# COMP10001 Foundations of Computing
# Advanced Functions

Semester 2, 2018

Chris Leckie & Nic Geard

Reminders

- Workshops 7 and 8 due 23:59 Monday 27/8.
- Solutions for Practice Project available soon
- Project 1 slides available on LMS
- Advice: start the project NOW

# Lecture Agenda

- Last lecture:
  - Dictionaries & Sets
- This lecture:
  - for loops (again)
  - Advanced Functions

# For loops (again)

Many students find for loops confusing.

```
for x in some_iterable:
    do thing 1
    do thing 2 (perhaps with x)

end of indentation so end of block
```

Remember:

- `for` in Python is like "for each" in English
- the variable x can be called anything and comes into existance at the for statement
- iterables are lists, dictionaries, tuples, strings, `range()`, views, ...

Use `http://www.pythontutor.com` to see what is going on.

```
for dog in ('spoodle', 'corgi', 'JRT'):
    print("line 1", end=" ")
    print("line 2", end=" ")
    print(dog)
    print("line 4")
print("End of block")
```

Use `http://www.pythontutor.com` to see what is going on.

```
DOGS = ('spoodle', 'corgi', 'JRT')

for i in (0,1,2):
  print("Dog number {0}".format(i), end="")
  print(" is {0}".format(DOGS[i]))

print("End of block")
print("i = {0}".format(i))
```

Use `http://www.pythontutor.com` to see what is going on.

```
DOGS = ('spoodle', 'corgi', 'JRT')

for i in range(len(DOGS)):
  print("Dog number {0}".format(i), end="")
  print(" is {0}".format(DOGS[i]))

print("End of block")
print("i = {0}".format(i))
```

Use `http://www.pythontutor.com` to see what is going on.

```
for i in range(8):
    for j in range(8):
        print("*", end="")
    print("")
```

```
for i in range(8):
    for j in range(i):
        print("*", end="")
    print("")
```

Use `http://www.pythontutor.com` to see what is going on.

```
for i in range(8):
    if i % 2 == 0:
        print('  ', end="")
    for j in range(8):
        print(" * ", end="")

    print("")
```

# For Loops Summary

- `for` in Python means "for each" in English
- Remember, indentation controls the end of a block
- Use for loops to do something to each element of an iterable (list, string, dictionary, tuple, etc)
- Use for loops to do something a fixed number of times (each element of `range()`)
- Alter your iterable to control how the loop variable changes with each iteration of the loop

# Parameters and Arguments I

To allow us to talk precisely about functions, we define

- Parameters are the names that appear in a function definition
- Arguments are the values actually passed to a function when calling it

From `https://docs.python.org/3/faq/programming.html#faq-argument-vs-parameter`

# Parameters and Arguments II

```
def cnt_pos(tup):    # tup is the paramter
    count = 0
    for i in tup:
        if i > 0:
            count += 1
    return(count)

print(cnt_pos((-1,2,3))) # (-1,2,3) is the
                         #          argument
```

(Aside: this is a very common pattern of looping. Remember it as a template for your own coding.)

# Default arguments I

- We have already seen that parameters can be given default arguments.

```
def seconds_in_year(days=365):
    return(days*24*60*60)
```

```
>>> seconds_in_year()
31536000
>>> seconds_in_year(366)
31622400
```

But what is the scope of a default argument value?

# Default arguments II

```
NUM_DAYS_IN_YEAR = 365

def seconds_in_year(days=NUM_DAYS_IN_YEAR):
    return(days*24*60*60)
```

```
>>> seconds_in_year()
31536000
>>> NUM_DAYS_IN_YEAR = 100
>>> seconds_in_year()
```

The default values are evaluated *once* at the point of function definition in the *defining* scope.

# Default arguments III

This means you must be careful with mutable default arguments.

```
def add_on_end(value, lst=[]):
    lst.append(value)
    return(lst)

print(add_on_end(1))
print(add_on_end(2))
print(add_on_end(3))
```

```
print(add_on_end(1, []))
print(add_on_end(2, []))
print(add_on_end(3, []))
```

# Default arguments IV

If you want a mutable default (eg empty list) but not shared between calls

```
def add_on_end(a, L=None):
    if L is None:
        L = []
    L.append(a)
    return L

print(add_on_end(1))
print(add_on_end(2))
print(add_on_end(3))
```

`None` is a predefined constant in Python that has no value.

# Default arguments V

Where can you put default arguments in the function definition?

```
def add_on_end(lst=[], value):
    lst.append(value)
    return(lst)

print(add_on_end(1))
```

```
  File "program.py", line 1
    def add_on_end(lst=[], value):
                   ^
SyntaxError: non-default argument follows
             default argument
```

# Keyword arguments I

So far we have been using *positional* arguments: arguments are matched to their parameters by their position.

```
def f(a, c=3, d=4):
    print("{0} {1} {2}".format(a,c,d))
    return(None)

x = f(1, 2)
```

But we can also match based on keywords (parameter names)

```
x = f(1, d=2)
```

# Keyword arguments II

```
def f(a, c=3, d=4):
    print("{0} {1} {2}".format(a,c,d))
    return(None)

x0 = f()
x1 = f(a=1, 7)
x2 = f(1, a=2)
x3 = f(b=8)
x4 = f(c=8, a=2, d=9)
```

# Keyword arguments III

```
def f(a, c=3, d=4):
    print("{0} {1} {2}".format(a,c,d))
    return(None)

x0 = f()          # f() missing 'a'
x1 = f(a=1, 7)    # Default before non-default
x2 = f(1, a=2)    # f() multiple values for 'a'
x3 = f(b=8)       # what's 'b'?

x4 = f(c=8, a=2, d=9)    # all good
```

# Returning Many Values?

- Functions only return one thing. Once a return statement is executed, the funciton ends.
- Sometimes you want to return two or more things from a function, so return a tuple.

```
def get_stats():

    height = input("Enter height (m): ")
    weight = input("Enter weigth (kg): ")

    return( (height, weight) )

stats = get_stats()
```

# Returning Early I

If your function has the answer it needs, you can return straight away.

```
def any_fail(myList):
    '''
    Returns True if any mark below 50,
    False otherwise. (Inefficient)
    '''

    hasFail = False
    for mark in myList:
        if mark < 50:
            hasFail = True

    return(hasFail)
```

# Returning Early II

```
def any_fail(myList):
    '''
    Returns True if any mark below 50,
    False otherwise. (Smart!)
    '''

    for mark in myList:
        if mark < 50:
            return(True)  # why wait?

    return(False)
```

# Namespaces I

- A "namespace" is a mapping (dictionary!) from names to objects (eg variables and functions).
- When Python starts up there is the global namespace
- When a function is called, a local namespace for that function is called, and then forgotten when the funciton ends
- Scope is the area of python code where a particular namespace is used

# Namespaces II

```
a = 3
def f(x):
    i = 2
    return(x+i)
b = 6
```

In this code snippet

- The global namespace contains `a`, `f` and `b`
- When `f` is called, its local namespace has `x` and `i`

When Python tries to find an object, it first looks in the local namespace, and then in the global namespace

# Namespaces III

```
1. i = 3
2. def f(x):
3.     i = 1
4.     return(x+i)
5. print(f(10))
```

- In this case, the Line 4 code uses the `i` in its local namespace (Line 3)
- Scope of `x` is Lines 2,3,4.
- Scope of global `i` is Lines 1, 2 and 5.
- Scope of the `i` in `f` is Lines 3 and 4.

# Namespaces IV

Now for the tricky part: functions within functions

```
1. i = 3
2. def f(x):
3.     i = 2

4.     def g(x):
5.         i = 1
6.         return(x+i)

7.     return(g(x+i))

8. print(f(10))
9. print(g(10))
```

`f`'s namespace contains `g` and others

# Namespaces V

```
1. def f(x):
2.     i = 1

3.     def g(x):
4.         return(x+i)

5.     return(g(x+i))

6. print(f(10))
```

Python searches local namespace, and then enclosing function namespaces, and then global namespace.
You can list the current namespace with `dir()`.

# Make life easy

- Don't use the same parameter names in sub-functions
- Avoid global variables wherever possible (always!)
- Capitalize constants (a common convention)

```
def f(x):
    ADDER_F = 2

    def g(y):
        ADDER_G = 1
        return(y + ADDER_G)

    return(g(x + ADDER_F))
```

# Lecture Summary

- What is the difference between parameters and arguments?
- What is the difference between a positional argument and a keyword argument?
- In what order must positional and keyword arguments be specified?
- What is a namespace?
- How do you add to a namespace?
- In what order are namespaces searched?