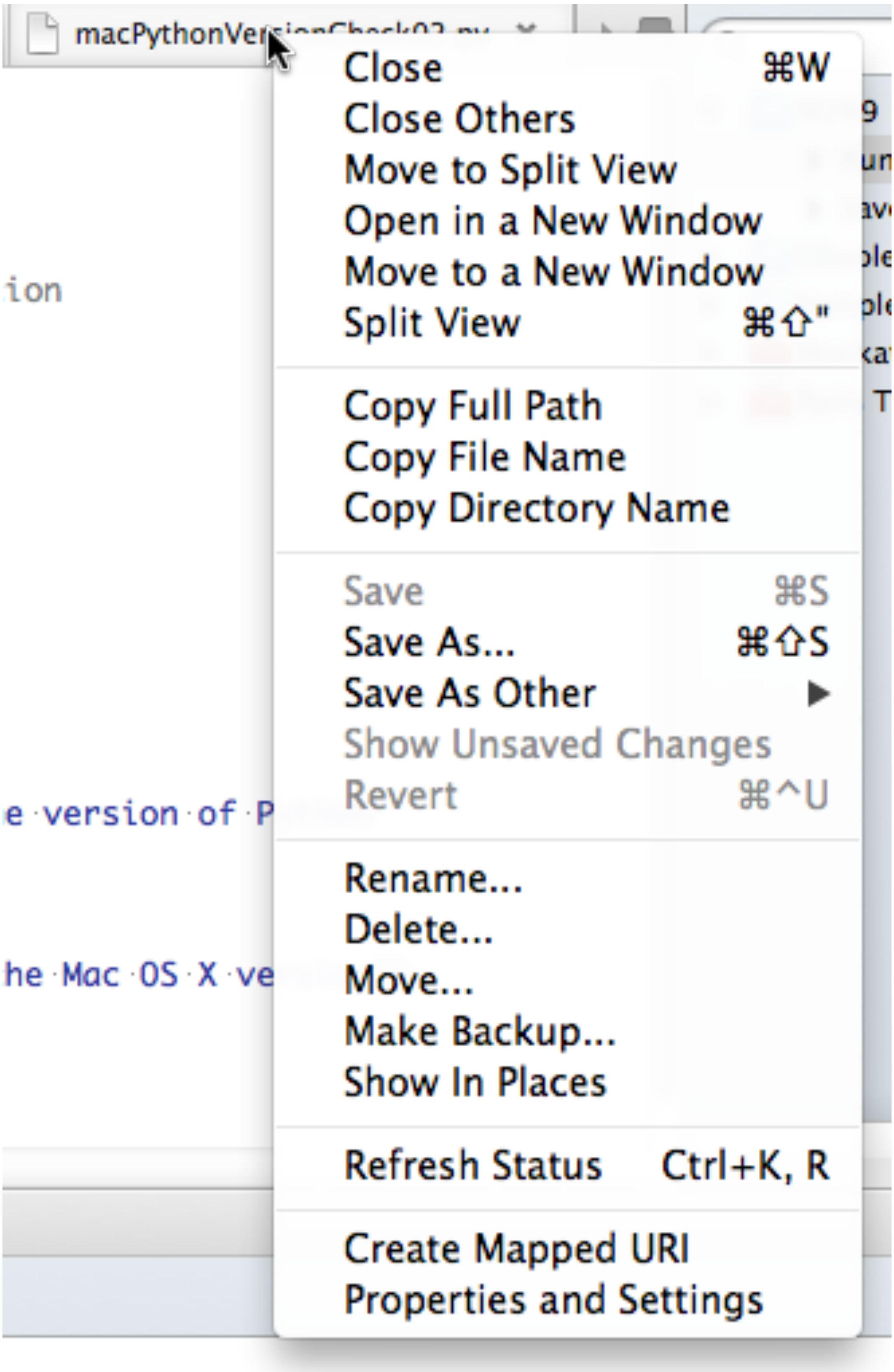
Komodo Run Command Context Menu



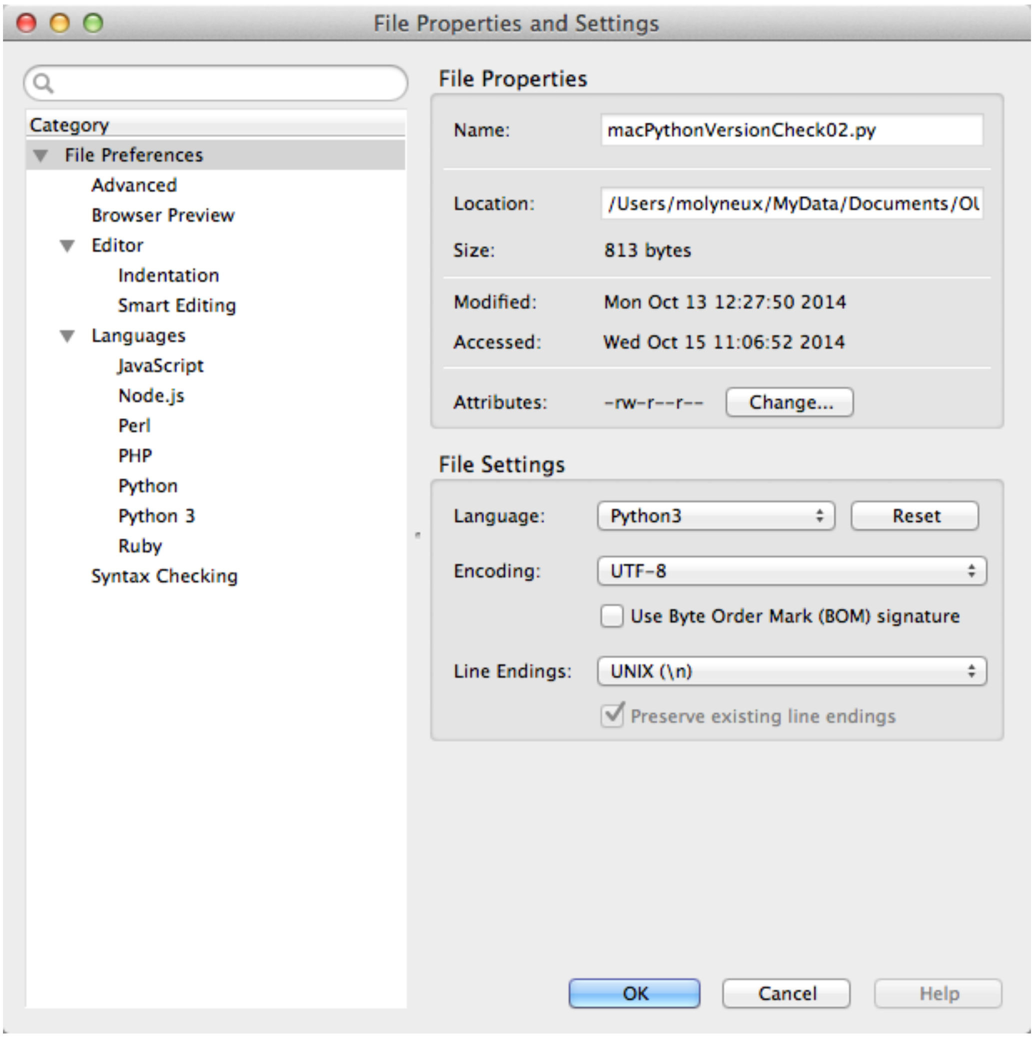
Komodo *Run Python File* Properties



Komodo File Tab Context Menu



Komodo File Properties and Settings



Indentation and Tabs

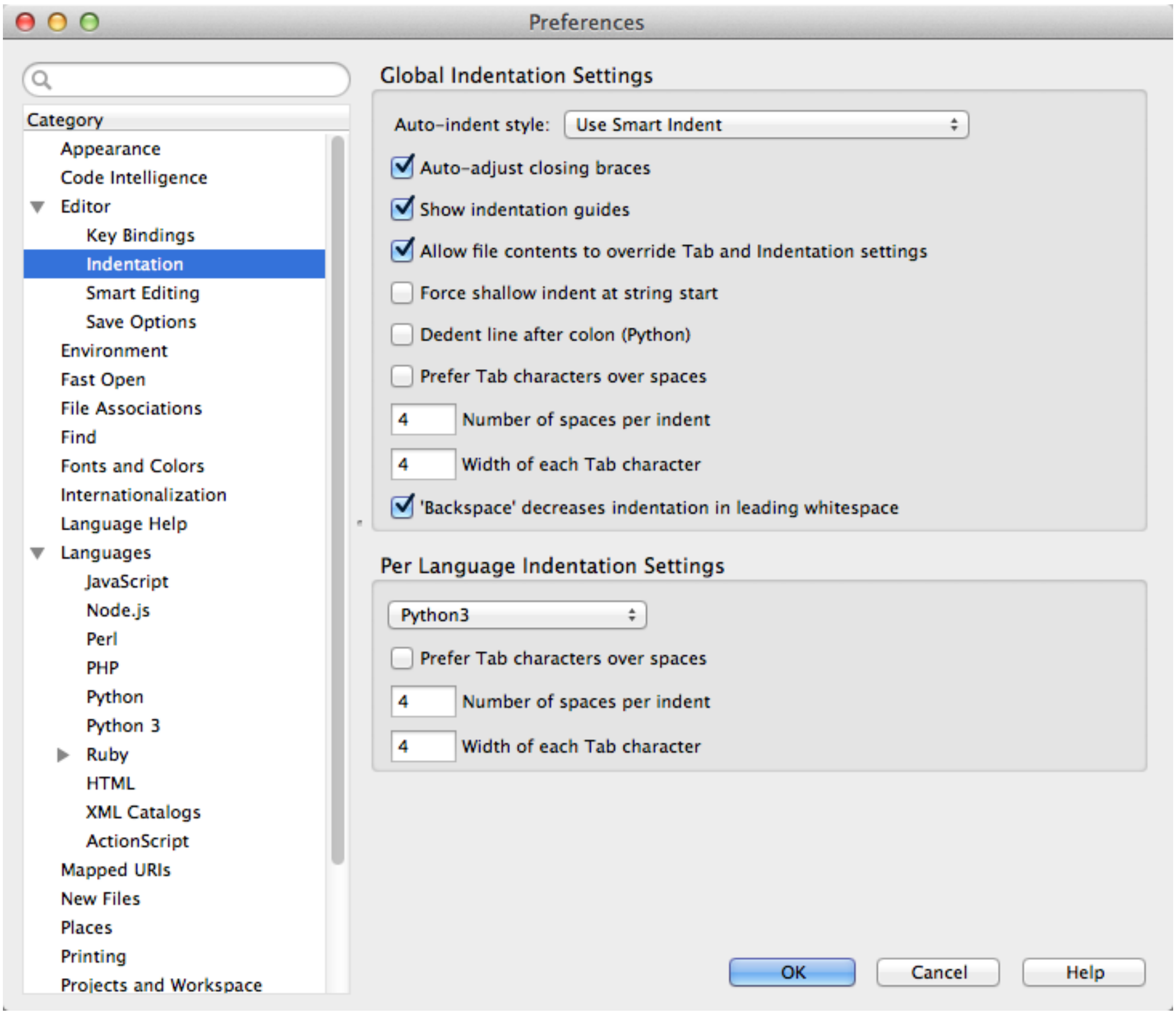
- How do you set spaces per indent to 2 or 4 ?
- How do you make the *Tab* key issue spaces ?
- Why is the *Tab* character *evil* ?
- How do you set spaces per indent to 2 or 4 ?

  Preferences... › Editor › Global Indentation Settings (default 8)
- How do you make the *Tab* key issue spaces ?

  Preferences... › Editor › Global Indentation Settings and uncheck *Prefer Tab characters over spaces*
- Why is the *Tab* character *evil* ?

  See Tabs vs Spaces, Tab key
- See Python Enhancement Proposals (PEP 8) — Style Guide for Python Code
- Mixing tabs and spaces can lead to inconsistent layout when copying from one editor to another or *MS Word*
- Tab character is Unicode U+0009 or ^I or HT or \t see C0 and C1 control codes

Komodo Preferences: Editor Indentation



4.3 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

```
xs[1]
xs[0]
xs[5]
ys
xs[1:3]
xs[::2]
xs[1:-1]
xs[-3]
xs[:]
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

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

4.4 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 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 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

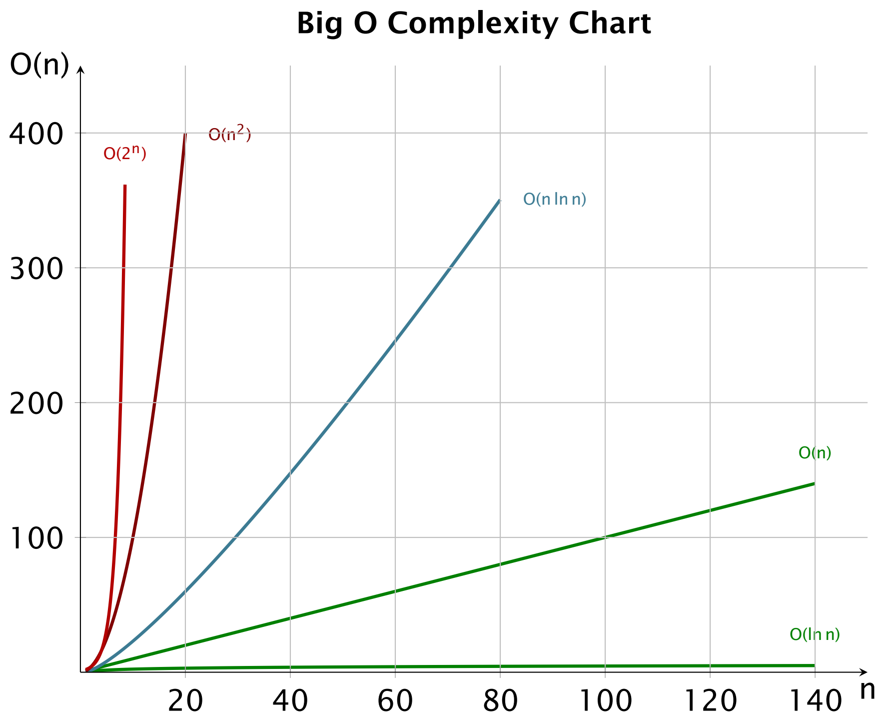
Orders of Common Functions

- $O(1)$ **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 FORTRAN$3.0n^3$ 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 $|h(x)| \leqslant C|g(x)|$ 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

  $k_1 g(x) \leqslant f(x) \leqslant k_2 g(x)$ 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

5.1 Complexity Example

```python
def someFunction(aList) :
  n = len(aList)
  best = 0
  for i in range(n) :
    for j in range(i + 1, n + 1) :
      s = sum(aList[i:j])
      best = max(best, s)
  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

```python
def maxSubSeg01(xs) :
  n = len(xs)
  maxSoFar = xs[0]
  for i in range(1,n) :
    for j in range(i + 1, n + 1) :
      s = sum(xs[i:j])
      maxSoFar = max(maxSoFar, s)
  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

  $$(n - 1) + (n - 2) + \ldots + 2 + 1 = \frac{(n - 1)}{2} \times n$$

- 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()`

```python
def maxSubSeg02(xs) :
  maxToHere = xs[0]
  maxSoFar  = xs[0]
  for x in xs[1:] :
    # Invariant: maxToHere, maxSoFar OK for xs[0..(i-1)]
    maxToHere = max(x, maxToHere + x)
    maxSoFar  = max(maxSoFar, maxToHere)
  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

```python
def maxSubSeg03(xs) :
  maxSoFar = maxToHere = xs[0]
  startIdx, endIdx, startMaxToHere = 0, 0, 0
  for i, x in enumerate(xs) :
    if maxToHere + x < x :
      maxToHere = x
      startMaxToHere = i
    else :
      maxToHere = maxToHere + x

    if maxSoFar < maxToHere :
      maxSoFar = maxToHere
      startIdx, endIdx = startMaxToHere, i

  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

```python
egList = [-2,1,-3,4,-1,2,1,-5,4]

egList01 = [-1,-1,-1]

egList02 = [1,2,3]

assert maxSubSeg03(egList) == (6, [4, -1, 2, 1], 3, 6)

assert maxSubSeg03(egList01) == (-1, [-1], 0, 0)

assert maxSubSeg03(egList02) == (7, [1, 2, 3], 0, 2)
```

5.2 Complexity & Python Data Types

Lists

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

```python
class Bag:

  def __init__(self):
    self.list = []

  def add(self, item):
    self.list.append(item)

  def remove(self, item):
    self.list.remove(item)

  def contains(self, item):
    return item in self.list

  def count(self, item):
    return self.list.count(item)
```

```python
  def size(self):
    return len(self.list)

  def __str__(self):
    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(`total`/(1 + `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 $[r_0, r_1, \ldots, r_{n-1}]$ such that r_i = `tf`(`term`, d_i)$\times$`idf`(`term`, `documents`)

6 Exponentials and Logarithms

6.1 Exponentials and Logarithms — Definitions

- **Exponential function** $y = a^x$ or $f(x) = a^x$
- $a^n = a \times a \times \cdots \times a$ (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 (Chanbers Dictionary 13th edition 2014)

6.2 Rules of Indices

1. $a^m \times a^n = a^{m+n}$
2. $a^m \div a^n = a^{m-n}$
3. $a^{-m} = \dfrac{1}{a^m}$
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 $a \neq 0$

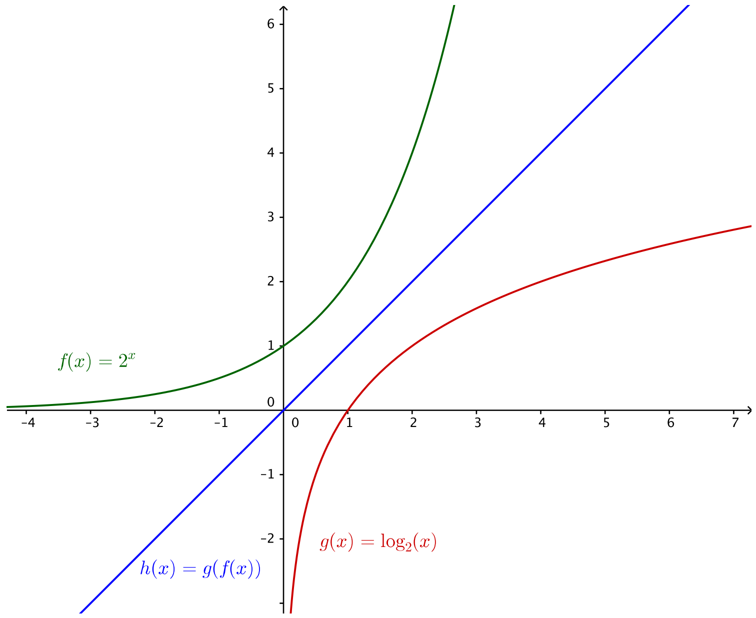
- **Exercise** Justify the above rules
- What should 0^0 evaluate to ?
- See Wikipedia: Exponentiation
- The *justification* above probably only worked for whole or *rational* numbers — see later for exponents with *real* numbers (and the value of *logarithms*, *calculus*...)

6.3 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...

6.4 Exponentials and Logarithms — Graphs

- See GeoGebra file `expLog.ggb`



6.5 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**

$$
\begin{aligned}
x &= a^{\log_a x} \\
y &= a^{\log_a y} && \text{by definition of log} \\
xy &= a^{\log_a x} a^{\log_a y} \\
&= a^{\log_a x + \log_a y} && \text{by laws of indices} \\
\text{Hence } \log_a xy &= \log_a x + \log_a y && \text{by definition of log}
\end{aligned}
$$

6.6 Arithmetic and Inverses

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

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

Arithmetic Operations — Commutativity and Associativity

- **Commutativity** $x \circledast y = y \circledast x$
- **Associativity** $(x \circledast y) \circledast z = x \circledast (y \circledast z)$
- $(+)$ and $(\times)$ are *semantically* commutative and associative — so we can leave the brackets out
- $(-)$ and $(\div)$ 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 $(\div)$ 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

6.7 Change of Base

- **Change of base**

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

Proof: Let $y = \log_a x$

$$\begin{aligned} a^y &= x \\ \log_b a^y &= \log_b x \\ y \log_b a &= \log_b x \\ y &= \frac{\log_b x}{\log_b a} \end{aligned}$$

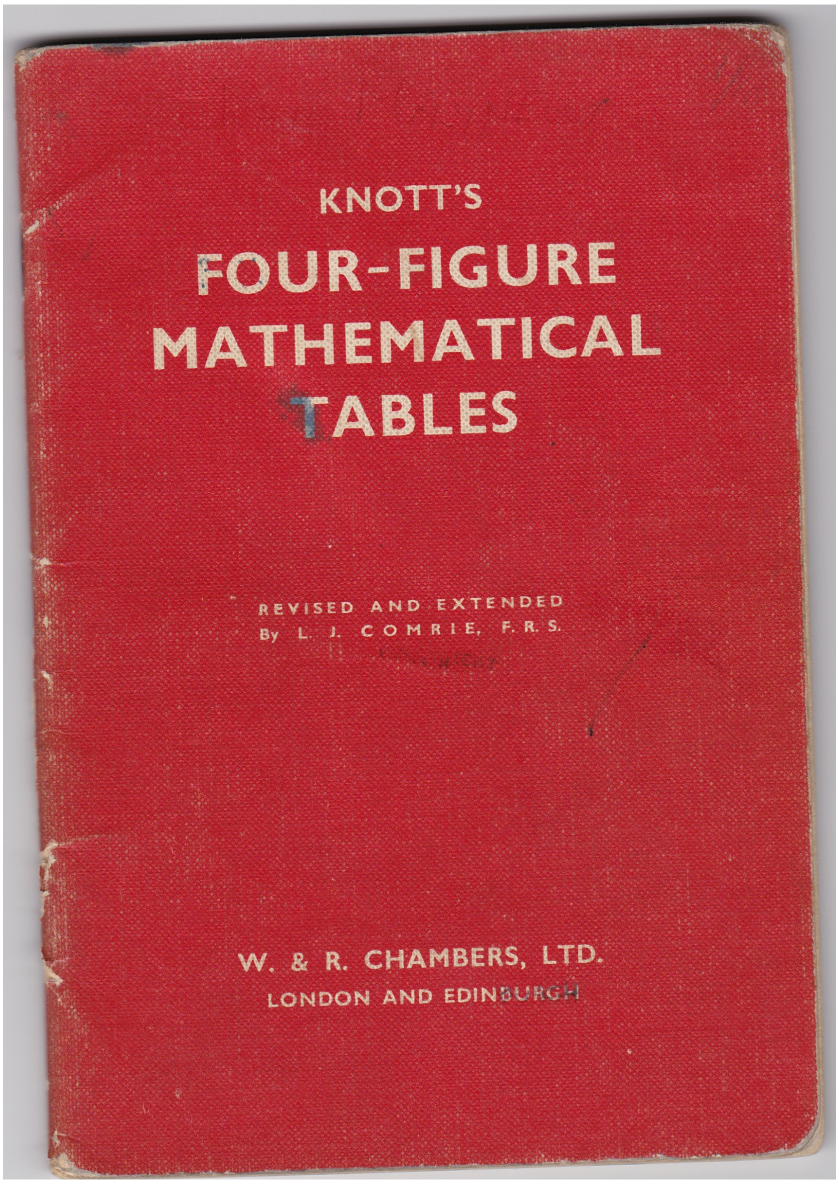
- Given x, $\log_b x$, find the base b
  - $b = x^{\frac{1}{\log_b x}}$
- $\log_a b = \dfrac{1}{\log_b a}$

7 Before Calculators and Computers

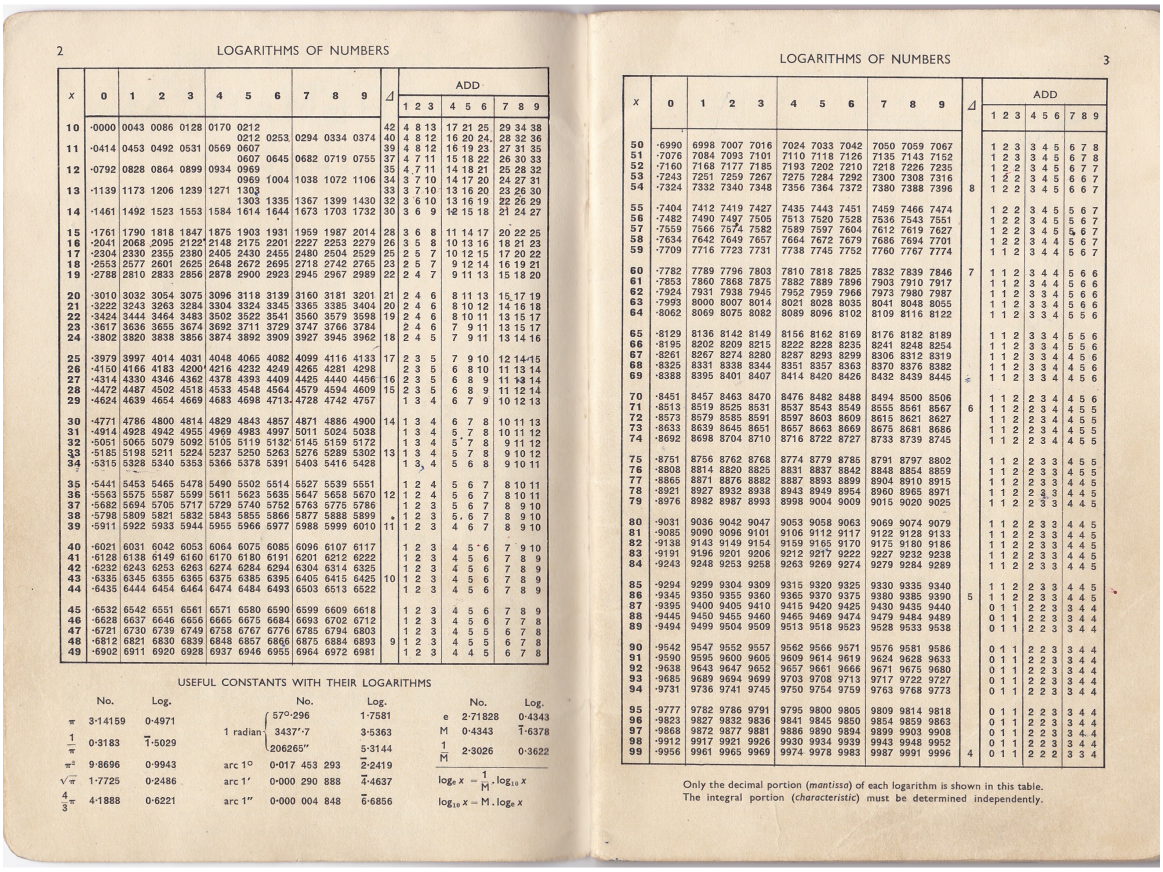
- We had computers before 1950 — they were *humans* with pencil, paper and some further aids:
- **Slide rule** invented by William Oughtred in the 1620s — major calculating tool until pocket calculators in 1970s
- **Log tables** in use from early 1600s — method of logarithms propounded by John Napier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

7.1 Log Tables

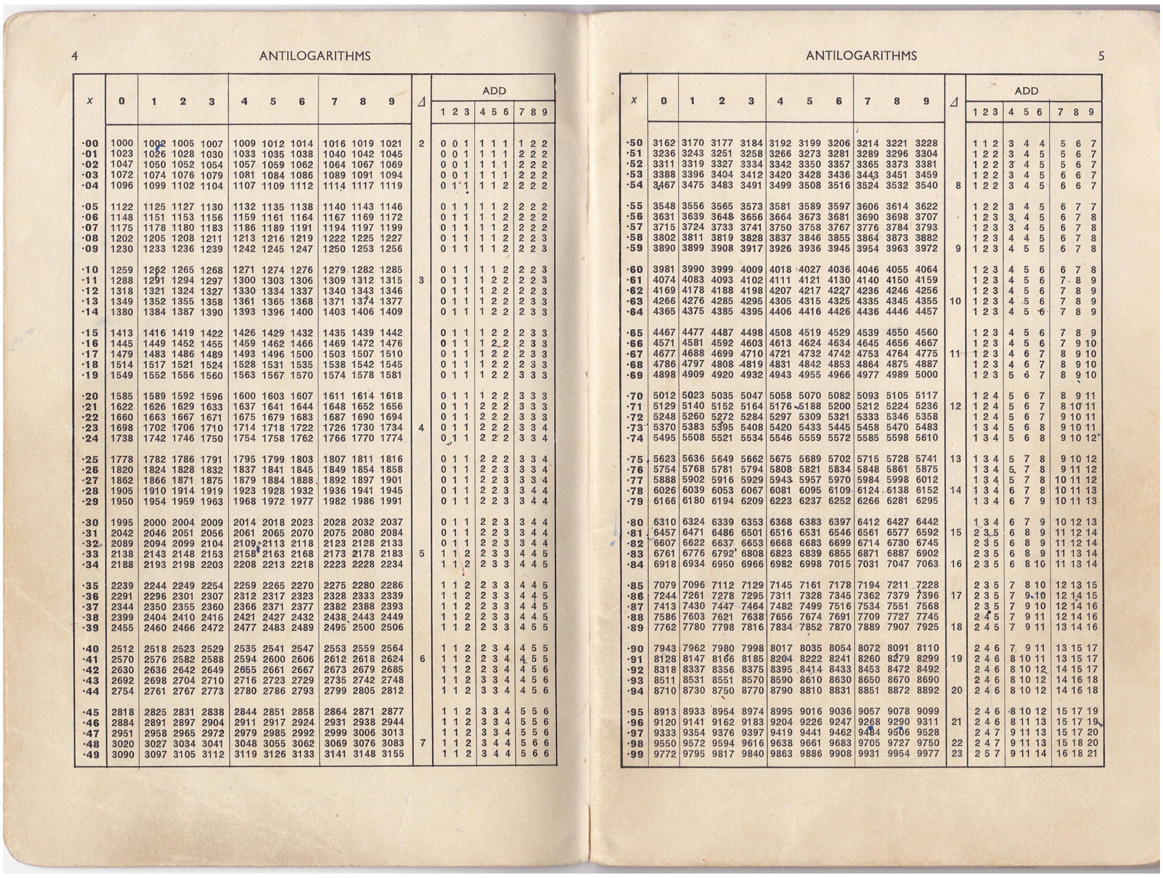
Knott's Four-Figure Mathematical Tables



Logarithms of Numbers

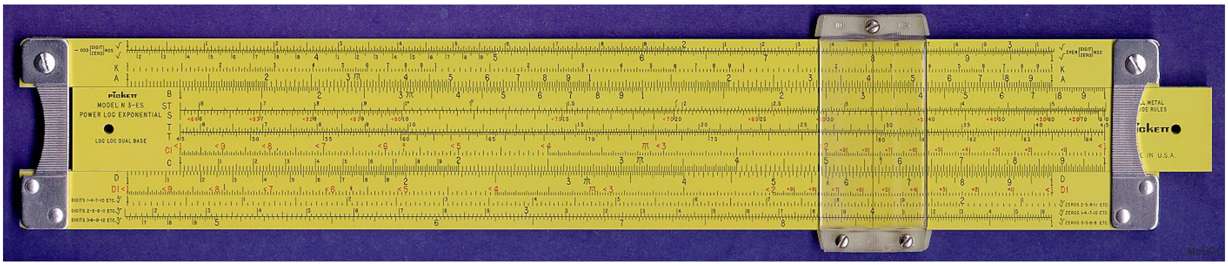


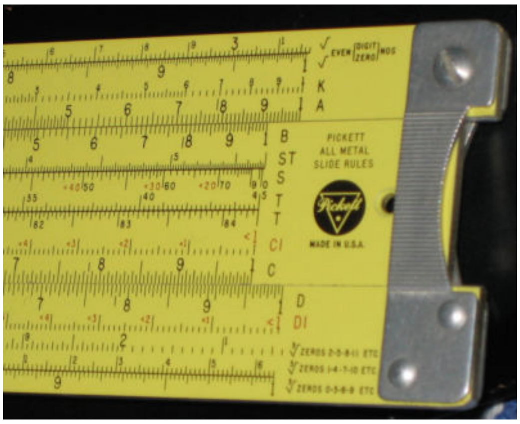
Antilogarithms



7.2 Slide Rules

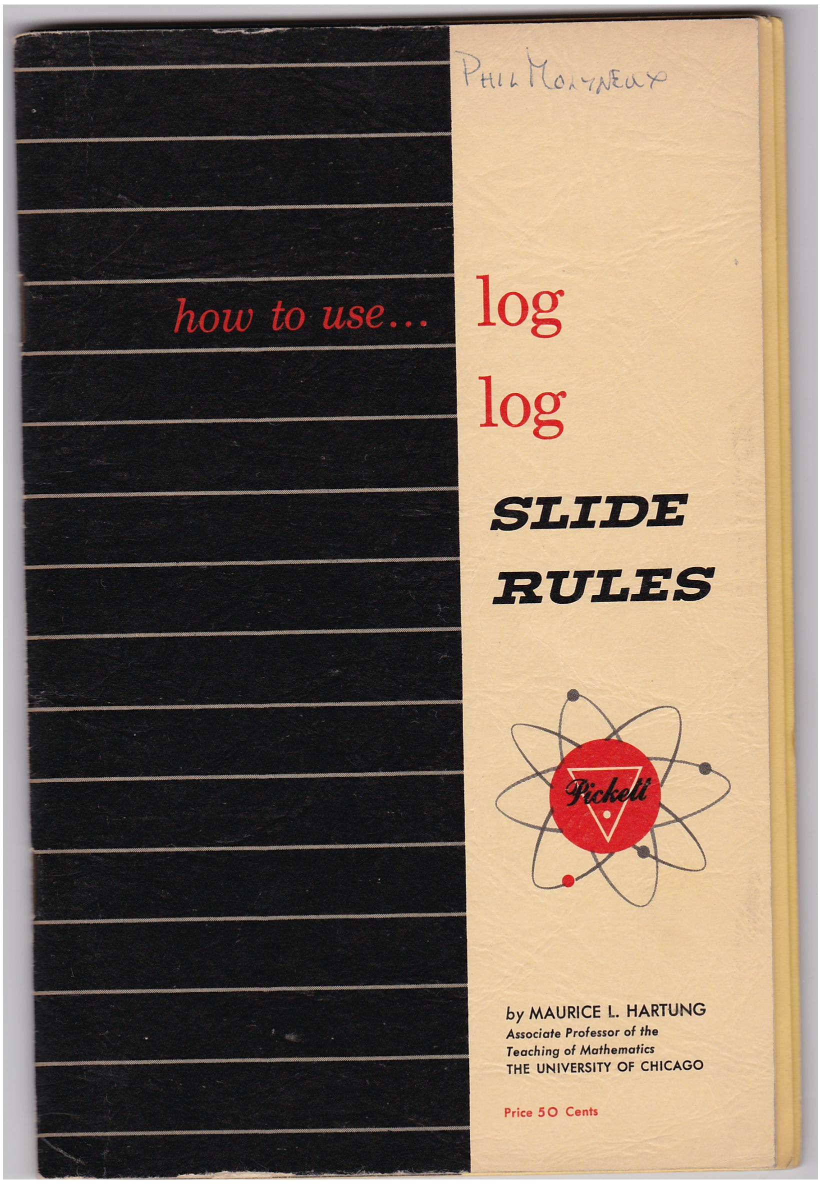
Pickett N 3-ES from 1967





- See Oughtred Society
- UKSRC
- Rod Lovett's Slide Rules
- Slide Rule Museum

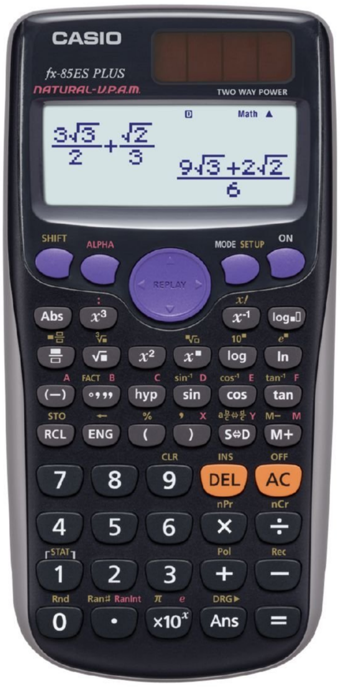
Pickett log log **Slide Rules Manual 1953**



7.3 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

7.4 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 $\boxed{\text{log}}$ = 1.9528 and 597 $\boxed{\text{log}}$ = 2.7760
- $\boxed{+}$ displays 4.7288
- 10 $\boxed{\text{ENTER}}$, $\boxed{x \rightleftarrows y}$ and $\boxed{y^x}$ displays 53550.9000
- **Casio fx-85GT PLUS**
- $\boxed{\text{log}}$ 89.7 $\boxed{)}$ = 1.952792443 $\boxed{+}$ $\boxed{\text{log}}$ 597 $\boxed{)}$ = 2.775974331 $\boxed{=}$
- 4.728766774 $\boxed{\text{Ans}}$+$\boxed{10^x}$ gives 53550.9

8 Logic and Truth Tables

8.1 Boolean Expressions and Truth Tables

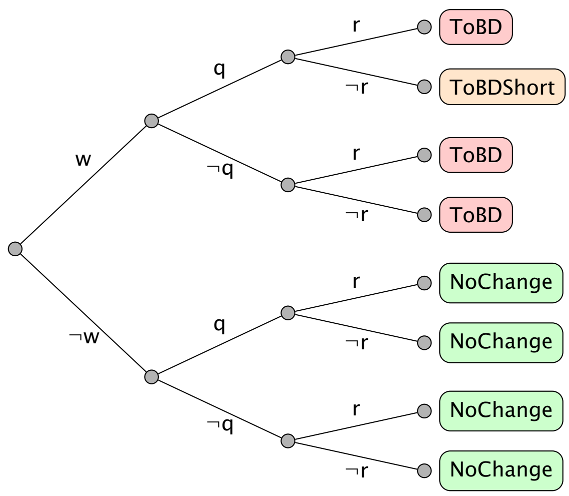
Traffic Lights Example

- 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

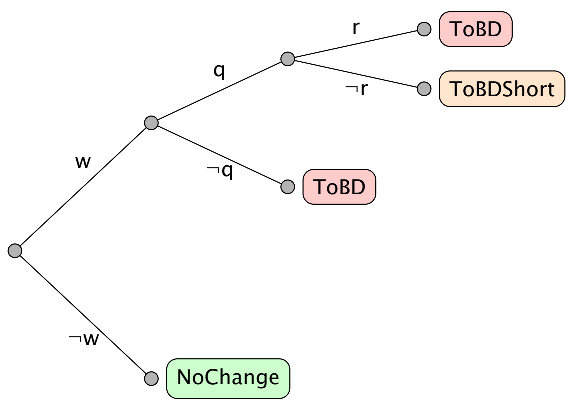
- Given the following events:
  - ToBD Change flow to BD
  - ToBDShort Change flow to BD for short time
  - NoChange No Change to lights
- Express above as truth table, outcome tree, boolean expression
- **Traffic Lights outcome table**

w	q	r	Event
T	T	T	ToBD
T	T	F	ToBDSort
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

```
def trafficLights01(w,q,r) :
  """
```

```
  Input 3 Booleans
  Return Event string
  """
  if w :
    if q :
      if r :
        evnt = "ToBD"
      else :
        evnt = "ToBDShort"
    else :
      evnt = "ToBD"
  else :
    evnt = "NoChange"
  return evnt
```

- **Traffic Lights test code 01**

```
trafficLights01Evnts = [((w,q,r), trafficLights01(w,q,r))
                              for w in [True,False]
                              for q in [True,False]
                              for r in [True,False]]

assert trafficLights01Evnts \
  == [((True, True, True), 'ToBD')
       ,((True, True, False), 'ToBDShort')
       ,((True, False, True), 'ToBD')
       ,((True, False, False), 'ToBD')
       ,((False, True, True), 'NoChange')
       ,((False, True, False), 'NoChange')
       ,((False, False, True), 'NoChange')
       ,((False, False, False), 'NoChange')]
```

- **Traffic Lights code 02** compound Boolean conditions

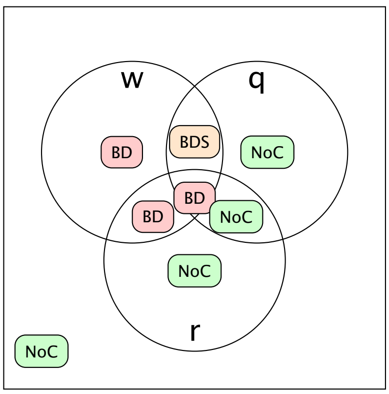
```
def trafficLights02(w,q,r) :
  """
  Input 3 Booleans
  Return Event string
  """
  if ((w and q and r) or (w and not q)) :
    evnt = "ToBD"
  elif (w and q and not r) :
    evnt = "ToBDShort"
  else :
    evnt = "NoChange"
  return evnt
```

- What objectives do we have for our code ?

- **Traffic Lights test code 02**

```
trafficLights02Evnts = [((w,q,r), trafficLights02(w,q,r))
                              for w in [True,False]
                              for q in [True,False]
                              for r in [True,False]]

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

8.2 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 the programming language processor to warn us otherwise — not always possible

8.3 Boolean Expressions Exercise

Rail Ticket Exercise

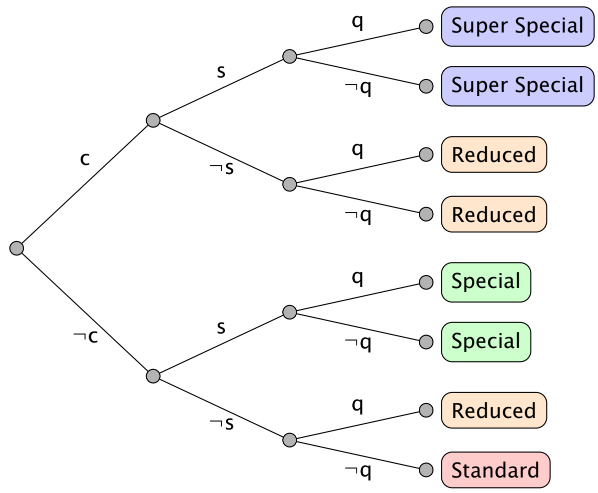
- Rail ticket discounts for:
  - c Rail card
  - q Off-peak time
  - s Special offer
- 4 fares: Standard, Reduced, Special, Super Special
- Rules:
  1. Reduced fare if rail card or at off-peak time
  2. Without rail card no reduction for both special offer and off-peak.
  3. Rail card always has reduced 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

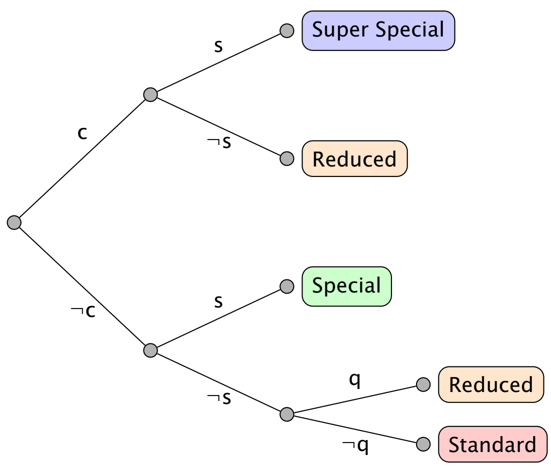
- Rail ticket outcome table
- Note that it may be more convenient to change columns

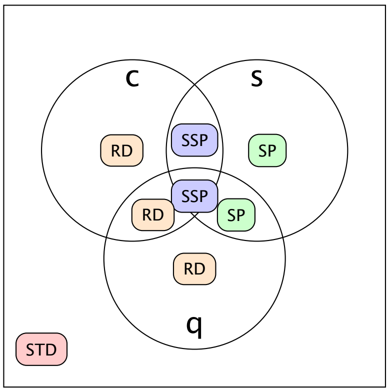
c	s	q	Event
T	T	T	Super Special
T	T	F	Super Special
T	F	T	Reduced
T	F	F	Reduced
F	T	T	Special
F	T	F	Special
F	F	T	Reduced
F	F	F	Standard

- 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**

```python
def railTicket01(c,s,q) :
  """
  Input 3 Booleans
  Return Event string
  """
  if c :
    if s :
      evnt = "SSP"
    else :
      evnt = "RD"
  else :
    if s :
      evnt = "SP"
    else :
      if q :
        evnt = "RD"
      else :
        evnt = "STD"
  return evnt
```

- **Rail Ticket test code 01**

```python
railTicket01Evnts = [((c,s,q), railTicket01(c,s,q))
                              for c in [True,False]
                              for s in [True,False]
                              for q in [True,False]]

assert railTicket01Evnts \
  == [((True, True, True), 'SSP')
       ,((True, True, False), 'SSP')
       ,((True, False, True), 'RD')
       ,((True, False, False), 'RD')
       ,((False, True, True), 'SP')
       ,((False, True, False), 'SP')
       ,((False, False, True), 'RD')
       ,((False, False, False), 'STD')]
```

- **Rail Ticket code 02** compound Boolean expressions

```python
def railTicket02(c,s,q) :
  """
  Input 3 Booleans
  Return Event string
  """
  if (c and s) :
    evnt = "SSP"
  elif ((c and not s) or (not c and not s and q)) :
    evnt = "RD"
  elif (not c and s) :
    evnt = "SP"
  else :
    evnt = "STD"
  return evnt
```

- **Rail Ticket test code 02**

```
railTicket02Evnts = [((c,s,q), railTicket02(c,s,q))
                      for c in [True,False]
                      for s in [True,False]
                      for q in [True,False]]

assert railTicket02Evnts == railTicket01Evnts
```

8.4 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 $(\mathbb{B}, \mathbb{B}) \to \mathbb{B}$ — 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 ⇒ T == T and T ⇒ F == F
- Hence 4 possibilities for truth table

p	**q**	**p ⇒ q**	**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

- (⇒) 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 $\mathbf{p \bar{\wedge} q}$, $\mathbf{p \uparrow q}$, Sheffer stroke
- NOR $\mathbf{p \bar{\vee} q}$, $\mathbf{p \downarrow q}$, 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.

8.5 Truth Function

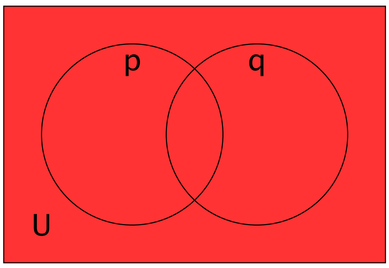
- The following appendix notes illustrate the 16 binary functions of two Boolean variables
- See Truth function
- See Functional completeness
- See Sheffer stroke
- See Logical NOR

Table of Binary Truth Functions

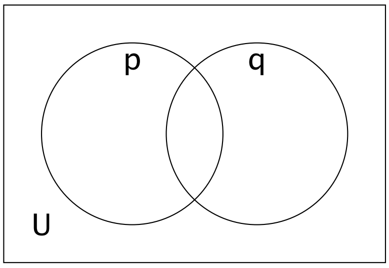
p	**q**	$\top$	$p \vee q$	$p \Leftarrow q$	p	$p \Rightarrow q$	q	$p \Leftrightarrow q$	$p \wedge q$
T	T	T	T	T	T	T	T	T	T
T	F	T	T	T	T	F	F	F	F
F	T	T	T	F	F	T	T	F	F
F	F	T	F	T	F	T	F	T	F

p	**q**	$\bot$	$p \overline{\vee} q$	$p \not\Leftarrow q$	$\neg p$	$p \not\Rightarrow q$	$\neg q$	$p \not\Leftrightarrow q$	$p \overline{\wedge} q$
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	F	T	F	T

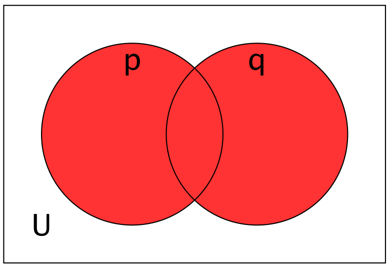
- **Tautology** True, $\top$, *Top*



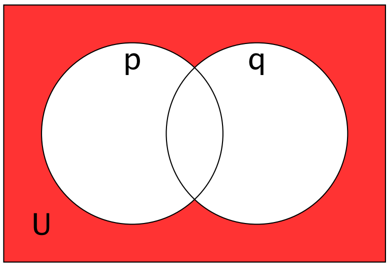
- **Contradiction** False, $\bot$, *Bottom*



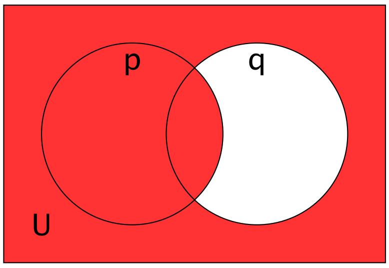
- **Disjunction** OR, $p \vee q$



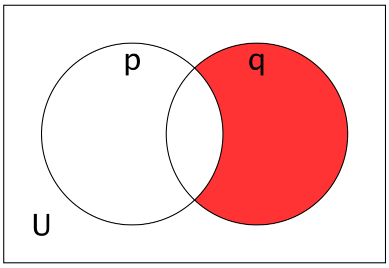
- **Joint Denial** NOR, $p \overline{\vee} q$, $p \downarrow q$, *Pierce's arrow*



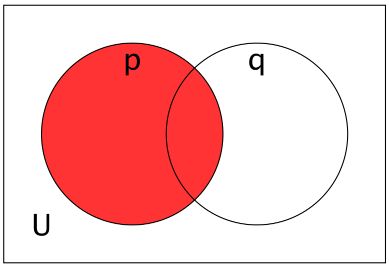
- **Converse Implication** $p \Leftarrow q$



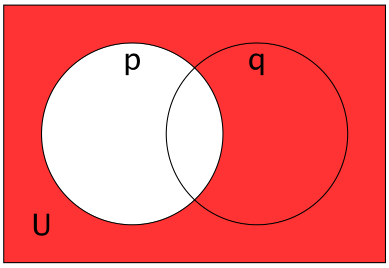
- **Converse Nonimplication** $p \nLeftarrow q$



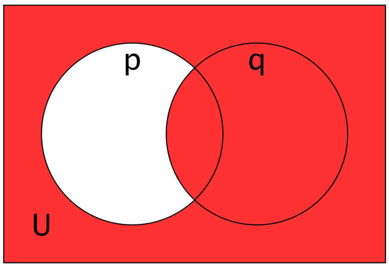
- **Proposition** p



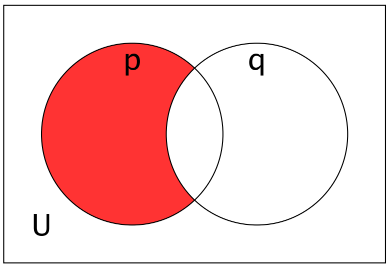
- **Negation of** p



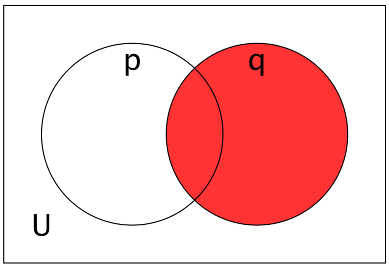
- **Material Implication** $p \Rightarrow q$



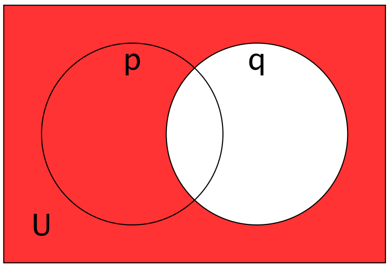
- **Material Nonimplication** $p \not\Rightarrow q$



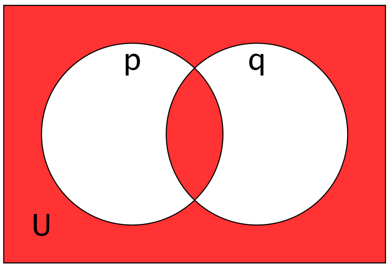
- **Proposition** q q



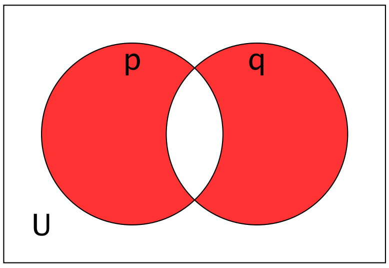
- **Negation of** q $\neg q$



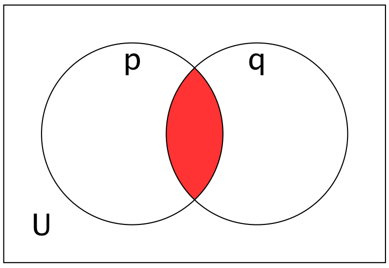
- **Biconditional** If and only if, IFF, $p \Leftrightarrow q$



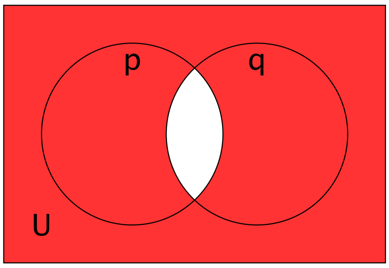
- **Exclusive disjunction** XOR, $p \nLeftrightarrow q$



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



- **Alternative denial** NAND, $p \barwedge q$, $p \uparrow q$, *Sheffer stroke*



9 Abstract Data Types

9.1 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 `Circle`s, `Square`s, ... 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)

```haskell
data Shape
  = Circle Point Radius
   | Square Point Size

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

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

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

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

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

```haskell
class IsShape shape where
  draw :: shape -> Pict
  moveTo :: Point -> shape -> shape

data Shape = forall a . (IsShape a) => Shape a

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

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

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

shapes01 :: [Shape]
shapes01 = map (moveShapeTo (2,2)) shapes
           where
           moveShapeTo p (Shape s) = Shape (moveTo p s)
```

9.2 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 permissable 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

```
from collections import namedtuple

Qp1 = namedtuple('Qp1',['frs','rbks'])
```

- **Queue Implementation 1** main operations

```
def makeEmptyQp1():
  return Qp1([],[])

def isEmptyQp1(q):
  return q.frs == []

def addToQp1(q,x):
  return checkQp1(q.frs, [x] + q.rbks[:])

def headOfQp1(q):
  if q.frs == [] :
    RunTimeError("headOfQp1 applied to empty queue")
  else:
    return q.frs[0]

def tailOfQp1(q):
  if q.frs == [] :
    RunTimeError("tailOfQp1 applied to empty queue")
  else:
    return checkQp1(q.frs[1:], q.rbks[:])
```

- **Queue Implementation 1** `checkOp1()`

```
def checkQp1(frs, rbks):
  if frs == [] :
    bks = rbks[:]
    bks.reverse()
    return Qp1(bks, [])
  else :
    return Qp1(frs, rbks)
```

- Note copying of arguments — see below for reason
- Note also in `stringQp1Items` below at line 47 on page 53
- 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

```
def stringQp1(q) :
  return ("<" + stringQp1Items(q) + ">")

def stringQp1Items(q) :
  if isEmptyQp1(q) :
    return ""
  elif isEmptyQp1(tailOfQp1(q)) :
    return str(headOfQp1(q))
  else :
    return ( str(headOfQp1(q))
            + ", " + stringQp1Items(tailOfQp1(q)) )

def buildQp1(xs,q) :
  if xs == [] :
    return q
  else :
    return buildQp1(xs[1:],addToQp1(q,xs[0]))

def listToQp1(xs) :
  return buildQp1(xs, makeEmptyQp1())
```

- **Queue Implementation 1** test code

```
q11 = listToQp1([1,2,3,1])

q12 = tailOfQp1(q11)

assert q11 == Qp1(frs=[1], rbks=[1, 3, 2])

assert stringQp1(q11) == '<1, 2, 3, 1>'

assert q12 == Qp1(frs=[2, 3, 1], rbks=[])

assert stringQp1(q12) == '<2, 3, 1>'
```

- **Queue Implementation 2**
- Modify to add `size`
- Store in tuple to save calculating each time

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

- **Exercise** Add `size()` operation and other modifications
- **Queue Implementation 2** main operations

```
def makeEmptyQp2():
  return Qp2([],[], 0)

def isEmptyQp2(q):
  return q.frs == []
```

```
def addToQp2(q,x):
  return checkQp2(q.frs, [x] + q.rbks[:], q.sz + 1)

def headOfQp2(q):
  if q.frs == [] :
    RunTimeError("headOfQp2␣applied␣to␣empty␣queue")
  else:
    return q.frs[0]

def tailOfQp2(q):
  if q.frs == [] :
    RunTimeError("tailOfQp2␣applied␣to␣empty␣queue")
  else:
    return checkQp2(q.frs[1:], q.rbks[:], q.sz - 1)
```

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

```
def sizeOfQp2(q) :
  return q.sz

def checkQp2(frs, rbks, sz):
  if frs == [] :
    bks = rbks[:]
    bks.reverse()
    return Qp2(bks, [], sz)
  else :
    return Qp2(frs, rbks, sz)
```

- Note also in stringQp2Items below at line 118 on page 54
- 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) + ">")
111
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)) )
120
121 def buildQp2(xs,q) :
122   if xs == [] :
123     return q
124   else :
125     return buildQp2(xs[1:],addToQp2(q,xs[0]))
126
127 def listToQp2(xs) :
128   return buildQp2(xs, makeEmptyQp2())
```

- **Queue Implementation 2** test code

```
q21 = listToQp2([1,2,3,1])

q22 = tailOfQp2(q21)

assert q21 == Qp2(frs=[1], rbks=[1, 3, 2], sz=4)

assert stringQp2(q21) == '<1,␣2,␣3,␣1>'

assert q22 == Qp2(frs=[2, 3, 1], rbks=[], sz=3)

assert stringQp2(q22) == '<2,␣3,␣1>'
```

10 Future Work

- Units 3 and 4
- 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
- Unit 3 Sorting
- Unit 4 Searching
- Both use binary tree structure
- See Sorting Tutorial
- See Binary Trees Tutorial
- 2 December 2020 TMA01
- Saturday 5 December 2020 Tutorial Online 10:00–12:00
- 2 January 2021 Unit 4
- 6 January 2021 iCMA43
- 30 January 2021 Unit 5
- 3 February 2021 iCMA44
- Sunday 28 February 2021 Tutorial Online 10:00–11:00
- 6 March 2021 Unit 6
- 10 March 2021 iCMA45
- Saturday 13 March 2021 Tutorial Online 10:00–13:00

11 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)

11.1 Binary Search — Haskell — version 1

Binary Search — Haskell — 1 (a)

```
module M269BinarySearch where

  import Data.Array
  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 = length 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)

```
binarySearch01 :: Ord a
  => [a] -> a -> (Int, Int) -> Maybe Int

binarySearch01 xs val (lo,hi)
  = if  hi < lo then Nothing
    else
      let mid = (lo + hi) ‘div‘ 2
          guess = xs !! mid
      in
      if val == guess
        then Just mid
      else if val < guess
        then binarySearch01 xs val (lo,mid-1)
      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

11.2 Binary Search — Haskell — version 2

Binary Search — Haskell — 2 (a)

```
binarySearchGen :: Integral a
  => (a -> Ordering) -> (a, a) -> Maybe a
binarySearchGen p (lo,hi)
  | hi < lo = Nothing
  | otherwise =
      let mid = (lo + hi) ‘div‘ 2 in
      case p mid of
        LT -> binarySearchGen p (lo, mid - 1)
        GT -> binarySearchGen p (mid + 1, hi)
        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)

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

binarySearchList :: Ord a
                 => [a] -> a -> Maybe Int
binarySearchList xs val
  =  binarySearchGen p (0, length xs - 1)
     where
     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

11.3 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.

12 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`.

Bentley, Jon (1986). *Programming Pearls*. Addison Wesley. ISBN 0201103311.

Bentley, Jon (2000). *Programming Pearls*. Addison Wesley, second edition. ISBN 0201657880.

Bird, Richard (1998). *Introduction to Functional Programming using Haskell*. Prentice Hall, second edition. ISBN 0134843460.

Bird, Richard (2010). *Pearls of Functional Algorithm Design*. Cambridge University Press. ISBN 0521513383.

Bird, Richard (2014). *Thinking Functionally with Haskell*. Cambridge University Press. ISBN 1107452643. URL http://www.cs.ox.ac.uk/publications/books/functional/.

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.

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.

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.

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.

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.

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.

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-lp5e.html.

Miller, Bradley W. and David L. Ranum (2011). *Problem Solving with Algorithms and Data Structures Using Python*. Franklin, Beedle Associates Inc, second edition. ISBN 1590282574. URL http://interactivepython.org/courselib/static/pythonds/index.html.

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 deques. *Journal of functional programming*, 5(04):583–592.

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

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.