# COMP10001 WORKSHOP #4

Shevon Mendis shevonm@unimelb.edu.au





## **Functions & Methods**

- In a *practical* sense, they are the same-they both execute a pre-defined set of instructions
- Methods can be thought to be "attached" to objects
  - When called, they work directly on the object that they are attached to

```
>>> "hello!!!".rstrip('!')
'hello'
```







- Data structure that stores elements in a particular order (ie. a sequence of elements)
- Defined using square brackets and different elements are separated using commas eg. ["Mewtwo", "Deoxys", "Smoochum", "Nicki Minaj"]

















```
>
```

```
>>> jonahs_favs = ["James Jonah Jameson", "Mary Jane", "Cardi B"]
>>> peters_favs = jonahs_favs
>>> peters_favs[0] = "Spider-Man"
>>> peters_favs
['Spider-Man', 'Mary Jane', 'Cardi B']
>>> jonahs_favs
['Spider-Man', 'Mary Jane', 'Cardi B']
```











ISTS

• lists are references!

```
>>> print(id(jonahs_favs) == id(peters_favs))
True
```

```
Recall id()

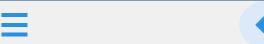
>>> help(id)
  Help on built-in function id in module builtins:
  id(obj, /)
    Return the identity of an object.

This is guaranteed to be unique among simultaneously existing objects.
```

(CPython uses the object's memory address.)









So how do we circumvent this issue?

```
>>> jonahs_favs = ["James Jonah Jameson", "Mary Jane", "Cardi B"]
>>> peters_favs = jonahs_favs.copy()
>>> peters_favs[0] = "Spider-Man"
>>> peters_favs
['Spider-Man', 'Mary Jane', 'Cardi B']
>>> jonahs_favs
['James Jonah Jameson', 'Mary Jane', 'Cardi B']
```

```
>>> print(id(jonahs_favs) == id(peters_favs))
False
```













- **Useful methods:** 
  - .append()
    - Adds a **single** element to the **end** of the **list**
    - Returns None
  - .pop()
    - Removes the element at a specified index (defaults to -1 if index unspecified)
    - Returns the removed element
  - .sort()
    - Sorts the list in place
    - Returns None
  - .copy()
    - Makes a copy of the list
    - Returns a **new** list





## tuples



- Data structure that stores elements in a particular order (ie. a sequence of elements)
- Defined using commas- the (circular) brackets only serve to improve readability
  - Defining an empty tuple:

```
my_tup = ()
```

Defining a tuple with a single element:

```
my_tup = (1,)
```

Defining a tuple with multiple elements:

```
my_tup = ("North", "South", "East", "West")
```







# Mutability



• lists are mutable- once we define it, we can change its contents

```
>>> my_list = [1, 2, 3]
>>> my_list[2] = 7
>>> my_list
[1, 2, 7]
```

tuples are immutable- once defined, we can't change its contents

```
>>> my_tuple = (1, 2, 3)
>>> my_tuple[2] = 7
Traceback (most recent call last):
   File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```





# Mutability



 However, it's actually the bindings that are unchangeable, not the objects they are bound to. In other words, if the tuple contains a mutable object, we can change that object

```
>>> my_tuple = ([1, 2, 3], 14)
>>> my_tuple[0][1] = 9
>>> my_tuple
([1, 9, 3], 14)
```







# Mutability



- Mutable data types:
  - o list
  - o set
  - o dict
- Immutable data types:
  - o bool
    - int
    - float
  - str
  - tuple











- In a programming content, iteration is the process of executing a block of code repeatedly (with a slight difference each time) until some condition has been fulfilled
- Python uses for loops and while loops for this







# for loops



- A for loop iterates over a collection of items
- Skeleton:

```
for < loop_variable > in < collection > :
    # do something
```

• eg. Print the numbers from 15 to 30

```
for num in range(15, 31):
    print(num)
```

eg. Print the elements in the list

```
for elem in [1, 3, 4, 7, 9]:
    print(elem)
```



# while loops



- A while loop runs as long as some condition is True
- Skeleton:

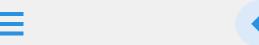
```
while < condition > :
    # do something
```

• eg. Print the numbers from 1 to 10

```
num = 1
while num <= 10:
    print(num)
    num += 1</pre>
```









- When deciding between the two:
  - If you know in advance how many times you want to iterate, whether that's through a collection or range of numbers, use a for loop
    - eg. Print out all the numbers in the range 0 to 50 inclusive
  - Otherwise use a while loop
    - eg. Keep asking the user to guess a certain word until their guess is correct
- Generally, we can convert any for loop into a while loop by making use of some additional variables. However, the converse is not true- there may be some while loops that run indefinitely until some action is made by the user, in which case you can't convert into a for loop as it runs a quantifiable number of times





#### for to while



eg. Print the elements in the list

```
movie_characters = [("Pilgrim", "Scott"), ("Flowers", "Ramona")]
# using a for loop
for character in movie_characters:
   print(character)
# using a while loop
index = 0
while index < len(movie_characters):</pre>
   print(movie_characters[index])
   index += 1
```





#### break



• We can prematurely *break* out a the loop by using a **break** statement

```
# prints the number of rolls of a die until your lucky number
# appears
LUCKY_NUM = 4
n_rolls = 0
while True:
    n_rolls += 1
    if (randint(1, 6) == LUCKY_NUM):
        break
print(n_rolls)
```





#### continue

• **continue** is similar to **break**, but instead of exiting the loop, it skips ahead to the next iteration

```
# prints the even numbers in the list
nums = [12, 3, 6, 7, 11, 33, 8]
for num in nums:
   if (num % 2 != 0):
        continue
   print(num)
```