{{ TOC }}

COMP1730

Lecturer: Patrik Haslum

W1

L1

NO FRIDAY LECTURE this week Log in to STREAMS at cs.anu.edu.au/streams,

Assessments: - 3 lab assignments, 30%, best 2 - 1 homework assignmen, 20% - Mid term, 20% - Final exam, 30%

L2

Procedural abstraction, with robots

Not much is happening lol. Lecturer runs Mint :(((((((((((i think he does at least, we can hope that he doesn't, because that would make me a sad boy

"If it was hard to write, it's probably hard to read. Add a comment"

Abstraction: To use a function, we only need to know *what* it does, not *how*

Function suite: sequence of statements within a function

L3

Variables, expressions and more functions

"\" works to continue a line on to the next one

Expressions are built up of: - literals - variables - operators - function calls

Values and Types

Value types: - integers - floating points - strings - booleans - ...and others that come later

Operators in Python: +,-,*,/ - std

** - power // - floor % - remainder

Function: a piece of the program that is given a name, and can be called by that name (think i already wrote that but yolo)

Variables

L6

while statement

L7

Floating point numbers

We do have a lecture tomorrow! Stuff about the assignment we have next week **Sequential encoding system**: represents each item by a sequence of symbols; the order ofa symbol in the thing *is* important Remember that binary is a *base 2* system

FP Representation: x = ±m x b^e consists of three components: - the sign (+ or -) - the significand (m) - the exponent (e) Number is *normalised* if 1≤m Zero not representable as a normalised number Floats can represent infinity (1/ 1e-320) When adding FP, the absolute rounding erros is porportional tothe magnitude of the largetst number that is rounded

L8

Course Reps on wattle page Need to bring student ID to assignment next week Make sure to name the functions as they are described, otherwise the testing program won't find them and then you'll be shitted

L9

Sequence Data Types; NumPy arrays; Indexing, Length and Slicing Sequence: contains zero or more values Each item in a sequence has a position, or index, ranging from 0 to n-1 Some built-in sequence types: - Strings: only text - List: can contain a mix of arbitrary types - Tuples: like lists, but immutable

NumPy and SciPy are not part of the Python standard library, but are basically essential for scientific/engineering applications

They provide: - an n-dimensional array data type (ndarray) - fast math operations on arrays/matrices - linear algebra, Fourier transform, random number generation, etc.

ndarray: all values in the array must be of the same type, typically numbers

linspace, evenly spaces some shit for you

note: can do import x as y, so that you can referrence y instead of x. Sort of aliasing.

len(sequence) returns length of sequence negative indexing is a thing, allowing you to count backwards. End is -1, 2nd last entry is -2, etc.

NO LECTURE ON FRIDAY

Can perform maths on NumPy arrays, perform operations element-wise on the array

W5

L1

Sequence types, part 2

Indexing a list returns an element, but slicing a list returns a list Arrays support element-wise maths operations Arrays are more time and memory efficient, although this only matters when they are large.

Mutable objects and references In python, every value is an *object* Every object has a unique identifier *Immutable* objects never change (e.g. numbers, strings and tuples) *Mutable* objects can change (e.g. arrays and lists)

Operations on immutable objects create new objects, leaving the original unchanged A mutable object can be modified without it's identity changing. Lists an arrays can be modified through: - element and slice assignment - modifying methods/functions

Tuples Tuples are like lists, but immutable Why both lists and tuples? Immutability is required for certain uses A comma-separated sequence of expressions *without square brackets* creates a tuple, e.g. date = 12, "August", 2016

L2

Functions and Namespaces

Functions (Recap)

Why use functions? - *Abstraction*: To use a function, we only need to know *what* it does, not *how*. - Readability - Divide and conquer - break a complex problem into simpler problems - A function is a logical unit of testing - Reuse: Write once, use many times (and by many).

Namespaces

Assignment associates a (variable) name with a reference to a value (object). This association is stored in a *namespace* Whenever a function is caleed, a new *local namespace* is created Assignments to variables (including parameters) during execution of the function are done in the local namespace The Local namespace disappears when the function ends

Scope: the set of program statements over which a variable exists (i.e. can be referred to). Because there are several namespaces, there can be *different variables with the same name in different scopes*.

Local Assignment Rule Python considers a variable that is assigned *anywhere* in the function suite to be a local variable* (this includes parameters) When a non-local variable is evaluated, only the local namespace is checked The rule considers only *variable assignment*

Modifying is not assignment!!!!!!!!!!!!!

When a function is called, its parameters are assigned *references* to the argument values

Recursion

A recursive function is often described as "a function that calls itself" Function calls form a *stack*: when the n^th function call ends, execution returns to where the call was made in the (n-1)th function suite The function suite must have a branching statemnt, such that a recursive call does not always take place ("base case"): otherwise recursion never ends

Guidelines for good functions: - Within a function, access only local variables - Use parameters for all inputs to the function - Parameter default values make this easy - Use multiple return (a tuple) for all function outputs, **unless the specific purpose of the function is to send output elsewhere** - Don't modify mutable argument values, **unless the specific purpose of the function is to do that**

W6

L1

Algorithm and problem complexity

The time (memory) consumed by an algorithm: - Counted in "elmentary operations" - Expressed as a function of the size of its arguments

Complexity describes scaling behaviour: How much does runtime grow if the size of the arguments grow a certain factor?

Big-O notation - O(f(n)) means roughly "a function that grows at the rate of f(n), for large enough n" - E.g. n^2 + 2n is O(n^2), 100n is O(n), 10^12 is 0(1)

Problem Complexity: the time (memory) that *any* algorithm *must* use, in the worst case, to solve the problem, as a function of the size of the arguments.

Hierarchy Theorem: for any computable function f(n) there is a problem that requires time greater than f(n).

L2

Control, part 3

Dynamic Programming

Idea of storing answers to (recursively defined) subproblems, to avoid computing them repeatedly. Trade memory for computation time. By computing subproblem solutions "from the bottom up", we can also transform a recursive algorithm into an iterative one: - Solve the base cases first - Repeatedly, solve problems whose subproblems are already solved - do until whole problem is solved

W7

L1

I/O and Files

I/O A common way for programs to interact with the world, e.g. reading data (keyboard, files, network) and writing data (screen, files, network) Scientific computing often meants processing or generating large volumes of data.

Files and Directories A *file* is a collection of data on secondary storage (hard drive, USB key, NFS) A program can *open* a file to read/write data Data in a file is a sequence of *bytes* (integer $0 \leq b \leq 255$) A *text file* contains (encodings of) printable characters (including spaces, newlines, etc). A *binary file* contains arbitrary data, which may not correspond to printale characters, e.g. images, audio/video, word documents

Directory Structure Files on secondary storage are organised into *directories* This is an abstraction provided by the operating system (appear differently on different operating systems) The directory structure is typically tree-like

File Path A *path* is string that identifies the location of a file in the directory structure Consists of directory names with a *separator* between each; the last name in the path is the name of the file Two kinds of paths: - Absolute - Relevant to the current working directory

Reading and Writing Text Files When we open a file, python creates a *file object* (or "stream" object) The file object is our interface to the file: all reading, writing, etc, is done through methods of this object The type of file object (and what we can do with it) depends on the *access mode* specified when the file was opened. E.g. text mode vs binary mode, read-only, write-only, read-write mode, etc.

Opening a file: `open(*file path*, *access mode*)` Don't forget to close the file when done! `my_file.close()` Once closed, can't read or write to file without reopening it.

Python Access Modes: - *r* - read only - *w* - write only, erasing content - *a* - write only, appending to pre-

existing content - *r+* - read/write, from beginning of file (overwrite) - *w+* - read/write, erases file content

| | | if file exists... | if it does not exist...| |:-|:-|:-|:-| | r | read only | | error | | w | write only | erases file content | creates a new (empty) file | | a | write only | appends new content at end of file | creates a new (empty) file | | r+ | read/write | reads/overwrites from beginning of file | error | | w+ | read/write | erases file content | creates a new (empty) file | | a+ | read/write | reads/overwrites starting at end of file | creates a new (empty) file | | U | *(python 2.x only) Enable universal newline support* | | b | Open as a binary file (default is text) | | | | | |

Be careful with write modes. Erased or overwritten files *cannot be recovered*

It is possible to check if an existing will be overwritten using `os.path.exists(file_path)` (returns `True` or `False`)

Reading Text Files `*file_obj*.readline()` reads the next line of text returns it as a string, *including* the newline character (`\n`) `*file_obj*.read(*size*)` reads at most *size* characters and returns them as a string * if *size* less than 0 reads to end of file

If already at end-of-file, `readline` and `read` return an empty string `*file_obj*.readlines()` reads all remaining lines of text returning them as a list of strings

File Position A file is a sequence of bytes - but *not* a sequence type! The file object keeps track of where in the file to read (or write) next * the next operation (or iteration) starts from the current position

When a file is opened for reading (mode `r`), the starting position is 0 (beginning of file) File position is *not* a line number

Writing Text Files Access mode `w` (or `a`) opens a file for writing text `*file_obj*.write(*string*)` writes the string to the file * Note: `write` does not add a newline to the end of the string

`print(..., file=*file_obj*` prints to the specified file instead of the terminal

Buffering File objects typically have an I/O buffer * Writing to the file object adds data to the buffer; when full, all data in it is written to the file (“flushing” the buffer)

Closing the file flushes the buffer * If the program stops without closing an output file, the file may end up incomplete

Always close the file when done!

W8

L1

Abstract Data Types

No lecture on Fridau, have exam

- Sequence (length, index, slice)
- Iterable (for loop)

Interface: a set of functions (or methods) that implement operations (create, inspect and modify) on the abstract data type. - The interface creates an abstraction, e.g. "a date has a year, a month and a day" instead of "a date is a list with length 3" - The user of the ADT (i.e. the programmer) must use only the interface functions to operate on values of the ADT - accessing/modifying the structure

Makes code easier to read and understand Makes code *refactorable*; implementation behind the interface can be replaced without changing any code that uses it

Mapping A mapping (a.k.a. dictionary) stores key-value pairs; each key stored in the mapping has exactly one value. Keys do not have to be consecutive integers Uses: - Storing a look-up index - Organising data with "complex" labels (like a multidimensional table) - Storing solutions to subproblems in a dynamic programming algorithm

Interface (first cut): - Create an empty mapping - Returns a new, empty mapping - Store a value with a key - Modifies the mapping (no return value) - Overwrites existing value for the key, if any - Is a given key stored in the mapping? - Look up the value stored for a given key (error if key is not stored)

Implementations of mapping: - Store key-value pairs in a list - Simple to implement - All operations are linear time - Store key-value pairs in a list, sorted by keys - More complicated - key exists, look-up and value replacement in O(logn) time - Adding a new key takes linear time - Hashtable (built-in python type `dict`)

Any programe that deals with complex data needs data structures Creating and using abstract data types hels structure larger programs, making them easier to write, debug, read and maintain Several ways to implement ADTs in python: - Function interface - Using (nested) sequences, and other built-in types (`dict`,`set`) - Defining classes (more later in the course)

W9

L1

Types Of Errors: - Syntax errors: evident as soon as you try to run the code - Runtime errors: arise when the code runs (and maybe only under certain conditions) - Applying a function or operator to the wrong value, or wrong type of value - Indexing past the beginning/end of a list - and many more - Semantic errors: runss without error, but does the wrong thing (e.g. returns the wrong answer)

Exceptions: a control mechanism for handling runtime errors An exception is *raised* when the error occurs The exception moves up the call chain until it is *caught* by a *handler* If no handler catches the exception, it moves all the way up to the Python interpreter, which prints an error (and quits, if in script mode) Python allows the programmer to both raise and catch exceptions

Exception names: - `TypeError`, `ValueError` (incorrect type of value for operation) - `NameError`, `UnboundLocalError`, `AttributeError` (variable or function name not defined) - `IndexError` (invalid sequence index) - `KeyError` (key not in dictionary) - `ZeroDivisionError`

Assertions - `assert(conditon, "fail message")` - evaluates `condition` (of type `bool`) - if the value is not True, raises an `AssertionError` with the (optional) message - Else, continues to next statment Assertions are used to check the programmer's assumtions (including correct use of functions) Function's docstring states assumptions; assertions can check them

Why assert? - "Fail fast": it is usually better for a function to raise an exception as soon as a violation of an assumption is detected - Provide specific error information - "average of emtpy sequence is undefined" is more explanatory than `ZeroDivisionError` - It is *always* better to raise an exception than return an incorrect (garbage) result - Semantic errors are the hardest to find!

The raise statement - `raise *ExceptionName*(...)` - Raises the named exception. Exception arguments depend on exception type - Can be used to raise any runtime error - Typically used with programmer-defined exception types

Catching Exceptions

Exception Handling

```
try:
    suite
except ExceptionName:
    error-handling suite
```

- Execute suite
- If no exception arises, skig *`error-handling suite`* and continue as normal
- If the named exception arises from executing *`suite`* immediately execute *`error-handling suite`*, then continue as normal
- If any other error occurs, fail as normal

E.g:

```
number = None
while number is None:
    try:
        ans = input("Enter PIN:")
        number = int(ans)
    except ValueError:
        print("That’s not a number!")
        number = None
```

Never catch an exception unless there is a sensible way to handle it If a function does not raise an exception, it's return value (or side effect) should be correct

L2

Modules

Every python file is a module A module is a *sequence of statments* Every module has a name

When you run the python shell in 'script mode', the file you're executeing becomes the "main module". - Its name becomes `__main__` - Its namespace is the global namespace Every loaded module becomes a separate (permanent) namespace

When executing `import *modname*`, the python interpreter: - Checks if `*modname*` is already loaded - if not, it: - finds the module file - executes the file in a new namespace - stores the module object in the system dictionary

of loaded modules - and then associates *modname* with the module object in the current namespace

sys.modules is the dictionary of all loaded modules dir(*module*) returns a list of names defined in *module*'s namespace dir() list the current (global) namespace

W10

L1

Introduction to Classes

Classes and Objects

In python, every value (number, string, list, dictionary, etc) is an *object* Every object has a type - We say the object is an *instance* of the type In python, every type is a *class* By defining a class, you add a new type to the language

Example:

```
class Student:
    ''' Simple student class '''
    def __init__(self, first='', last='', unum=0):
        self.first_name = first
        self.last_name = last
        self.u_unumber = unum

    def __str__(self):
        return (self.last_name + ', ' + \
            self.first_name + \
            ' (' + str(self.u_number) + ') ')

    # end class Student
```

Class v. Instance: A class is a template for creating a certain type of objects; they are *instances* of the class.

Creating an Object

To make a new object, call a function with the same name as the class:

```
a_dict = dict()
 a_student = Student()
 a_student = Student('Jane', 'Doe', 1234567)
 an_int = int()
```

The instance-creator function is known as the class' *constructor* - Constructor arguments depend on the class - typically used to initialise the new object

Attributes

Objects are used to store information in a structured way Every object has its own namespace - Names defined in the objects namespace are called *(instance) attributes* Access object attributes with dot notation:

```
>>> a_student.last_name
'Doe'
>>> a_student.year
AttributeError:  ...
```

Programmer-defined classes are *mutable*

```
>>> a_student.last_name = 'Smith'
```

Attributes can be created by assignment:

`>>> a_student.year = 1

dir(a_student)`

A newly created object has no attributes (except for some internal to python, such as __class__, which stores a reference to it's class) A class *initialiser* (__init__ method) can be defined to ensure instances of the class have right attributes.

Defining a class

```
class ClassName:
    ...suite...
```

The `suite` is executed when the class is defined The class has its own namespace - Names (functions and variables) defined in the class namespace are caled *(class attributes)* - The functions defined in the class namespace are also known as its *methods*

Methods

Methods are functions defined in a class A method is always called on an object `an_object.method_name(...*args*...)` The acted-on object is always, implicitly an argument to the method, bound to the first parameter (usually name `self`)

The class initialiser

The `__init__` method is called every time a new instance of the class is created Arguments are the new object (`self`), followed by arguments to the class' constructor: `new_instance = MyClass(5)` results in `MyClass.__init__(new_instance, 5)` `__init__` must not return a value Every class should define `__init__`, to ensure objects have the right attributes for the class

Methods vs. Functions

Methods *are* functions - Defined in a class' namespace - Called with a different syntax Ordinary (globally defined) functions can take objects of any type (including programmer-defined) as arguments and use or modify them Defining methods instead of functions helps reduce "namespace clutter" - Different classes can have different methods with the same name

```
def get_student_name(student):
return student.first_name + ' ' \
+ student.last_name
```

vs.

```
class Student:
    .
    .
    .
    def get_name(self):
        return self.first_name + ' ' \
        + self.last_name
```

L2

Attribute resolution

An object knows what class it is an instance of. To resolve `an_object.name`: 1. First, see if *name* is defined in the object 2. If not, see if *name* is defined in the objects class (*`an_object.__class__`*) 3. If not, check the class' superclass, recursively 4. Else, generate an AttributeError

Applies to methods as well

Polymorphism

Means having several functions/methods with the same name but different number and/or types of parameters. In python, there can only be one function with a given name in a given namespace. However, functions *can* take a variable number of arguments (by supplying default values for arguments), and argument values can be of any type Function can examine argument types using `type` and `isinstance`.

Python standard methods

- **`__str__`** : called to obtain a string representation of the object, e.g. `str(the_object)`
- **`__eq__`** : python evaluates `x == y` by calling `x.__eq__(y)`