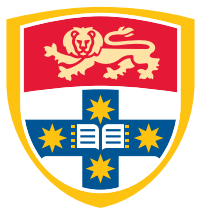
INFO1110 & COMP9001: Introduction to Programming

School of Information Technologies, University of Sydney



Copyright Warning

Lecture 10: More functions, Dictionaries

Local variables and values

Functions: variables

Look at the variable num in the following

```
1  def is_even(num):
2      even = False
3      if num % 2 == 0:
4          even = True
5      return even
6
7  num = 15
8  result = is_even(num)
9  print("result of IsEven: " + str(result))
```

What is the output of this program?

Are all the variables called num the same?

Functions: Naming variables

The name of function arguments can be anything, it is not related to other parts of the program

```
1  def is_even(myNumber):
2
3  def find_maximum(sailorMoonPowerUp):
```

e.g. argument object named "`the_output`" makes more sense to the reader:

```
10 def print_row_stream(the_output, width):
11     i = 0
12     while i < width:
13         the_output.write("*")
14         i = i + 1
15     the_output.flush()
```

Local variables

Function arguments are new variables that exist only for the code block of the function. They are said to be local variables.

```
1  def is_even(num):
2      result = False
3      if num % 2 == 0:
4          result = True
5      return result
6
7  num = 15
8  result = is_even(num)
9  print("result of IsEven: " + str(result))
```

What is the lifetime of `result`?

What about the value of `result`? is it stored in the same place?

What happens to my arguments?

Consider this piece of code:

```
1  def get_total(x, y):
2      x += y
3      return x
```

and now suppose we call the `get_total` function from somewhere else, like this:

```
1  x = 5
2  y = 7
3  s = get_total(x, y)
4  print("s: " + str(s) )
5  print("x: " + str(x) )
```

What is the value of x at the end?

Function call and local variables

When we call a function some values may be given as *arguments*.

When the function begins, it receives the *values* of those arguments.

Local variables are made to store those values, so they can be used. This is equivalent to an = operation.

```
1 def get_total(local_x, local_y):
2     ...
```

```
1 get_total( 23, 2 )
```

get_total(local_x = **23**, local_y = **2**) ...

```
1     int num = 2;
2     get_total( num, num*2 )
```

get_total(local_x = **2**, local_y = **4**) ...

A function call

Consider this example

```
1  def increment(x):
2      x = x + 1
3
4  p = 1
5  increment(p)
6  print(p)
```

What is the outcome?

Using built in functions

`abs(x)` Return the absolute value of a number. $|x|$

```
x = 5
y = -4
result = abs( x - y ) + y + abs(x)
```

`pow(x, y)` Return `x` to the power `y` x^y

```
x = 2
result1 = pow(x, 2)
result2 = pow( pow(x, 2) , 2 )
```

`round(number[, ndigits])` Return number rounded to ndigits precision after the decimal point. If ndigits is omitted or is None, it returns the nearest integer to its input.

```
pi = 3.141592365358979
```

Building on built in functions

Square root of x, $\sqrt{x} = x^{\frac{1}{2}}$.

Suppose we want to calculate $result = -9\sqrt{2} - 3\sqrt{3} + 3\sqrt{5}$

As code, using our builtin `pow(x, 1/2)` we could replicate this everywhere

```
result = -9 * pow(2,1/2) - 3 * pow(3,1/2) + 3 * pow(5,1/2)
```

Building on built in functions

```
result = -9 * pow(2,1/2) - 3 * pow(3,1/2) + 3 * pow(5,1/2)
```

Simplification is good. Factor out the common parts.

```
def mysqrt(x):
    return pow(x, 1/2)

result = -9 * mysqrt(2) - 3 * mysqrt(3) + 3 * mysqrt(5)
```

What if our program exhibits a pattern? can we write a function to remove all these hard coded numbers and deal with an array?

Building on built in functions

We may need to capture further constraints. Wrap that within the function

```
1  # Returns the correctly rounded positive
2  # square root of a numeric value.
3  def mysqrt(x):
4      if x < 0:
5          return 0
6      result = pow(x, 1/2)
7      return result
```

```
1  p = 64.0
2  answer = mysqrt(p)
3  print("mysqrt(p) = " + str(mysqrt(p)) )
4  print("mysqrt(p) = " + str(answer) )
5  p = 9.0
6  print("mysqrt(p) = " + str(answer) )
```

What is happening with the above, how is the calculation being used in each case?

Using Python Standard library: random

`random.random()`
Return the next random floating point number in the range [0.0, 1.0)

```
1  # Returns a double value with a positive sign,
2  # greater than or equal to 0.0 and less than 1.0.
3  import random
4  i = 0;
5  while i < 100:
6      value = random.random()
7      print("random value = " + value)
8      if value < 0.1:
9          print("winner ")
10     i = i + 1;
```

How many times will "winner" be printed?

Dictionaries

Arrays are a contiguous area of memory

Indexed by a number

Ultimately, we want to store data and search for it.

A dictionary stores a collection of values. A **value** is added, updated, removed from the dictionary by searching for the value using a **key**.

Dictionaries are mutable

Dictionaries – creation

A dictionary can be initialised with opening and closing curly brackets { and }.

```
1  # declare an empty dictionary
2  x = { }
3  print("dictionary is " + str(x))
4
5  # declare an initialise a dictionary of 1 key:value pair
6  key = "donuts"
7  value = 3
8  y = { key:value }
9  print("dictionary y is " + str(y))
10
11 # declare an initialise a dictionary of 3 key:value pairs
12 z = { "ice cream":4, "donuts":1, "lollipop":12 }
13 print("dictionary z is " + str(z))
```

Dictionaries access

Index is the key!

```
1  store = { "ice cream":4, "donuts":1, "lollipop":12 }
2
3  print(store['ice cream'])
4  print(store['lollipop'])
5  print(store['polywaffle'])
```

```
4
12
Traceback (most recent call last):
  File "<stdin>", line 5, in <module>
KeyError: 'polywaffle'
```

Dictionaries access (cont.)

Different access with `get()` we can use the return value to check for errors

```
1  store = { "ice cream":4, "donuts":1, "lollipop":12 }
2
3  value = store.get('ice cream')
4  if value != None:
5      print(value)
6  value = store.get('lollipop')
7  if value != None:
8      print(value)
9  value = store.get('polywaffle')
10 if value != None:
11     print(value)
```

Dictionaries access (cont.)

Access all keys and or values of the dictionary using a for loop

```
1 store = { "ice cream":4, "donuts":1, "lollipop":12 }
2 for key in store:
3     print("key: " + key)
4
5 print()
6 for kvpair in store.items():
7     print("kvpair: " + str(kvpair))
```

```
key: ice cream
key: donuts
key: lollipop

kvpair: ('ice cream', 4)
kvpair: ('donuts', 1)
kvpair: ('lollipop', 12)
```

dict.items() returns a collection of key value pairs as tuples

Dictionaries insertion

Setting a new item requires a new `key` and `value`

```
1 store = { "ice cream":4, "donuts":1, "lollipop":12 }
2 store['gum'] = 4
3 print(store)
```

```
{'ice cream': 4, 'gum': 4, 'donuts': 1, 'lollipop': 12}
```

What if that `key` already exists?!

```
1 store = { "ice cream":4, "donuts":1, "lollipop":12 }
2 store['donuts'] = 'homer ate them'
3 print(store)
```

```
{'ice cream': 4, 'donuts': 'homer ate them', 'lollipop': 12}
```

Dictionaries remove

Using the `del` keyword, we remove that object from the dictionary.
`del d[key]`

```
1  company = { 'sales':250000, 'legal':30000, 'promotions':90000,
2     'technical':477000, 'supplies':68000, 'taxation':120000}
3  del company['promotions']
4  print(company)
5  del company['fred']
6  print(company)
```

```
{'legal': 30000, 'sales': 250000, 'supplies': 68000, 'taxation': 120000, '
Traceback (most recent call last):
  File "<stdin>", line 5, in <module>
KeyError: 'fred'
```

Dictionaries remove (cont.)

`del` only deletes from the relevant context. In this case, within the dictionary.

```
1  x = 'The banana has legs!'
2  y = 'Abstraction is often one floor above you'
3  z = 'The old apple revels in its authority'
4  phrases = {3:x, 17:y, 9:z}
5  print(id(y))
6  print(id(phrases[17]))
7  del phrases[17]
8  print(id(y))
9  print(y)
```

```
4394962992
4394962992
4394962992
Abstraction is often one floor above you
```

It does not mean deleted object (not strictly!)

Dictionary more operations

The `in` keyword can be used to test if an object can be found within a dictionary.

```
1 store = { "ice cream":4, "donuts":1, "lollipop":12 }
2
3 if "ice cream" in store:
4     print("Yes! we have ice cream")
5 else:
6      print("oh no!")
```

`len` returns the number of elements in that container. In this case, the key/value pairs

```
1 store = { "ice cream":4, "donuts":1, "lollipop":12 }
2 print(len(store))
```

Dictionary more operations (cont.)

```
1 store = { "ice cream":4, "donuts":1, "lollipop":12 }
```

Iterating through the dictionary can be

- keys only

```
1 for k in store.keys():
2     print(k)
```

- values only

```
1 for v in store.values():
2     print(v)
```

- keys and values

```
1 for (k,v) in store.items():
2     print("key: {}\tvalue: {}".format(k,v))
```

That's all, folks!

This is the end of the lecture material covered in Week 5.