GOVERNMENT OF WESTERN AUSTRALIA

TAFE WA South Metropolitan

Functions (2)

(Session 16)

Review - Writing a functions

- A function is defined using the **def** keyword
- Syntax:

  def *functionName(parameter):*

  functionBody

- *Example:*

```
def my_function(num):
    print(num)
my_function(5)    # calling the function by typing its name and value
```

```
5
```

Returning values

- Let's revisit an extended example from last week.

```
def number():
    print("1")          ← Part of the function code
    return 2            ← Returns from function, with 2 as the result
    print("3")          ← Doesn't run

print("And the number is....")
print(number())    # remember to add parentheses () after the function's name
```

- *What will we get?*

Overview

- Returning None and None
- Argument vs Parameters
- Functions vs Scope
- Global Variables
- Bubble Sort

Returning none and non-none

- Let's have a look at a new keyword: None
- None is not technically a value. However, it depicts the lack of a value.
- The None value cannot be used in equations (it's not the same as 0).
  - If attempted, e.g. *print(None - 1)*, what happens?
- None can, however, by used for some specific uses.
  - You can assign None to a variable or even specifically return it as a result for a function.
  - You can use it to compare the state of a variable (whether it is a None or non-None value)
- By default, a function will implicitly have the result of None, unless it is given a return value.

Returning none and non-none

- By default, a function will implicitly have the result of None, unless it is given a return value.
- So:

```
def funky_function():
    print("Is it funky?")

var_a = funky_function()

if var_a == None:
    print("It's not very funky...")
else:
    print("That's pretty funky!")
```

- Is it funky or not?

Arguments VS parameters

- It can get confusing when looking at the terms arguments and parameters as they're essentially two sides of the same coin, let's clear things up a bit.
- Arguments: Live **outside** the associated function and feed information into the functions.
- Parameters: Only live **inside** a function and need to be stated inside the brackets of the def statement.

Arguments VS parameters

- Let's look back to the previous example:

```
def if_even(n):
    if (n % 2 == 0):
        return True

var_num=int(input("Enter a number, is it truly even? "))
print("For", var_num, "it is :", if_even(var_num))
```

- *So let's find the arguments in this.*

Arguments VS parameters

- Let's look back to the previous example:

```
def if_even(n):
    if (n % 2 == 0):
        return True

var_num=int(input("Enter a number, is it truly even? "))
print("For", var_num, "it is :", if_even(var_num))
```

- *Here it is, in the function call. We've given the argument the value of the var_num variable.*
- *What about the parameter?*

Arguments VS parameters

- Let's look back to the previous example for the parameter:

```
def if_even(n):
    if (n % 2 == 0):
        return True

var_num = int(input("Enter a number, is it truly even? "))
print("For", var_num, "it is :", if_even(var_num))
```

- *Our parameter is the internal name for the value we fed in from the function call.*
- *So we can use the variable of n internally and it will have the stored value.*

Functions vs Scopes

- A scope of something in Python is the areas in which it will be recognised.
- For instance, our previous functions have had parameters. The parameters scope is the function associated, you can’t use it outside of the function, as it isn’t recognised.

Functions vs Scopes

- For instance:

```
def num():
    varNum=20

num()
print(varNum)
```

- What happens? Why?

How scopes work

- This illuminates the first major scope rule:
  - ***A variable that exists outside of a function has a scope internal to that function.***
- *But what happens if we add a little bit to it?*

```
def num():
    varNum=10
    print("Do I know the number? ", varNum)

varNum=20
num()
print(varNum)
```

- *Now what do we get?*

How scopes work

- This illuminates the second major scope rule:
  - ***A variable that exists outside of a function has a scope internal to that function, excluding functions that internally define a variable with the same name.***
- *This is another case of shadowing (from last week).*
- *This is a critical rule to remember!*

Global Variables

- So does that mean scopes are a limiting factor to modifying external variables inside a function? Nope!
- We can shift and extend scopes if we feel the need.
- The global keyword allows us to do this and is used inside the associated function.
- Syntax:

  global variable1, variable2, variable3, …

Global Variables

- So:

```
def num():
    global varNum
    varNum=10
    print("Do I know the number? ", varNum)


varNum=20
num()
print(varNum)
```

- What do we get now?

Interacting with arguments

- Further to scopes, we need to look over how functions interact with their arguments.
- Let's have a look at the following example:

```
def num(n):
    print("It is: ", n)
    n+=1
    print("Now it is: ", n)

varNum=20
num(varNum)
print(varNum)
```

- Changing the value of the parameter **won't cause** the original value of the variable (varNum) to change.
- This is because the parameter only takes on the **value** of the argument, not the argument itself.
- This is the case with scalar (singular) values, but let's look at lists.

Interacting with arguments

```
def num(fList):
    print(fList)
    del fList[0]

L=[4,5,6]
num(L)
print(L)
```

- What happened? Does the previous rule still apply?
- Remember that creating a secondary variable for the same list links to the original, it doesn't make a copy.
- This exception to **lists** still transfers here, too.
- If you structurally change a list associated with a parameter (not the parameter, the list) these changes will be reflected in the original list.

Designing and writing functions

- Over the previous two weeks we have looked at creating functions, all of which looked something like this:

```
def functionName(parameter):
    functionBody
```

- Let's try developing a slightly more tricky function.

Bubble Sort

Bubble Sort is the simplest sorting algorithm

It works by repeatedly swapping the adjacent elements if they are in wrong order.

```
def bubble_sort(lst):
    for i in range(len(lst)):
        for j in range(len(lst) - 1):
            if lst[j] > lst[j+1]:                    # if element is greater
                lst[j], lst[j+1] = lst[j+1], lst[j]  # swapping adjacent element

lst = [7, 12, 9, 2, 15, 6, 1, 4]  # list of numbers
bubble_sort(lst)      #calling function
print(lst)            # printing sorted list
```

```
[1, 2, 4, 6, 7, 9, 12, 15]
```

Linear Search – for finding an element within a list


```
# Linear search algorithm
# If val is present then return True
# else return False
def search(lst, val):
    for i in range(len(lst)):
        if lst[i] == val:
            return True        # we could also return index,   return i
    return False               # we could also return -1

if(search([2, 3, 1, 8, 9], 2) == True):   # is 2 in the list?
    print("Found")
else:
    print("Not Found")
```

Questions?