INFO1110 & COMP9001: Introduction to Programming

School of Information Technologies, University of Sydney

Copyright Warning

Week 5: Functions, Tuples, Dictionaries

We will cover: Defining functions, using functions, understanding function operations

You should read: §§2.1 of Sedgewick

Lecture 9: Defining Functions

Building more complex programs

What are functions?

A *function* is a series of instructions that will produce an output based on a number of input.

e.g. the sine function $y = sin(x)$. `sin()` produces one output value for one input value

Functions can have many inputs

```
total = sum(4, 8)
```

Functions can have many outputs

```
(rank, wins, games) = get_player_stats(player_list)
```

What are functions? (cont.)

A familiar case for strings

```
    name = ".....";
    name_len = len(name)
```

Functions can produce output in different forms

```
print_transaction(x)
save_transaction(x)
send_transaction(x)
```

A function is a separate part of a program that performs some operations, which can be invoked from somewhere else in the program.

Why use functions?

Answer 1: it's tidy and easier to understand:

Even slightly complex programs contain nested loops, conditions, and many variables

Consider the point of view from

- Designing the new solution
- Designing new test data
- Code maintainers point of view

Who are these people?

Why use functions?

An example of using functions

```python
if get_kettle_filled() < 200:
    fillKettle()

turn_kettle_on()

while not is_kettle_boiling():
    pass

turn_kettle_off()

pour_kettle_amount(200)
```

During design, we can identify pieces of the problem. The logic of each can be described in its own function.

Why use functions?

Answer 2: it allows for code re-use:

Don't have to reinvent the wheel each time

e.g. calculate it again, search for it again, read from input again, displaying output again

Why use functions?

Answer 3: to reduce the chance of error:

```
# turn on kettle
if model == 34:
        rc = 32
        pin = 40
        hval = 5
if model == 50:
        rc = 48
        pin = 36
        hval = 3
...
iv = (hval * (rc + 5)) / rc
write_pin_value(pin, iv, 40, 0)

# test until kettle is boiling
while True:
    if model == 34:
        rc = 32
        pin = 40
        hval = 5
    if model == 50:
        rc = 48
        pin = 36
        hval = 3
    ...
    val = read_pin_value(pin, iv, 40, 0)
    boil = False
    if model == 34:
        boil = val > 990
    if model == 50:
        boil = val > 7542
    ...
    if boil:
        break

# test until kettle is boiling
if model == 34:
   rc = 32
   pin = 40
   lval = 0
if model == 50:
   rc = 48
   pin = 36
   lval = 1
...
iv = (lval * (rc + 5)) / rc
write_pin_value(pin, iv, 40, 0)
```

Why use functions?

- do not repeat the same code

```
# turn on kettle
if model == 34:
        rc = 32
        pin = 40
        hval = 5
if model == 50:
        rc = 48
        pin = 36
        hval = 3
...
iv = (hval * (rc + 5)) / rc
write_pin_value(pin, iv, 40, 0)

# test kettle is boiling
while True:
    if model == 34:
        rc = 23
        pin = 40
        hval = 5
    if model == 50:
        rc = 48
        pin = 36
        hval = 3
    ...
```

```
    val = read_pin_value(pin, iv, 40, 0)
    boil = False
    if model == 34:
        boil = val > 99
    if model == 50:
        boil = val > 780
    ...
    if boil:
        break

# turn off kettle
if model == 34:
   rc = 32
   pin = 40
   lval = 0
if model == 50:
   rc = 48
   pin = 26
   lval = 1
...
iv = (lval * (rc + 5)) / rc
write_pin_value(pin, iv, 40, 0)
```

Why use functions?

```python
if get_kettle_filled() < 200:
    fill_kettle()

turn_kettle_on()

while not is_kettle_boiling() :
    pass

turn_kettle_off()

pour_kettle_amount(200)

if is_kettle_keep_hot_enabled():
    turn_kettle_on()
```

Any block of code required for `turn_kettle_on()` is not repeated

Splitting code up into functions

Here's a program to print if a letter is a vowel to the console:

```python
input = 'a'
if input == 'a':
    print('a' + " is a vowel")
elif  input == 'e':
    print('e' + " is a vowel")
elif input == 'i':
    print('i' + " is a vowel")
elif input == 'o':
    print('o' + " is a vowel")
elif input == 'u':
    print('u' + " is a vowel")
else:
    print(input + " is not a vowel")
```

Splitting code up into functions (cont.)

It seems to be doing everything at once,

1. trying to find out if it is a vowel

2. printing a specific message for each case

what if we do something else with vowels? do we just copy/paste this code and modify?

Splitting code up into functions

Identify what is the *fundamental*[1] task...do this as a function.

```python
def is_vowel(ch):
    if ch == 'a':
        return True
    elif  ch == 'e':
        return True
    elif ch == 'i':
        return True
    elif ch == 'o':
        return True
    elif ch == 'u':
        return True
    else:
        return False
```

```python
input = 'd'
answer = ""
if not is_vowel(input):
    answer = "not "
```

```python
print(input + " is " + answer + "a vowel")
```

[1] principal, essential, most important, exactly one useful thing, quintessential

Function Anatomy

There are four parts to each function:

- the function *name* (what it's called)
- the function *arguments* (the information / variables we pass it)
- the *return type* (what kind of thing is returned by the function)
- the function *body* (the actual code that does the work)

The function name and list of argument types make up the *function signature.*

Arguments and Parameters

These terms will come up often, but are often interchangeable

argument something that is *passed* to a function

parameter something that is *used* by a function

```
import sys

def print_row(width):
    i = 0
    while i < width:
        print("*", end='')
        i = i + 1
    print("")
```

```
printRow(10)
```

Why are there two?

Scope matters with functions

When was `myvariable` created?

How many names `myvariable` exist at present?

How can `myvariable` move around?

Returning

When you call a function you can give it information to process. e.g. a calculation $f(x)$ requires x

You also have the option of getting something back from the function – a message to say "yes it's prime" or a value that's the square root of the number given: $message = \sqrt{x}$.

Calling a Function

- Get together information I want the function to handle
- Invoke the function by using its name and supplying the information as arguments
- Do something with the returned item

Wait wait wait…return means print, right?

☠ ***Wrong***

If your code is intended to *return* something then you should NOT just print it out — this is horribly incorrect:

```
def get_max(nums):
	# ... clever code to find out the maximum
	print(maximum)
```

and so is this:

```
def get_max(nums):
	# ... clever code to find out the maximum
	print(maximum)
	return
```

Returning from functions

Once you `return` from a function, that function ceases execution.

```
def double_my_number(n):
      if (n < 0:
         print("0")
      else:
         print( 2*n )
      return 0
```

```
def double_my_number(n):
      if n < 0:
         return 0

      return (2 * n)
```

What are the differences to compare with the above functions?

Returning from functions (cont.)

What is the expected behaviour with the following?

```python
def double_my_number(n):
      if n < 0:
         return 0

      return ( 2 * n )
```

```python
import sys

n = int(sys.argv[1])
twiceSize = double_my_number(n)
print(twiceSize)
```

⚠ You can `return` from anywhere in a function, but be careful! once you return you can only begin the function from the very start.

Returning from functions (cont.)

Here's an example of a potential problem with return:

```python
def get_total_from_input():
    total = 0
    numbers = [None] * 3

    numbers[0] = input()
    if numbers[0].isalpha():
      return total

    numbers[1] = input()
    if numbers[1].isalpha():
      return total

    numbers[2] = input()
    if numbers[2].isalpha():
      return total

    total = numbers[0] + numbers[1] + numbers[2]
    return total
```

Returning what?

The function return type is usually given at the beginning of the definition of the function. Any return statement must, *must* return something of that type.

In Python, it can be anything, but something will always be returned, even if it is `None`.

```python
def foo():
    # ...
    return

x = foo()
print(type(x))
```

Returning what? (cont.)

Why would not knowing the type of returned object be considered bad style?

```python
def useful_function():
    # ...
    return ?

x = foo()
if x is int:
    # ...
elif x is float:
    # ...
elif x is str:
    # ...
elif x is Xenomorph:
    # ...
elif x is None:
    # ...
else:
    # ... we don't know, but should handle
```

How many things can be returned?

You can return only *one* data type.

Don’t forget a list has many elements and is just one object!

```python
def get_random_two_words(words):
    '''return two words from the list'''
    index1 = random.randint(0, len(words))
    index2 = random.randint(0, len(words))
    results = [ words[index1] , words[index2] ]
    return results

colours = ['Green', 'Blue', 'Black', 'Red', 'White', 'Yellow']
results = get_random_two_words(colours)
print(results[0])
print(results[1])
```

How many things can be returned? (cont.)

We have seen simpler ways to do this

```python
def get_random_two_words(words):
    '''return two words from the list'''
    index1 = random.randint(0, len(words))
    index2 = random.randint(0, len(words))
    return ( words[index1] , words[index2] )

colours = ['Green', 'Blue', 'Black', 'Red', 'White', 'Yellow']
(colour1,colour2) = get_random_two_words(colours)
print(colour1)
print(colour2)
```

Can you adjust the code to return two colours that are never the same assuming the list size >= 2?

Tuples: a restricted list

There are cases where there is no need to modify the contents of a list, or it's length.

List is mutable, contents can change []
Tuple is immutable, contents cannot change ()

Returning from a function is a good example of this.

- When the function returns, it must hand over control of the object to the caller. The function then ceases to exist.
- The information returned are values as an output of the function.
- The order in which those return values are presented is fixed.

These restrictions are useful for enforcing *safety* when dealing with packing and unpacking of sequence data.

Tuples: a restricted list (cont.)

Recall `enumerate()`. It will provide an index and the value of each element in the list.

```python
words = ['book', 'lamp', 'desk', 'chair', 'pen']
for index,value in enumerate(words):
    print(index)
    print(value)
```

For this to work, there needs to be a match between the input mapping and the output mapping.

> Sequence unpacking requires that there are as many variables on the left side of the equals sign as there are elements in the sequence.

Programmer *expects* enumerate to return in this form:

```
[(0, ’book’), (1,’lamp’), (2,’desk’), (3,’chair’), (4,’pen’)]
```

Tuples: a restricted list (cont.)

Programmer can pass an object without worrying about changes. They are sharing the object, sharing that memory region

```python
def untrusted_code_plugin(stats):
    (name, score) = stats

    # hmm, according to documentation, 2nd element is the score...
    # I try to modify server data so I am the best player there is.
    stats[1] = 999999
    stats[0] = stats[0] + " is the best you know"
```

```python
def untrusted_code_plugin(account_details):
    (account, balance) = account_details

    # hmm, according to docs, this function is called every day
    # I will add a bit of interest to my offshore bank balance hehe
    if account == 9293728349239022 8:
        account_details[1] *= 1.01
```

Tuples: a restricted list (cont.)

```python
s = (1, 2, 3)
s[1] = 4
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

Immutable vs mutable objects

Immutable

- int, float, bool, string are never updated, there is a new object each time.

```
x = 5000
print(id(x))
x += 1
print(id(x))
```

- Tuple can store multiple elements, has an index, cannot change

Immutable vs mutable objects (cont.)

Every change to an immutable type can force the construction of a new immutable type.

How many objects?

```python
def add(x,y):
    total = x + y
    return total

a = 500
b = 800
total = add(a, b)
```

How many objects?

```python
a = 500
b = 800
total = 0
while a < 800:
    total += add(a, b)
    a += 1
```

Immutable vs mutable objects (cont.)

Mutable

- List can store multiple elements, has an index, can change
- Set can store multiple elements, no order, no duplicates, can change
- Dictionaries can store multiple elements, duplicates has a key, can change
- File objects can be modified, though we will see this next week!

Immutable vs mutable objects (cont.)

```python
def add(x, y, total_list):
    total_list[0] += x + y

a = 500
b = 800
total_list = [0]
while a < 800:
    add(a, b, total_list)
    a += 1

print(total_list[0])
```

Remember, scope matters whether immutable or mutable.