Tuples & Dictionaries

- Tuples (introduced in COMP10280)
  - A sequence of values, a bit like a list
  - Tuples have a big impact on the way things are coded in Python
- *args
  - Using tuples to allow functions have a variable number of arguments
- Dictionaries
  - A data structure that allows entries be retrieved by key
    - (rather than index)
- Memoization
  - Using a dictionary to speed up processing
    - cache results already calculated
- **kwargs
  - Using a dictionary to allow a function have optional arguments

Tuples

- Pronounced
  - ‘tewple’ as in ‘quadruple’ or
  - ‘tupple’ as in ‘supple’
  - No matter which you choose, people will judge you.
    - Mathematicians are inclined to say ‘tewple’
    - CS people say ‘tupple’ and think ‘tewple’ sounds a bit poncey.

The arguments in favour of ‘tewple’ seem more convincing but I am going to keep saying ‘tupple’ anyway.

Tuples

- A sequence of values
  - like a list
  - but immutable

```
# not allowed
Zone2[0] = 61
```

```
Zone1 = 50,60
Zone2 = (60,70) # brackets are optional
print(Zone1)
print(Zone2)
Zone2[0]

(50, 60)
(60, 70)
Out[40]:
60


type(Zone1)
Out[43]:
tuple
```

Tuple assignment

- Expressions on right evaluated before assignment
  - swap can be done without needing a temp variable
    - imagine an apple in one hand an orange in another and swap them

- Functions can return more than one argument

```
x1 = 11
x2 = 12
x1,x2 = x2,x1
```

```
addr = 'monty@python.org'
uname, domain = addr.split('@')
uname
```

```
Out[57]:
'monty'
```

Tuples as function arguments

- Tuples allow functions to have variable number of args

```
max (33,44,22,11)
44
max (88,22,99,44,33)
99
```

- * gathers arguments into a tuple

```
def printall(*args):
    print("\n Here are the args: ", end="")
    for a in args:
        print(a, " ", end="")

printall(33,44,22,11)
printall(88,22,99,44,33)
```

```
 Here are the args: 33  44  22  11
 Here are the args: 88  22  99  44  33
```

Arrays (again)

- Structured way to store data
- But index needed for access

```
numbers =['one','two','three','four','five']
lookup = [['one','two','three', 'four'],
          ['aon','dó', 'trí','ceathair']
         ]
```

```
In [24]:
print(numbers[2], lookup[1][2])

three trí
```

- Access by content (or key) would be nice

Dictionaries

- A set of key:value pairs
- Values can be retrieved by key
- Keys are unique

```
eng2ir = {'one': 'aon', 'two': 'dó',
          'three': 'trí', 'four':'ceathair',
          'five':'cúig','six': 'sé',
          'seven':'seacht'}
emptyD = dict()

'four' in eng2ir
Out[3]:
True

eng2ir['five']
Out[4]:
'cúig'
```

Dictionary update

- Two ways to update
  - assignment or .update function

```
eng2ir['eight']='ocht'
eng2ir
Out[6]:
{'eight': 'ocht',
 'five': 'cúig',
 'four': 'ceathair',
 'one': 'aon',
 'seven': 'seacht',
 'six': 'sé',
 'three': 'trí',
 'two': 'dó'}
```

```
type(eng2ir)
Out[7]:
dict

type(emptyD)
Out[32]:
dict

emptyD.update({'s1':'string1V1'})
emptyD.update({'s1':'string1V2'})

emptyD   # No longer empty.
Out[9]:
{'s1': 'string1V2'}
```

Using Dictionaries: Token counting

- Classic use of dictionaries
- Access entries by key

```
def histogram(s):
        d = dict()
        for c in s:
            if c not in d:
                d[c] = 1
            else:
                d[c] += 1
        return d
```

```
h = histogram('Muckanaghederdauhaulia')
h
Out[12]:
{'M': 1,
 'a': 5,
 'c': 1,
 'd': 2,
 'e': 2,
 'g': 1,
 'h': 2,
 'i': 1,
 'k': 1,
 'l': 1,
 'n': 1,
 'r': 1,
 'u': 3}
```

Dictionaries: Word counting

- Same as previous function except
  - iterating over a list of words
  - not a string of characters

```
def histWords(s):
        wl = s.split(" ")
        d = dict()
        for c in wl:
            if c not in d:
                d[c] = 1
            else:
                d[c] += 1
        return d

ws = 'she sells sea shells by the
sea shore the shells she sells are
sea shore shells to be sure'
wl = ws.split(" ")
type(ws), type(wl)
Out[10]:
(str, list)
```

Dictionaries: Word counting

```
ws = 'she sells sea shells by the sea shore the
shells she sells are sea shore shells to be sure'
wl = ws.split(" ")
type(ws), type(wl)
Out[10]:
(str, list)

histWords(ws)
Out[6]:
{'are': 1,
 'be': 1,
 'by': 1,
 'sea': 3,
 'sells': 2,
 'she': 2,
 'shells': 3,
 'shore': 2,
 'sure': 1,
 'the': 2,
 'to': 1}
```

```
def histWords(s):
        wl = s.split(" ")
        d = dict()
        for c in wl:
            if c not in d:
                d[c] = 1
            else:
                d[c] += 1
        return d
```

Dictionaries: Memoization

- **Memoization** is an optimization technique used primarily to speed up computer programs by storing the results of expensive function calls and returning the cached result when the same inputs occur again.
- A function for calculating Fibonacci numbers with Memoization

```
def fib(n):
    print('V2 Calc… Fibo… for %d : fib_dict %s' % (n, fib_dict))
    if n<2 : return n
    elif not n in fib_dict :
            fib_dict[n]= fib(n-1) + fib(n-2)
    return fib_dict[n]
```

Why Memoization?

- Otherwise function does the same work multiple times.

```
def fibonacci(n):
    print('Calculating Fibonacci for %d' % n)
    if n<2 : return n
    else: return fibonacci(n-1) + fibonacci(n-2)
```

```
fibonacci(7)
```

```
Calculating Fibonacci for 7
Calculating Fibonacci for 6
Calculating Fibonacci for 5
Calculating Fibonacci for 4
Calculating Fibonacci for 3
Calculating Fibonacci for 2
Calculating Fibonacci for 1
Calculating Fibonacci for 0
Calculating Fibonacci for 1
Calculating Fibonacci for 2
Calculating Fibonacci for 1
Calculating Fibonacci for 0
Calculating Fibonacci for 3
Calculating Fibonacci for 2
Calculating Fibonacci for 1
Calculating Fibonacci for 0
Calculating Fibonacci for 1
Calculating Fibonacci for 4
Calculating Fibonacci for 3
Calculating Fibonacci for 2
Calculating Fibonacci for 1
Calculating Fibonacci for 0
Calculating Fibonacci for 1
Calculating Fibonacci for 2
Calculating Fibonacci for 1
Calculating Fibonacci for 0
Calculating Fibonacci for 3
Calculating Fibonacci for 2
Calculating Fibonacci for 1
Calculating Fibonacci for 0
Calculating Fibonacci for 3
Calculating Fibonacci for 2
Calculating Fibonacci for 1
Calculating Fibonacci for 0
Calculating Fibonacci for 1
Calculating Fibonacci for 2
Calculating Fibonacci for 1
Calculating Fibonacci for 0
Calculating Fibonacci for 1
Calculating Fibonacci for 2
Calculating Fibonacci for 1
Calculating Fibonacci for 0
Calculating Fibonacci for 1
Calculating Fibonacci for 2
Calculating Fibonacci for 1
Calculating Fibonacci for 0
Calculating Fibonacci for 1
```

With Memoization

- Store results in a dictionary
- Only do calculation if result not found in dictionary

```
def fib(n):
    print('V2 Calc… Fibo… for %d : fib_dict %s' % (n, fib_dict))
    if n<2 : return n
    elif not n in fib_dict :
            fib_dict[n]= fib(n-1) + fib(n-2)
    return fib_dict[n]
```

With Memoization

- Work is only done once…

```
#dictionary which store Fibonacci values
fib_dict = {}
result = fib(7)
result
```

```
V2 Calculating Fibonacci for 7 : fib_dict {}
V2 Calculating Fibonacci for 6 : fib_dict {}
V2 Calculating Fibonacci for 5 : fib_dict {}
V2 Calculating Fibonacci for 4 : fib_dict {}
V2 Calculating Fibonacci for 3 : fib_dict {}
V2 Calculating Fibonacci for 2 : fib_dict {}
V2 Calculating Fibonacci for 1 : fib_dict {}
V2 Calculating Fibonacci for 0 : fib_dict {}
V2 Calculating Fibonacci for 1 : fib_dict {2: 1}
V2 Calculating Fibonacci for 2 : fib_dict {2: 1, 3: 2}
V2 Calculating Fibonacci for 3 : fib_dict {2: 1, 3: 2, 4: 3}
V2 Calculating Fibonacci for 4 : fib_dict {2: 1, 3: 2, 4: 3, 5: 5}
V2 Calculating Fibonacci for 5 : fib_dict {2: 1, 3: 2, 4: 3, 5: 5, 6: 8}
Out[11]:
13
```

Keyword Arguments

- print has a keyword argument ‘end’
- In fact it has four

```
print(*objects, sep=' ', end='\n', file=sys.stdout, flush=False)
```

  - because these are normally left out the keyword helps identify them when they are included

```
print("Hello!")
print("Hello!")

Hello!
Hello!
In [22]:
print("Hello!",end=' ')
print("Hello!")

Hello! Hello!
```

Keyword Arguments `**kwargs`

- Tuples allow a function to have a variable number of arguments.
- Dictionaries allow functions to have keyword arguments.
- Program logic must handle them.

```
def demoKwargs(narg, **kwargs):
    print ("Normal arg:", narg)
    for key in kwargs:
        print ("Keyword arg: %s: %s" % (key, kwargs[key]))

demoKwargs(86, KW1="KW1", KW2=22)

Normal arg: 86
Keyword arg: KW1: KW1
Keyword arg: KW2: 22
```

Keyword Arguments `**kwargs`

- Some program logic to handle a simple keyword

```
def printWKwargs(string, **kwargs):
    if 'reps' in kwargs:     # check that 'reps' is a keyword
        r = kwargs['reps']
    else: r = 0
    for i in range(0,r):
        print (i,string)
In [49]:
printWKwargs("Hello!",reps=4)

0 Hello!
1 Hello!
2 Hello!
3 Hello!
In [50]:
printWKwargs("Hello!",repetitions=4)
```

Tuples & Dictionaries

- Tuples
  - A sequence of values, a bit like a list
  - Tuples have a big impact on the way things are coded in Python
- *args
  - Using tuples to allow functions have a variable number of arguments
- Dictionaries
  - A data structure that allows entries be retrieved by key
    - (rather than index)
- Memoization
  - Using a dictionary to speed up processing
    - cache results already calculated
- **kwargs
  - Using a dictionary to allow a function have optional arguments