THE UNIVERSITY OF SYDNEY

INFO1910 S2 2023 Week 12 Tutorial

Options

The contents of this tutorial are non-assessable, but may further your understanding. If you're interested in any other topics feel free to ask!

Additionally the end of semester survey is now available, feedback on this unit is greatly appreciated.

Numpy

You may have previously noticed that all collections that are native to Python can store objects of any type. However, the type of each element of the collection can be determined in advance, and this can be used for error checking, better performance and for integration with other programming languages (i.e. it runs it in C).

An array is similar to a list, however it should be of a single pre-defined type. Arrays are implemented in the numpy package and are initialised from an existing Python list.

```
import numpy as np
my_arr = np.array([1,2,3,4,5])
print(my_arr)
```

We can specify a type for an array using a 'dtype' object, such as an integer number, a string or a floating point number,along with specifying how many bits each element should be allocated. These types are hard coded into Python (and into numpy) but you can create your own custom types if it is necessary.

```
my_arr_of_integers = np.array([1,2,3,4,5], dtype=np.int32)
my_arr_of_floats = np.array([1,2,3,4,5], dtype=np.float64)
```

New dtypes can also be composed as a collection of existing dtypes as a single object. They are defined by an array of tuples of strings and dtypes. The strings form keys that access a value of the specified type when a variable of that type is used. In all it can be thought of as a rigid implementation of a dictionary collection.

When properly implemented, dtypes can allow for very descriptive and readable Python.

```
# Our dtype is defined as a list of tuples containing a string as a key,
# a standard numpy dtype and a length as an optional argument
dt = np.dtype([('name', np.unicode_, 24), ('age', np.int32)])
```

```
# We can initialise our array with a certain dtype by specifing
# it as a keyword argument
students = np.array([('alice', 19), ('bob', 18), ('eve', 25)], dtype=dt)

# We can now use the keys specified in the definition of our
# dtype to access elements by name
print(students[0]['name'])
print(students['name'])
print(students[:]['age'])
```

Question 1: Reshape and Broadcast

Write a Python function that takes a single n-dimensional numpy array and takes the sum of every element without using numpy's own sum method except for error checking.

You should write your own unit tests for this function.

Some code for generating a randomly sized multidimensional numpy array follows:

```
import numpy as np
from functools import partial

# Some useful variables to limit the number of dimensions
min_dimension_size = 3  # Minimum rank and size of a dimension
max_dimension_size = 10 # Maximum rank and size of a dimension

# A function that truncates our random numbers from range 0 to 1 to
# a random whole number in the range of the number of dimensions
# Note that when casting every element in an array we use the
# astype method with a dtype as the argument
r_array_size = lambda min_dim_size, max_dim_size, random_number: (
  np.dot(
     random_number , (max_dim_size - min_dim_size))
     + min_dim_size
  ).astype(np.int32)

# Partial away the min and max dimension size, just for neatness
r_array_size = partial(
  r_array_size,
  min_dimension_size,
  max_dimension_size)

# We now build a random number of dimensions of random sizes with random co
# ease of re-use
r_array_gen = lambda: np.random.random(
  r_array_size(
     np.random.random(
```

```
        r_array_size(
           np.random.random()
  ))))

# Finally we generate our random array
rand_array = r_array_gen()
```

You may find the size and numpy's reshape functions very useful for implementing an efficient solution. Explain what is happening in the above code to generate the random arrays. Why is *np.dot* used in place of just a regular multiplication?

What does this imply about the difference between operations on numpy ndarrays and regular lists?

Can you find a much neater solution using numpy's randint function?

Question 2: Dtype Builder

Write a python program to read the following file and to build an appropriate dtype and store all the relevant data.

```
sid, name    , grade
1  , alice   , 75
2  , bob     , 60
3  , cathy   , 80
4  , douglas , 85
```

Once you have read from the file and added it to your 'database', find a Pythonic way to sort the students by their grade.

Linear Algebra

The main advantage to arrays over other collections is its applications in linear algebra and mathematics. It would be hard to suggest how to multiply two lists or tuples containing elements of arbitrary typing. This allows us to perform vector and matrix multiplication along with other mathematically useful transformations that might be poorly defined when applied to a collection such as a dictionary.

Question 3: Vectorisation

Once we can deal with variables as single mathematical objects rather than collections, it becomes possible to 'vectorise' our code to perform single operations on a variable rather than looping over each element of the collection. However, this requires that our code is written to expect both array and single values.

Vectorise the following functions such that they can still take a single value, or they can perform the operation on an array.

```
def greater_than_two(qwop):
    if qwop > 2:
        return qwop + 1
    else:
        return qwop - 3

# When vectorised, this function should return an array of
# the maximum value from each of the arrays in a 2D array
def array_max(arr):
    return max(arr)

# Currently this function takes a coordinate and the map
# and returns the updated coordinate. You should
# Remove the dependance on the coordinates and simply update the
# Map and return it.
# The input to this function should be a two dimensional array
def limited_game_of_life(game_map, x, y):
    # Boarders are fixed at zero to prevent attempting to access
    new_value = 0
    # Make sure we are not on the boarder
    if (0 < x * y
    and len(game_map) - 1 > y
    and len(game_map[y]) - 1 > x):
        new_value = (game_map[y, x]
            + game_map[y + 1, x] + game_map[y - 1, x]
            + game_map[y, x + 1] + game_map[y, x - 1])
    # Return the new value
    return new_value
```

One approach is to simply use numpy's own 'vectorize' function, however this is essentially a for loop and grants none of the improved performance that vectorising your own functions does.

Question 4: Multi Map

Using the shape property construct a multi-dimensional map function of one variable such that given $mmap(fn, M)$,

$$M'_{i_0,i_1,\ldots} = fn(M_{i_0,i_1,\ldots}) \tag{1}$$

Question 5: Multi Multi Map

Expand this function to take a function that takes n arguments and n multi-dimensional arrays $M^0 - M^{n-1}$ of equal dimensions and performs

$$M'_{i_0,i_1,\ldots} = fn(M^0_{i_0,i_1\ldots}, M^1_{i_0,i_1\ldots} \ldots M^{n-1}_{i_0,i_1\ldots}) \tag{2}$$

Question 6: Perf

Compare the performance of your method to the following:

```
dims = np.arange(<args>), np.arange(<args>), ...
grid = np.meshgrid(*dims)
m_prime = vectorise(fn)(grid)
```

Question 7: Matrix Multiplication

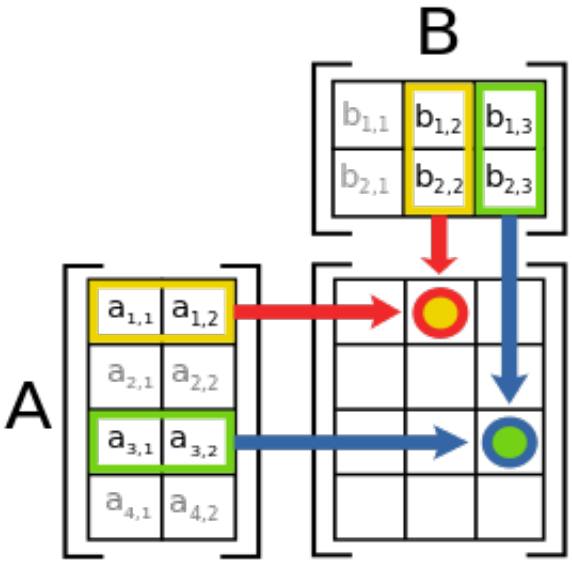
Matrix multiplication is a mathematical procedure that produces a resultant matrix from two input matrices. Given the following function prototype:

```
def matrix_mult(m1, m2)
```

Mathematically, the multiplication of $A \times B$ can be expressed as:

$$AB_{ij} = \sum_{k=0}^{m} A_{ik} B_{kj} \tag{3}$$

where A is an $n \times m$ matrix and B is an $m \times p$ matrix. Graphically, this can be seen as:



Implement your own matrix multiplication function without using numpy. After implementing your own, use the numpy's `numpy.matmult` function while creating arrays using `np.array([1,2,3,4,5], dtype=np.int32)`.

```
matrix_mult([[1,2],[3,4]], [[2,0],[1,2]]) #[[4,4], [10,8]]
```

```
matrix_mult([[1, 2, 3], [4, 5, 6]], [[9, 8, 7]]) # [[46, 118]]
```

Extension Create a function call `matrix_same(m1, m2)` that will check if two matrices are the same. Afterwards use numpy's `array_equal` function to compare the two matrices.

Question 8: Inner Products, Outer Products and Transposes

For the next section, and for 3D plotting in general, it is very useful to be able to generate an array of coordinates. Write a Python function that takes two arrays as arguments that represent the range of the X and Y coordinates and returns two, two dimensional arrays of X and Y coordinates.

You should then find a way to easily 'flip' these coordinates so that you can view your three dimensional map from another angle.

For example:

```
x = [1,2,3,4,5]
y = [10, 12, 14, 16]
```

should return

```
x_map = array(
      [[1, 2, 3, 4, 5],
        [1, 2, 3, 4, 5],
        [1, 2, 3, 4, 5],
        [1, 2, 3, 4, 5]])
y_map = array(
      [[10, 10, 10, 10, 10],
        [12, 12, 12, 12, 12],
        [14, 14, 14, 14, 14],
        [16, 16, 16, 16, 16]])
```

Numpy's outer product function, transpose function and ones function will be useful for doing this efficiently.

Matplotlib

Having presented some more scientific methods of using Python, it would be amiss to not introduce some plotting functions. For this we will use the matplotlib package.

```
import numpy as np
import matplotlib.pyplot as plt
x = np.linspace(0, 5, 100)
plt.plot(x, np.sin(x))
plt.savefig('name.png')
```

You should now be able to see your figure in the 'name.png' file. A useful command to display images is **feh**, which should show your new plot.

We can manipulate this plot to add characteristics such as titles, the spacing on each axis, labels for multiple lines and other features.

You can also generate 3D plots using the mplot3D package.

```
from mpl_toolkits.mplot3d import Axes3d
```

Question 9: Minimisation Visualisation

Plot a three dimensional representation of the Rosenbrock function. Pick some random point and use the gradient descent function to try to find the minimum value of the Rosenbrock function, try to plot the path of the points followed by the gradient descent function overlaid on the same plot.

```
# rosenbrock:
# Returns the value of the rosenbrock function
# at the point [x,y]
# :: x :: The x coordinate
# :: y :: The y coordinate
# Returns the associated z coordinate
def rosenbrock(x, y):
    return (1 - x)**2 + (y - x**2)**2

# gradient_descent:
# Attempts to find the lowest local point
# of a function given some coordinate [x,y]
# :: func :: The function to find the minima
# :: x :: The current x coordinate
# :: y :: The current y coordinate
# :: gamma :: The stepsize, the larger this is
# the less accurate the step, but the smaller
# the more iterations this requires
# Returns a new set of coordinates that should
# be closer to the minima than the first ones
def gradient_descent(func, x, y, gamma):
    dx, dy = [func(x + x * gamma, y)
    - func(x - x * gamma, y),
    func(x, y + y * gamma)
    - func(x, y - y * gamma)]
    return x - gamma * dx, y - gamma * dy
```

You may find it easier to start x and y between 5 and -5, and to set gamma to less than 0.01.

```
x = np.linspace(-10, 10, 100)
y = np.linspace(-10, 10, 100)
z = rosenbrock(x[:,None], y[None,:])
```

Question 10: Foxes and Rabbits

The population of rabbits and foxes is strongly coupled. When there are more rabbits the fox population grows. When there are more foxes the rabbit population shrinks.

We're going to simulate the cycle of the fox and rabbit populations using the Lotka-Volterra model.

```
rabbit_growth_rate = 1 # Rabbits produce more rabbits in the absence of fox
rabbits_eaten = 0.1 # Rabbits get eaten at a certain rate by the foxes
foxes_growth_rate = -1.5 # The foxes should shrink if there are no rabbits!
foxes_eating_rabbits = 0.75 # More foxes appear if they eat rabbits

#   change_in_foxes
#   Determines the change in the number of foxes
#   :: rabbits :: The current number of rabbits
#   :: foxes   :: The current number of foxes
#   Returns the change in the number of foxes
def change_in_foxes(rabbits, foxes):
    change = (foxes_growth_rate * foxes
              + foxes_eating_rabbits * rabbits_eaten * foxes * rabbits)
    return change

#   change_in_rabbits
#   Determines the change in the number of rabbits
#   :: rabbits :: The current number of rabbits
#   :: foxes   :: The current number of foxes
#   Returns the change in the number of rabbits
def change_in_rabbits(rabbits, foxes):
    change = rabbit_growth_rate * rabbits - rabbits_eaten * foxes * rabbits
    return change

#   update_population
#   Updates the population
#   :: rabbits :: The current number of rabbits
#   :: foxes   :: The current number of foxes
#   Returns the change in the number of foxes and rabbits
def update_population(rabbits, foxes):
    rabbits, foxes = [
    rabbits + change_in_rabbits(rabbits, foxes),
    foxes + change_in_foxes(rabbits, foxes)]
    return rabbits, foxes
```

Why would the following approach fail and produce a logical error (but not a syntax error)?

```
def update_population(rabbits, foxes):
    rabbits = rabbits + change_in_rabbits(rabbits, foxes)
    foxes = foxes + change_in_foxes(rabbits, foxes)
    return rabbits, foxes
```

Pick some initial number of rabbits, foxes and some of your favourite numbers for the rates and create an array by updating based off the previous population data. However this is not particularly fine grained and is quite unstable (after a few updates the population of either species can quickly reach positive or negative infinity!).

However the rate at which the population changes is more fine-grained than this approach and would normally require solving a differential equation to model properly. Instead of performing numerical calculus, we're going to just look at how the population changes for different initial values of rabbits and foxes.

```
import numpy as np
import matplotlib.pyplot as plt

max_rabbits = 50
max_foxes = 50

rabbits = np.linspace(0, max_rabbits)
foxes = np.linspace(0, max_foxes)

rabbits_map = 0 # use your earlier function here!
foxes_map = 0 # use your earlier function here!

d_rabbits, d_foxes = update_population(rabbits_map, foxes_map)

plt.quiver(rabbits_map, foxes_map, d_rabbits, d_foxes, cmap=plt.cm.jet)
plt.xlabel("Rabbits")
plt.ylabel("Foxes")
```

You may want to consider if you need to vectorise the update functions for your new rabbits and foxes maps. Look at dir for plt and see how you can include legends, change the colours, set titles and other essential features for a good plot.

Lambda Calculus

The Entscheidungs problem is a famous computer science problem posed by David Hilbert (of Hilbert's problems fame) and Wilhelm Ackermann. It is worded as: *Is it possible to write an algorithm that, given a set of axioms and a statement, that provides a yes or no answer for if the statement is universally valid*

In 1931 this problem was partially bounded by Godel's Completeness and Incompleteness theorems. A set of axioms is held to be complete if for any statement in the language of the axioms that statement or its negation is provable from just those axioms. A set of axioms is consistent if there exists no statement for which both the statement and the negation of the statement is provable from those axioms. We will handwave the notion of a 'grammar' to be an expression of all semantically valid statements given a set of axioms. The incompleteness theorems effectively demonstrated that it is not possible to construct a grammar that is both consistent and complete.

As mathematics is in effect a grammar (or set of grammars), and by extension all programming languages are too this theorem set a hard line in the sand that proved the existence of explicitly unsolvable and uncomputable problems.

In the context of programming; we speculate the existence of a 'halts' function. This takes a function as an argument, if that function halts then it returns true, else it returns false.

Consider the following function

```
def f():
    if halts(f):
        while True:
            pass
    else:
        return
```

If 'halts(f)' is true then the program loops forever end hence does not halt and halts(f) would be false, conversely if halts(f) is false then the program returns immediately and hence halts. From this we can determine that it is not possible to construct a function of this form that determines if a program halts.

With this we can begin to distinguish what a 'computable' function is; while many functions may be described, not all functions may be computed.

Let Λ denote a set of expressions that are computable (while we cannot prove this here, Λ spans the set of computable expressions). We then seek to find a set of rules that determine whether or not an expression is a computable expression.

We can consider a lambda expression to be a function that takes some argument x and performs some action M, we can denote it as: $\lambda x.M$

- If x is a variable then $x \in \Lambda$
- If x is a variable, $M \in \Lambda$, then $(\lambda x.M) \in \Lambda$ (abstraction)
- If $M, N \in \Lambda$, $(MN) \in \Lambda$

In lambda calculus it is worth noting that **everything** is a function. Taking the example from before, given $x \in \Lambda$, x is a valid lambda expression, it then follows that $\lambda x.x$ is a valid expression and denotes a function that takes x and returns x; the identity.

We can then define two operators over the space of expressions:

- α reduction; given $\lambda x.M$, all instances of x may be replaced with another variable without changing the expression
- β reduction: given $(\lambda x, M)N$, all instances of x in M are replaced with N.

Given our previous example with the identity expression we can demonstrate β reduction in applying the identity function to some argument y: $(\lambda x, x)y = \lambda y, y$, this is demonstrably still the identity function. To give another comparison; $f(x)$ would be expressed in lambda calculus as fx.

To give some examples:

- $(\lambda x.f(x))5 \rightarrow f(5)$
- $(\lambda x.f(x))y \rightarrow f(y)$
- $(\lambda x.(\lambda y.f(x,y)))y \rightarrow \lambda y.f(y,y)$

As a shorthand it may sometimes be denoted that:

$$(\lambda ab.M(a,b))x = (\lambda a(\lambda b.M(a,b)))x = \lambda b.M(x,b) \tag{4}$$

That is to say that arguments are reduced from left to right, and that arguments may be reduced without all arguments being present; another valid lambda expression is returned. This is analagous to the behaviour of the partial function from last week's tutorial.

Question 11: Mapping

Consider the function $f = x - y$, evaluate the following lambda expressions:

- $(\lambda x.x - y)(0)$
- $(\lambda y.x - y)(0)$
- $(\lambda x.(\lambda y.x - y))(0)$
- $(\lambda y.(\lambda y.x - y))(0)$
- $((\lambda x.(\lambda y.x - y))(0))(1)$

Expand the following lambda expression:

$$(\lambda xyz.xz(yz))xvw \tag{5}$$

Boolean Operations

Let's encode boolean logic in functions. We will let `TRUE` be a function that returns its first argument and `FALSE` be one that returns its section. This should be sufficient:

```
TRUE  = lambda x, y : x
FALSE = lambda x, y : y
```

From here we can construct some universal operations:

```
AND = lambda x, y : x(y, x)
OR  = lambda x, y : x(x, y)
NOT = lambda x : lambda a, b: x(FALSE, TRUE)
```

And as NAND and NOR are universal we now have reconstructed universal computation using only functions.

Question 12: XOR

Build an XOR operator using the above definitions of TRUE and FALSE.

Arithmetic

Consider the following function:

- $f(a) = 1$
- $f(MN) = f(M) + f(N)$
- $f(\lambda x.M) = 1 + f(M)$

In effect this function counts the 'length' or the number of times a particular variable a occurs within the function.

Using our counting function we can reconstruct the integers given just functions:

- $0 = \lambda fx.x$
- $1 = \lambda fx.fx$
- $2 = \lambda fx.ffx$
- $3 = \lambda fx.fffx$ ...

Where we can evaluate the function by counting on f.

A shorthand here is $n = \lambda fx.f^n x$, we note that 1 is in fact the identity function, and we can leverage this to construct addition.

First let's consider a recursive evaluation of the numbers above:

$$\lambda nx.n(g(x))(0) \tag{6}$$

Question 13: Additivity

Demonstrate that $1 + 1$ in lambda calculus is in fact 2.

Let's rewrite the above in Python, letting $g(x) := x + 1$.

```
int_generator = lambda n: n(lambda x: 1 + x)(0)

ONE = lambda f : lambda x : f(x)
TWO = lambda f : lambda x : f(f(x))
```

We can verify this using `f(ONE)` which will resolve 'ONE' to an integer.

If the addition operation is defined as $\lambda nmpq.np(mpq)$, construct an explicit 'PLUS ONE' operation and verify its correctness in Python.

Question 14: Generators

Instead of the integer interpretation of the integers, consider the following expressions and use them to evaluate some values.

```
g = lambda n: n(lambda x: 2 * x)(1)
g = lambda n: n(lambda x: [0] + x)([])
```

Consider the implications of the idea that lambda calculus in and of itself is not typed; until these objects are applied everything else is merely a function.

The behaviour of these objects appears to resemble iteration; when did that occur?

Consider the list pairing operator; could this be used to construct a linear memory?

Wizardry

Languages can be considered to be comprised of three components; a set of primitive expressions, a set of operations that combine those expressions and a method of abstracting those combinations.

Install a scheme compiler, you can find a bunch here. Scheme is one of the two main dialects of lisp, the other one being common lisp.

Lisp is a list processing language. All objects in lisp are either atoms or lists. Atoms are analogous to any object with a fixed size in memory; numbers, strings, arrays and structures, they may be constants or they may be variables.

The rules for evaluating an expression in lisp are very simple. Every expression is a list,

- Evaluate all sub-expressions of the combination
- Apply the procedure in the left most sub-expression to the arguments that are values

We note that evaluation is naturally recursive.

```
(+ 2 3)
(* 5 6 7)
(+ (+ 6 7) 5)
(print (+ 2 3 5))
(exit)
```

We also note that the action of these operators is identical to the Python 'reduce' function we defined in last week's tutorial.

Variable names in lisp are case invariant.

We can set variables using the define operator:

```
(define foo 7)
(print foo)
```

We can use the same operator to construct functions; instead of passing an atom as the first argument we instead pass a list.

```
(define (foo n) (print n))
```

You may notice at this point that from our previous definition that if lists are both an object in the language and the basic expression in the language, then all programs are themselves lisp objects. The lisp compiler is also a function in lisp.

Lisp is designed for recursive expressions, it also supports switches, loops and iterative constructs; the iterative constructs are special operations, native loops do not exist.

```
(define (pow a b)
    ( * a (
        if (< b 2)
            1
            (pow a (- b 1))
        )
    )
)
```

Given operators are also elements of lists we can use those as outputs of switches.

```
(define (abs_add a b)
    (
        (if (> b 0)
            +
            -)
        a b
    )
)
```

Question 15: Interpretation

Consider the following function fn

```
(define (fn n)
    (if (= n 1)
        1
        (* n (fn (- n 1))))
    )
)
(fn 6)
```

Predict the behaviour of this function.

Question 16: Construction

Implement the following functions in scheme:

- A function to calculate the average of two numbers
- Square root via Newton's method[1]: $x_{n+1} = \frac{1}{2}\left(x_n + \frac{a}{x_n}\right)$, bound your approximation with an argument.
- Newton's method for cube roots

Question 17: Extension

See if you can solve some of the previous in class tasks in scheme.

For further reading: Structure and Interpretation of Computer Programs.

[1] MIT notes on Newton's Method