

# Protocols and Function Overloading

---

COMP3522 OBJECT ORIENTED PROGRAMMING 2

WEEK 4



# List Slicing

---

# "Slicing" a list

---

- We can access a single element in a list using its index. `my_list[0]`
- We can also work with a **group of elements** in a list
- We call this group of items a **slice**

We use the square brackets and a colon:

```
>>> letters = list('Hello world')
>>> letters
['H', 'e', 'l', 'l', 'o', ' ', 'w', 'o', 'r', 'l', 'd']
>>> print(letters[0:5])
['H', 'e', 'l', 'l', 'o'] # indices 0 through 4
```

# "Slicing" a list

---

```
>>> breakfast = ["0 - Eggs", "1 - Toast", "2 - Granola", "3 - Yoghurt", "4 - Banana", "5 - Orange", "6 - Juice"]
```

## Omitting the first index starts at the beginning

```
>>> print(breakfast[:3])  
['0 - Eggs', '1 - Toast', '2 - Granola']
```

## Omitting the second index ends at the end

```
>>> print(breakfast[3:])  
['3 - Yoghurt', '4 - Banana', '5 - Orange', '6 - Juice']
```

# "Slicing" a list

---

Providing a 3<sup>rd</sup> Value indicates the step size when traversing the list

```
>>> print(breakfast[2:5:2])  
['2 - Granola', '4 - Banana']
```

You can skip values

```
>>> print(breakfast[::2])  
['0 - Eggs', '2 - Granola', '4 - Banana', '6 - Juice']
```

# Try Out These Statements

---

```
>>> breakfast = ["0 - Eggs", "1 - Toast", "2 - Granola", "3 - Yoghurt", "4 - Banana", "5 - Orange",  
                 "6 - Juice"]
```

```
>>> print(breakfast[:5])
```

```
>>> print(breakfast[2:])
```

```
>>> print(breakfast[2:5])
```

```
>>> print(breakfast[2:5:2])
```

```
>>> print(breakfast[5:2:-1])
```

# Try Out These Statements

---

```
>>> breakfast = ["0 - Eggs", "1 - Toast", "2 - Granola", "3 - Yoghurt", "4 - Banana", "5 - Orange",  
                "6 - Juice"]
```

```
>>> print(breakfast[:5]) #['0 - Eggs', '1 - Toast', '2 - Granola', '3 - Yoghurt', '4 - Banana']
```

```
>>> print(breakfast[2:]) #['2 - Granola', '3 - Yoghurt', '4 - Banana', '5 - Orange', '6 - Juice']
```

```
>>> print(breakfast[2:5]) #['2 - Granola', '3 - Yoghurt', '4 - Banana']
```

```
>>> print(breakfast[2:5:2]) #['2 - Granola', '4 - Banana']
```

```
>>> print(breakfast[5:2:-1]) #['5 - Orange', '4 - Banana', '3 - Yoghurt']
```

# RANGE

---



# It's easy to create lists of numbers

---

Python's `range( )` function can help us create lists composed of sequences of numbers:

```
>>> list(range(10))
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
>>> list(range(1, 11))
```

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

```
>>> list(range(0, 30, 5))
```

```
[0, 5, 10, 15, 20, 25]
```

# It's easy to create lists of numbers

---

Note the relationship between the arguments passed to the function and the list that is actually created:

```
>>> list(range(0, 10, 3))  
[0, 3, 6, 9]  
>>> list(range(0, -10, -1))  
[0, -1, -2, -3, -4, -5, -6, -7, -8, -9]  
>>> list(range(0))  
[]  
>>> list(range(1, 0))  
[]
```

# What's a range?

---

It represents an immutable sequence of numbers

It is commonly used for looping in a *for loop*

Stores a few values:

1. Start value
2. Stop value
3. Step value.

That's it!

So no matter how big the range is, the range object that stores it is *just three values*.

This is very memory efficient

# More about the range data type

---

```
>>> r = range(0, 20, 2)
>>> print(r)
range(0, 20, 2)
>>> print(type(r))
<class 'range'>
```

Range contains useful functions that provide:

1. Containment tests (in)
2. Element index lookup
3. Slicing
4. Support for negative indices.

```
>>> print(10 in r)
>>> print(r.index(10))
>>> print(list(r[2:5]))
>>> print(r[-1])
```

# Range example code

---

```
>>> r = range(0, 20, 2)          10
>>> r                            >>> r[:5] #get first 5 index values
range(0, 20, 2)                  [0, 2, 4, 6, 8]
>>> 11 in r                      >>> r[-1] #get last value
False                            18
>>> 10 in r
True
>>> r.index(10)
5
>>> r[5]
```

# We can find the min and the max easily

---

```
>>> digits = range(1, 11)
>>> digits
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> max(digits)
10
>>> min(digits)
1
>>> sum(digits)
55
>>> letters = list('Python > Java')
>>> sum(letters) # <-- <-- Will this work?
```

# Try Out These Statements

---

```
>>> print(list(range(1,10)))
```

```
>>> print(list(range(0, 10, 2)))
```

```
>>> print(list(range(10, 0, -2)))
```

```
>>> print(list(range(0)))
```

```
>>> print(list(range(1, 0)))
```

# Try Out These Statements

---

```
>>> print(list(range(1,10))) #[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
>>> print(list(range(0, 10, 2))) #[0, 2, 4, 6, 8]
```

```
>>> print(list(range(10, 0, -2))) #[10, 8, 6, 4, 2]
```

```
>>> print(list(range(0))) #[]
```

```
>>> print(list(range(1, 0))) #[]
```



# Iterators

---

# What's an iterator (the iter function)

---

Python and other languages support the concept of **iterating** over containers:

```
mytuple = ("apple", "banana", "cherry")  
myit = iter(mytuple)  
print(next(myit))  
print(next(myit))  
print(next(myit))
```

An iterator is an object that contains a countable number of values

We can repeatedly call the iterator's next method or use the next global function

We can iterate upon an iterator

we can traverse all the values

# How does an iterator work?

---

It **streams** the contents of the container

Calling `next( )` returns successive items in the stream

When no more data is available, a **StopIteration** exception is raised

When this happens, the iterator is exhausted and cannot be used again

A container object like a list produces a fresh iterator each time we:

1. Pass it to the `iter( )` function
2. Use it in a for loop.

# Final word about iterators (for now)

---

Strings are also iterable:

```
mystr = "banana"
myit = iter(mystr)
print(next(myit)) #b
print(next(myit)) #a
print(next(myit)) #n
print(next(myit)) #a
print(next(myit)) #n
print(next(myit)) #a
print(next(myit)) #StopIteration exception
```

# Final word about iterators (for now)

---

Strings are also iterable: **Using while loop**

```
mystr = "banana"  
myit = iter(mystr)
```

```
while True:  
    try:  
        item = next(myit)  
    except StopIteration:  
        break  
  
    print(item)
```

# Final word about iterators (for now)

---

Strings are also iterable: **Using for loop WITHOUT iterator**

```
mystr = "banana"
```

```
for item in mystr:  
    print(item)
```

# Final word about iterators (for now)

---

Strings are also iterable: **Using for loop WITH iterator**

```
mystr = "banana"
my_it = iter(mystr)

for item in my_it:
    print(item)
```

Can Iterators work with dictionaries? Try it out!

What gets printed when `next(dictionary_iterator)` is called?

Try out all the edge cases!



# Final word about iterators (for now)

---

```
animal_and_foods = {  
    'dog': 'bone',  
    'cat': 'fish',  
    'bird': 'seed',  
    'elephant': 'peanuts'  
}
```

```
iterator = iter(animal_and_foods)  
print(next(iterator))  
print(next(iterator))  
print(next(iterator))  
print(next(iterator))
```

Getting iterator from dictionary retrieves the keys

Output:

dog

cat

bird

elephant

# Final word about iterators (for now)

---

```
animal_and_foods = {  
    'dog': 'bone',  
    'cat': 'fish',  
    'bird': 'seed',  
    'elephant': 'peanuts'  
}
```

```
iterator = iter(animal_and_foods.values())  
print(next(iterator))  
print(next(iterator))  
print(next(iterator))  
print(next(iterator))
```

Output:  
bone  
fish  
seed  
peanuts

Need to explicitly call **values** method to get iterator of values

# Final word about iterators (for now)

---

```
animal_and_foods = {  
    'dog': 'bone',  
    'cat': 'fish',  
    'bird': 'seed',  
    'elephant': 'peanuts'  
}
```

```
iterator = iter(animal_and_foods.items())  
print(next(iterator))  
print(next(iterator))  
print(next(iterator))  
print(next(iterator))
```

Need to explicitly call **items** method to get iterator of key/value pairs

Output:

('dog', 'bone')

('cat', 'fish')

('bird', 'seed')

('elephant', 'peanuts')

# PROTOCOLS

---

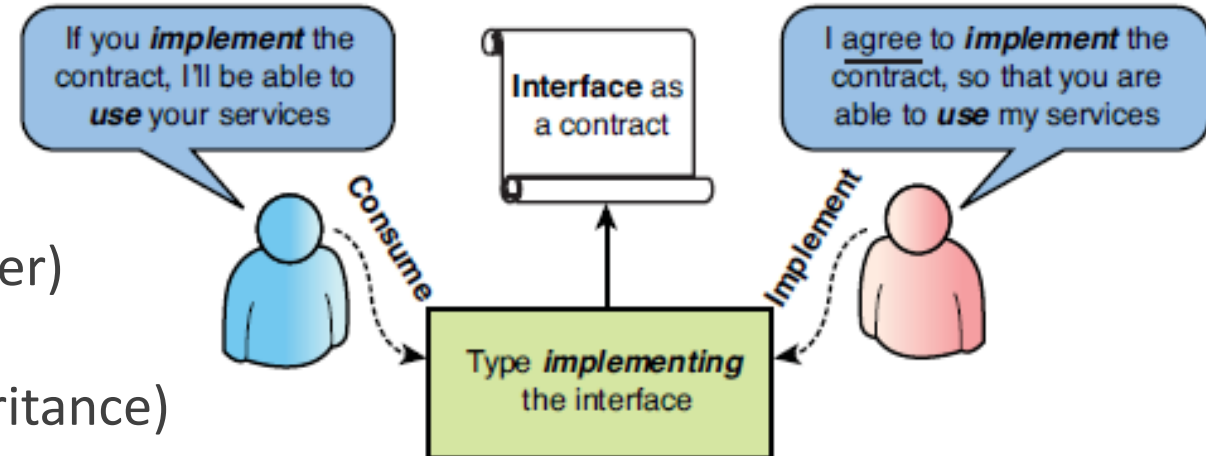
# Remember Interfaces? (I'm going to keep bringing this up)

An Interface is a contract demanding that the class implement certain methods and behaviors.

In more general OOP terms, an interface refers to the public methods and attributes that are **provided by a class**. An interface determines how we can interact with an object. These public attributes and methods are the only way to communicate (or interface) with the class.

In Python there are two kinds:

1. Informal interfaces
  - a) Duck typing
  - b) **Protocols** (We will look at this later)
2. Formal interfaces
  - a) Abstract base classes (using inheritance)



# The Python protocol

---

The lack of control doesn't mean there is no organization

Python has protocols described in the typing module

Protocols are collections of methods that Python developers and libraries expect certain “kinds” of things to have

This is analogous to interfaces in the Java Collections Framework

There are some very helpful protocols in Python

In order to implement a protocol, a type must support certain operations

# The Sized protocol

---

Implemented by having a method called `__len__(self)`

Invoked when an instance is passed to the built-in function `len()`

This is how an object reports the correct value to the `len()` function

We may assume Python's built-in `len()` function works something like this:

```
def len(obj):  
    return obj.__len__()
```

`__len__(self)` returns the length of the object, an integer  $\geq 0$

Implemented by `str`, `list`, `range`, `tuple`, `bytes`, `set`, `dict`

# The Container protocol

---

Provides the method `__contains__(self, item)`

When the membership operator **in** or **not in** is invoked on an object, this is the method invoked

This is how a class implements the membership test operator

`__contains__(self, item)` must return true if the item is **in** self, else false

Implemented by str, list, range, tuple, bytes, set, dict



# The Iterable protocol

---

Provides the method `__iter__(self)`

Invoked to generate an iterator object for a container

Should return a new iterator object that can iterate over all the objects in the container

Invoked by

1. passing the object to the global Python `iter( )` function
2. using the for-loop, i.e., `for item in iterable`

# Mutable vs Immutable Function parameters

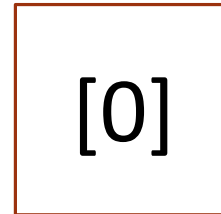
---

# Mutable function parameter

---

```
def spam(eggs):  
    eggs.append(1)  
    eggs = [2, 3]
```

```
ham = [0]  
spam(ham)  
print(ham)
```



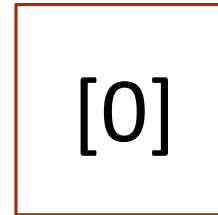
A new object is created in memory, and the variable `ham` is assigned to it

# Mutable function parameter

---

```
def spam(eggs):  
    eggs.append(1)  
    eggs = [2, 3]
```

```
ham = [0]  
spam(ham)  
print(ham)
```



ham  
eggs

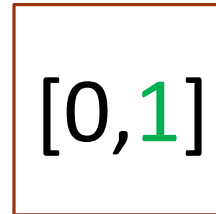
eggs is a new variable that points to the same value ham is pointing at. Eggs and ham are both pointing at the same object in memory. There is NO COPYING

# Mutable function parameter

---

```
def spam(eggs):  
    eggs.append(1)  
    eggs = [2, 3]
```

```
ham = [0]  
spam(ham)  
print(ham)
```



[0, 1]

ham  
eggs

eggs has the value 1 appended to it. This changes the original value ham is assigned to

# Mutable function parameter

---

```
def spam(eggs):  
    eggs.append(1)  
    eggs = [2, 3]
```

```
ham = [0]  
spam(ham)  
print(ham)
```

[0,1]

ham

[2, 3]

eggs

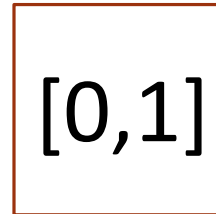
The object [2,3] is created in memory and eggs has now changed to point to it

# Mutable function parameter

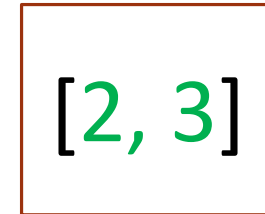
---

```
def spam(eggs):  
    eggs.append(1)  
    eggs = [2, 3]
```

```
ham = [0]  
spam(ham)  
print(ham)
```



ham



eggs

When we print ham, its value is [0,1]

Mutable parameters appear to treat function parameters as pass by reference. Changes made internally to the mutable parameter change the original object

However assignment (=) causes the parameter to point to different objects

# Immutable function parameter

---

```
def spam(eggs):  
    eggs += "world"  
    print(eggs)
```

```
ham = "hello"  
spam(ham)  
print(ham)
```



A rectangular box with a red border contains the text "hello".

ham

A new immutable string object is created in memory, and the variable ham is assigned to it



# Immutable function parameter

---

```
def spam(eggs):  
    eggs += "world"  
    print(eggs)
```

```
ham = "hello"  
spam(ham)  
print(ham)
```



ham  
eggs

"hello" is passed as a parameter to eggs. Before executing the first line, eggs and ham are both pointing at the same "hello" object

# Immutable function parameter

---

```
def spam(eggs):  
    eggs += "world"  
    print(eggs)
```

```
ham = "hello"  
spam(ham)  
print(ham)
```

"hello"

"helloworld"

ham

eggs

Strings are immutable. Meaning that we can't change a string in memory that already exists  
To append "world" to "hello" this requires creating an entirely new string

# Immutable function parameter

---

```
def spam(eggs):  
    eggs += "world"  
    print(eggs)
```

```
ham = "hello"  
spam(ham)  
print(ham)
```

"hello"

ham

"helloworld"

eggs

When we assign "helloworld" to eggs, this makes egg point to the new "helloworld" object

# Immutable function parameter

---

```
def spam(eggs):  
    eggs += "world"  
    print(eggs)
```

```
ham = "hello"  
spam(ham)  
print(ham)
```

"hello"

ham

"helloworld"

eggs

eggs is pointing at "helloworld" so this is printed to the screen

# Immutable function parameter

---

```
def spam(eggs):  
    eggs += "world"  
    print(eggs)
```

```
ham = "hello"  
spam(ham)  
print(ham)
```

"hello"

ham

"helloworld"

eggs

ham is still pointing at the string "hello" so it prints out "hello"

Immutable parameters make the function appear to treat parameters as pass by value

# Immutable vs mutable parameters summary

---

Be aware of what types of parameters are passed into a function

## Mutable parameters

- Can make functions appear to be pass by reference
- Can change the internal attributes
- Changes will affect the original object
- Assignment causes parameter to point to a different object. Causes function to appear to be pass by value

## Immutable parameters

- Can make functions appear to be pass by value
- Can NOT change the internal attributes. This is consistent with the concept of immutable types
- Any changes that appear to occur are actually new immutable objects that are created

# Function Overloading

---

# Overloading

---

In python, function/method overloading doesn't quite work like Java.

Since python is dynamically typed, we can't have a scenario like this

```
def add(int_a, int_b):  
    return int_a + int_b
```

```
def add(string_a, string_b):  
    return string_a + string_b
```

These are exactly the same method! We can't differentiate based on parameter data types.



# Overloading

Python functions are extremely powerful and flexible.

We can implement Function/Method Overloading in the following ways:

- Using Default Arguments
- Using Named Arguments
- Using Variable List Arguments
- Using Keyword Arguments (Later this week)

# But first! Positional Arguments

---

These are the standard functions we are used to.

- The parameters have fixed positions
- Parameters must be given in the order expected.

Eg:

```
def add_vectors(vector1, vector2):  
    x = vector1.x + vector2.x  
    y = vector1.y + vector2.y  
    z = vector1.z + vector2.z  
    return Vector(x, y, z)
```

```
add_vectors(position, velocity)
```

## Default Arguments

---

Just like Java and C++ we can allow our function parameters to have default arguments.

---

Default arguments need to be at the end of the argument list.

---

You cannot have a positional / non-default argument after a default argument.

---

The least likely default arguments should be to the left of the most likely default arguments.

# Default arguments

---

class Team:

```
def __init__(team_number, team_name = "", member_list=None):  
    self.team_number = team_number  
    self.name = team_name  
    self.members = member_list  
  
assignment1_team1 = Team(0, "Pugs 4 Life")  
assignment1_team2 = Team(1)  
assignment2_team3 = Team(2, "Tamagotchi App Developers", ["Jenkins",  
"Natalie", "Susan"])
```

# Remember! Python is interpreted!

---

What happens if we write a program like this?

Anything we provide to a default argument **is evaluated when the function is first interpreted, not when it is called**. That is, we **can't have dynamically generated default values**.

The empty list parameter is created once. (We'll look at scope in detail next week)

```
def print_message(message=[]):  
    message.append('Hi')  
    print(message)  
  
print_message()  
print_message()  
new_1 = ['new-1']  
print_message(new_1)  
new_2 = ['new-2']  
print_message(new_2)  
print_message()
```

# Remember! Python is interpreted!

---

```
def print_message(message=[]):  
    message.append('Hi')  
    print(message)
```

```
print_message()  
print_message()  
new_1 = ['new-1']  
print_message(new_1)  
new_2 = ['new-2']  
print_message(new_2)  
print_message()
```



[]

Default - message

A single message variable is created when code is interpreted

# Remember! Python is interpreted!

---

```
def print_message(message=[]):  
    message.append('Hi')  
    print(message)
```

```
print_message()  
print_message()  
new_1 = ['new-1']  
print_message(new_1)  
new_2 = ['new-2']  
print_message(new_2)  
print_message()
```



['Hi']

Default - message

The first function call appends 'Hi' to that single default instance of message

# Remember! Python is interpreted!

---

```
def print_message(message=[]):  
    message.append('Hi')  
    print(message)
```

```
print_message()  
print_message()  
new_1 = ['new-1']  
print_message(new_1)  
new_2 = ['new-2']  
print_message(new_2)  
print_message()
```

['Hi', 'Hi']

Default - message

The second function call appends another 'Hi' to that single default instance of message



# Remember! Python is interpreted!

---

```
def print_message(message=[]):  
    message.append('Hi')  
    print(message)
```

```
print_message()  
print_message()  
new_1 = ['new-1']  
print_message(new_1)  
new_2 = ['new-2']  
print_message(new_2)  
print_message()
```

['Hi', 'Hi']

Default - message

['new-1']

new\_1

new\_1 is a new list that is NOT using the default instance of message. New memory is allocated

# Remember! Python is interpreted!

---

```
def print_message(message=[]):  
    message.append('Hi')  
    print(message)
```

```
print_message()  
print_message()  
new_1 = ['new-1']  
print message(new_1)  
new_2 = ['new-2']  
print_message(new_2)  
print_message()
```

['Hi', 'Hi']

Default - message

['new-1', 'Hi']

new\_1 message

Message parameter is pointing at new\_1. Appends 'Hi' to new\_1 list

# Remember! Python is interpreted!

---

```
def print_message(message=[]):  
    message.append('Hi')  
    print(message)
```

```
print_message()  
print_message()  
new_1 = ['new-1']  
print_message(new_1)  
new_2 = ['new-2']  
print_message(new_2)  
print_message()
```

['Hi', 'Hi']

Default - message

['new-1', 'Hi']

new\_1

['new-2']

new\_2

new\_2 is a new list that is NOT using the default instance of message. New memory is allocated

# Remember! Python is interpreted!

---

```
def print_message(message=[]):  
    message.append('Hi')  
    print(message)
```

```
print_message()  
print_message()  
new_1 = ['new-1']  
print_message(new_1)  
new_2 = ['new-2']  
print_message(new_2)  
print_message()
```

['Hi', 'Hi']

Default - message

['new-1', 'Hi']

new\_1

['new-2', 'Hi']

new\_2      message

Message parameter is pointing at new\_2. Appends 'Hi' to new\_2 list

# Remember! Python is interpreted!

---

```
def print_message(message=[]):  
    message.append('Hi')  
    print(message)
```

```
print_message()  
print_message()  
new_1 = ['new-1']  
print_message(new_1)  
new_2 = ['new-2']  
print_message(new_2)  
print_message()
```

['Hi', 'Hi', 'Hi']

Default - message

['new-1', 'Hi']

new\_1

['new-2', 'Hi']

new\_2

Again using default parameter message, so 'Hi' appending to default message list

# Default arguments reminders

---

Be careful when using default arguments!

A new dynamic parameter is not generated every time the default parameter is used. The same default parameter is used every time

The behavior is different when operating on mutable vs immutable parameters in a function

[team\\_default\\_arguments.py](#)



# Named Arguments: Changing the Order

---

If we have named arguments, we can change the order of parameters in the function call by referring to its name.

# Positional arguments

---

```
def my_func(a_list, a_num, a_string):  
    print(a_list, a_num, a_string)
```

```
my_func(['cat', 'dog'], 5, 'hello')
```

A diagram illustrating the mapping of positional arguments. Three colored arrows point from the arguments in the function call to the corresponding parameters in the function definition. A blue arrow points from the list ['cat', 'dog'] to the parameter a\_list. A green arrow points from the integer 5 to the parameter a\_num. A yellow arrow points from the string 'hello' to the parameter a\_string.

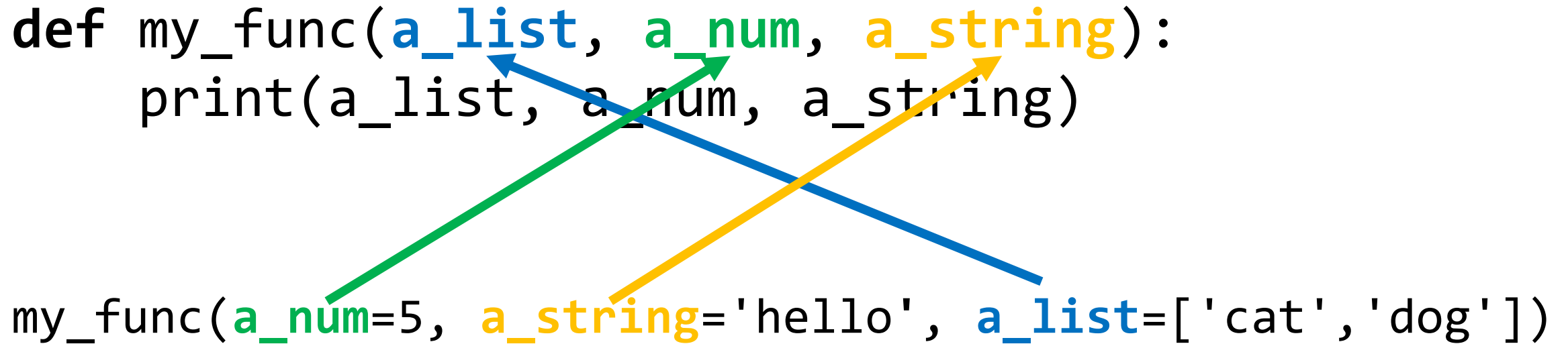
The order that parameters passed in matters with **positional arguments** (this is the one we're used to). Errors will occur otherwise



# Named Arguments: Changing the Order

---

```
def my_func(a_list, a_num, a_string):  
    print(a_list, a_num, a_string)  
  
my_func(a_num=5, a_string='hello', a_list=['cat', 'dog'])
```



The position of parameters that are passed in doesn't matter with named arguments. All that matters is explicitly assigning a value to a matching parameter name

# Named Arguments: Changing the Order

---

```
def my_func(a_list, a_num=0, a_string=' '):  
    print(a_list, a_num, a_string)
```

```
my_func(a_num=5, a_list=['cat', 'dog'])
```

A diagram consisting of two arrows. A green arrow originates from the 'a\_num=5' argument in the function call below and points to the 'a\_num=0' parameter in the function definition above. A blue arrow originates from the 'a\_list=['cat', 'dog']' argument in the function call below and points to the 'a\_list' parameter in the function definition above. This illustrates that named arguments can be passed in any order, regardless of their position in the function signature.

This is especially useful when functions have default parameters. Don't need to pass in all parameters. Only need to pass in parameters we care about

## Variable List Arguments

---

What if we wanted to provide a variable number of parameters?

---

E.g: `my_print("COMP3522", "is great", "but Thursdays are long")`

---

We could have the function expect a list and iterate through it.

---

Or maybe there is a better way to do this since python is cleaner and nicer?

# Pass unknown number of arguments

---

Can pass variable list of arguments into function using list

Wrap all arguments in list []

```
def my_print(list_of_items):  
    for item in list_of_items:  
        print(item)
```

```
my_print(['dog', 5, {1:'hi', 2:'hello'}])
```

# Variable Argument List - “Packing Arguments”

---

We specify an argument with an `*` preceding its name. The standard/convention is `*args`

This is known as **packing** arguments. All arguments are packed into a tuple named `args`

```
def my_print(*args):  
    for item in args:  
        print(item)
```

```
my_print('dog', 5, {1:'hi', 2:'hello'})
```

# Variable Argument List - “Unpacking”

---

The \* (asterisk) can be used with containers as well. This is known as **unpacking**

In the example below, the list is “unpacked” into the individual string, integer, and list objects

```
def my_print(str, int, list):  
    print(str, int, list)
```

```
my_list = ['dog', 5, {1: 'hi', 2: 'hello'}]  
my_print(*my_list) #unpack list
```

# Be Responsible!

---

1. If you need the caller to specify an argument, make it positional and mandatory (no default value).
2. Variable list arguments are extremely powerful and great to use. But it can be difficult to understand, read and maintain. The solution: **Comment your code!!!!**
3. Python is not like Java where you are expected to have lines and lines of comments. Python **demands** that you have meaningful comments **where necessary!**
4. From this week onwards, don't write comments because you have to. Imagine you are someone who has never seen your code. Write comments for that person.



# That's it for today!

---

Assignment 1 is up now! Due Feb 14<sup>th</sup> 11:59pm

Lab this week will get you started on  
Assignment 1

