COMP 1510
Autumn 2021

Week 04: Intro to data structures and repetition

Agenda for week 04

1. Documentation
2. Comments and docstrings for functions and modules including pre- and post-conditions
3. Testing code in our comments with doctests
4. Scope and the Python memory model (stack, heap, variables, references, addresses, objects, interning, identity is/id() vs equality ==)
5. Intro to data structures and containers
6. Lists
7. Working with lists
8. Membership operators in and not in
9. Slicing lists
10. Intro to repetition (looping) with the for-loop
11. The range function

COMMENTS AND DOCSTRINGS

Comments and docstrings

- **Code comments start with #**
- A comment helps someone reading our code
- Comments are ignored by the interpreter
- They are read by other developers
- Python programmers create comments for every module and function
- We have a special format and name: *docstrings* (it's a portmanteau aka a blending of the two words *document* and *string*)

Exactly what is a *docstring*

- A special kind of comment
- A *string literal* that describes a chunk of code
- Implemented as the first statement (or more!) inside each of these things:
  - Modules (source files)
  - Functions
  - Classes and method definitions (later this term).
- Used to create "official" documentation
- Wrapped in *triple quotes* (never use single quotes oh the humanity😱)

A good docstring consists of the following:

1. Short one-sentence description that begins with a verb in imperative tense and ends in a period
2. One blank line
3. Additional comments if the single line description is not sufficient (this is a good place to describe in detail how the function is meant to be used)
4. One blank line
5. PARAM statement (one for each parameter, in order!) which describes what the user should pass to the function
6. PRECONDITION statement for each precondition which the user promises to meet before using the function
7. POSTCONDITION statement for each postcondition which the function promises to meet if the precondition is met
8. RETURN statement which describes what will be returned from the function if the preconditions are met

Example:

```
def my_factorial(number):
    """Calculate factorial.

    A simple function that demonstrates good comment construction.

    :param number: a positive integer
    :precondition: number must be a positive integer equal to or
                   greater than zero
    :postcondition: calculates the correct factorial
    :return: factorial of number

    >>> my_factorial(0)
    1
    >>> my_factorial(1)
    1
    >>> my_factorial(5)
    120
    """
    return math.factorial(number)
```

How do we comment a module?

- Suppose we have a module called functions.py. The first statement in the file must be a triple-quote wrapped string, which will become the functions.py module's docstring when the file is imported and used by another project.

```
>>> import functions
>>> functions(__doc__)
>>> help(functions)
>>> help(functions.my_function)
```

DOCTESTS

Introducing doctest

- https://docs.python.org/3/library/doctest.html
- A module (like the math and random modules we talked about last week)
- Looks for pieces of text that look like interactive Python sessions in docstrings
- Executes the sessions to ensure they work exactly as shown
- Useful as very expressive documentation of the main use cases of a module

How do we use doctest?

1. Add these two lines to your code:

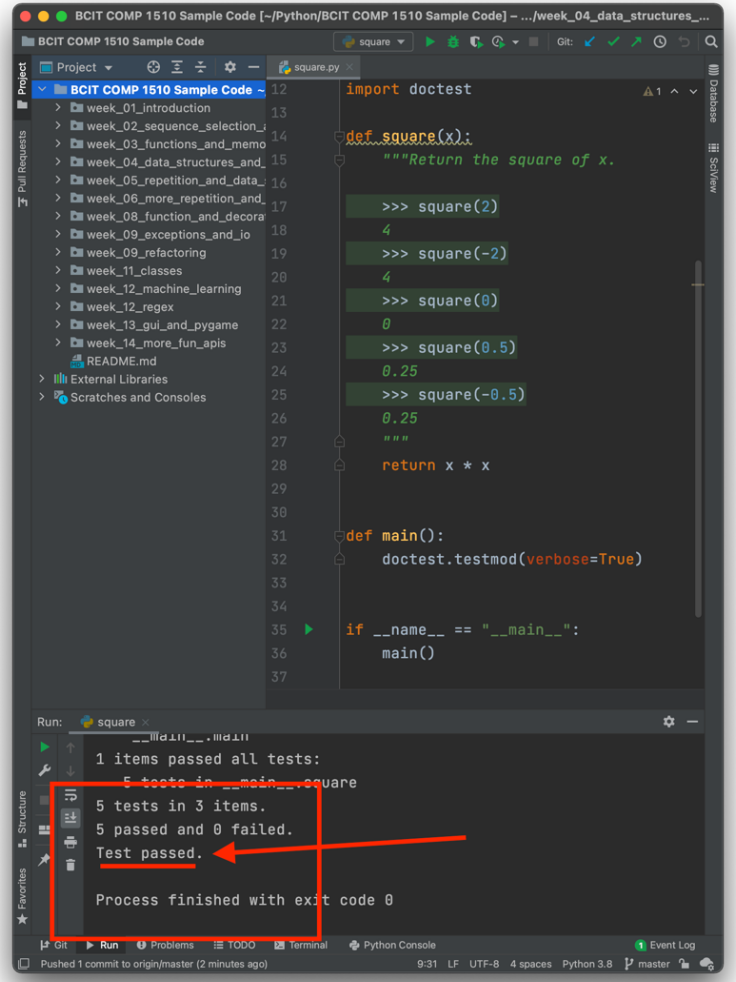
```
import doctest # at the top with other imports
doctest.testmod(verbose=True) # inside main
```

2. Modify your function docstring (see next slide for an example)

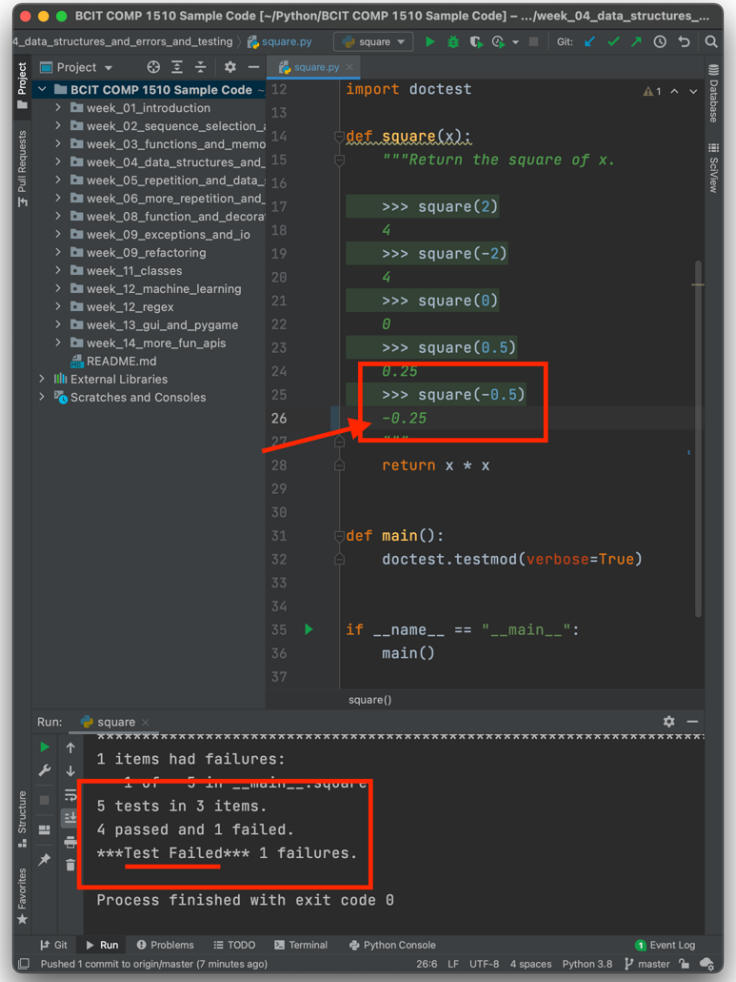
Sample doctest

```
def square(x):
    """Return the square of x.

    >>> square(2)
    4
    >>> square(-2)
    4
    """
    return x * x
```


BCIT COMP 1510 Sample Code [~/Python/BCIT COMP 1510 Sample Code] – .../week_04_data_structures_...
square.py
import doctest
def square(x):
"""Return the square of x.
>>> square(2)
4
>>> square(-2)
4
>>> square(0)
0
>>> square(0.5)
0.25
>>> square(-0.5)
0.25
"""
return x * x
def main():
doctest.testmod(verbose=True)
if __name__ == "__main__":
main()
Run: square
1 items passed all tests:
5 tests in 3 items.
5 passed and 0 failed.
Test passed.
Process finished with exit code 0

COMPARE


BCIT COMP 1510 Sample Code [~/Python/BCIT COMP 1510 Sample Code] – .../week_04_data_structures_...
square.py
import doctest
def square(x):
"""Return the square of x.
>>> square(2)
4
>>> square(-2)
4
>>> square(0)
0
>>> square(0.5)
0.25
>>> square(-0.5)
-0.25
"""
return x * x
def main():
doctest.testmod(verbose=True)
if __name__ == "__main__":
main()
square()
Run: square
1 items had failures:
5 tests in 3 items.
4 passed and 1 failed.
Test Failed 1 failures.
Process finished with exit code 0

SCOPE

Scope

- Q: Where is a variable visible?
- A: The name of a variable is only visible to part of a program
- This is called the variable's scope or **namespace**
- Scopes are “contexts” in which named references can be “accessed”
- We differentiate between different kinds of scope:
  1. Built-in
  2. Global
  3. Local

Built-in scope

- Anything from the built-in module has built-in scope
- The built-in scope contains all the names loaded into the Python variable scope when we fire up the interpreter
- They are pre-defined names
- This includes all the built-in functions: https://docs.python.org/3/library/functions.html#built-in-functions
- This includes all the built-in constants: https://docs.python.org/3/library/constants.html#built-in-constants
- When in the main module, __builtins__ is the built-in module builtins
- What gets printed: `dir(__builtins__)`

Global scope

- Defined at the top level of a module outside of a function
- **Scope extends from its assignment statement to the end of the module**
- Global variables can be accessed inside functions
- When accessing a global variable inside a function, we must use a global statement inside the function in order to change the value of the global variable.
- If we omit the global statement, the variable can be accessed as *read-only*.

Global variables can be used in functions

```
student_name = 'N/A'

def get_name( ):
    global student_name
    name = input('Enter student name: ')
    student_name = name

get_name( )
print('Student name: ', student_name)
```

Global variables used incorrectly

```
student_name = 'N/A'

def get_name( ):
    global student_name
    name = input('Enter student name: ')
    student_name = name # this is now a local variable!

get_name( )
print('Student name: ', student_name) # prints N/A
```

Local variables

- Defined inside the current function
- Invisible to the code outside the function
- Can only be used inside the function
- **A local variable's scope is limited to inside its function – when the function ends it is destroyed!**
- Scope begins when an object is bound to the variable name
- Scope ends at end of function

Local variable example

```
cm_per_inch = 2.54
inches_per_foot = 12

def height_US_to_cm(feet, inches):
    """ Convert a height in feet/inches to centimeters."""
    total_inches = feet * inches_per_foot + inches  # Total inches
    cm = total_inches * cm_per_inch
    return cm # returns the value to the calling function

feet = int(input('Enter feet: '))
inches = int(input('Enter inches: '))

print('Centimeters:', height_US_to_cm(feet, inches))
```

Example: Out of scope

```
cm_per_inch = 2.54
inches_per_foot = 12

def height_US_to_cm(feet, inches):
    """ Convert a height in feet/inches to centimeters."""
    total_inches = feet * inches_per_foot + inches  # Total inches
    cm = total_inches * cm_per_inch
    return cm

feet = int(input('Enter feet: '))
inches = int(input('Enter inches: '))

print('Total inches:', total_inches) # NO! OUT OF SCOPE!
```

Example: Out of scope

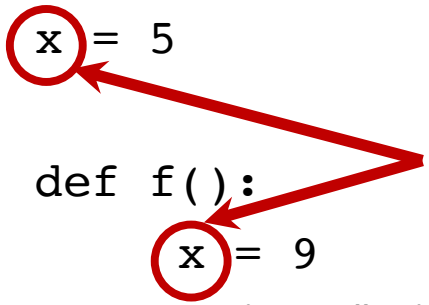
```
cm_per_inch = 2.54
inches_per_foot = 12

def height_US_to_cm(feet, inches):
    """ Convert a height in feet/inches to centimeters."""
    total_inches = feet * inches_per_foot + inches  # Total inches
    cm = total_inches * cm_per_inch
    return cm

feet = int(input('Enter feet: '))
inches = int(input('Enter inches: '))

print('Total inches:', cm) # NO! OUT OF SCOPE!
```

Problem # 1: What gets printed?



(This is actually called *shadowing*. I <u>*abhor*</u> this)

```
print("Before f: x =", x) # What gets printed?
f()
print("After f: x =", x) # What gets printed?
```

Problem # 2: What gets printed?

```
x = 5

def f():
     global x
     x = 9
     print("Within f: x =", x)

print("Before f: x =", x) # What gets printed?
f()
print("After f: x =", x) # What gets printed?
```

Problem # 3: What gets printed?

```
def f():
    global x
    print("Within f: x =", x)
    x = 9
    print("Within f: x =", x)

x = 5
print("Before f: x =", x) # What gets printed?
f()
print("After f: x =", x) # What gets printed?
```

Problem # 4: What gets printed (tricky)?

```
def f():
    print("Within f: x =", x)
    x = 9
    print("Within f: x =", x)

x = 5
print("Before f: x =", x) # What gets printed?
f()
print("After f: x =", x) # What gets printed?
```

Problem # 5: What gets printed (tricky)?

```
x = 5

def f():
     global x
     print("Within f: x =", x)
     y = 9
     print("Within f: y =", y)

print("Before f: x =", x) # What gets printed?
f()
print("After f: y =", y) # What gets printed?
```

Take-away points

- We *eschew global variables* in favour of encapsulated variables with narrow scope
- One form of encapsulation is to put all of our variables inside functions
- Every variable has an "owner" and a defined, limited scope
- *Minimize scope*
- This makes programming (and debugging and testing!) easier!
- (We avoid global variables in almost all programming situations – it's sloppy)

Whitespace and indentation

- ***Whitespace*** makes code easier to read
- Code that is easy to read is easier to maintain
- Code that is easier to maintain is cheaper to maintain
- Place one white space around operators
- Place two blank lines above and below functions
- Use indentation in functions
  - Every line of a function must be indented
  - That's how we can tell that the line of code 'belongs' to the function!

PYTHON MEMORY MODEL

What's going on under the hood?

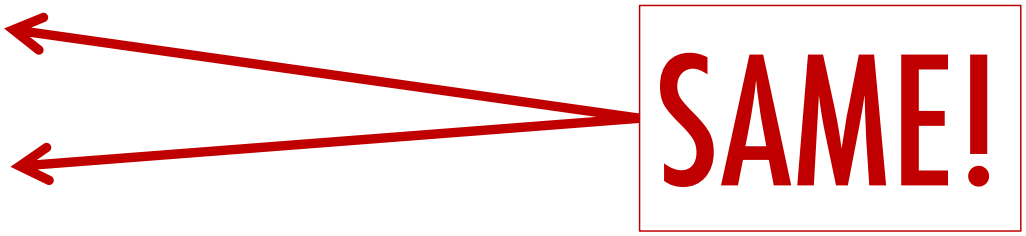
- Any time we store something like “Hello world” or 299792458 in our program, an “object” is created (instantiated), given a location in memory, and a variable is bound to it
- Recall can find the address of an object using the id( ) function:

```
a = 5
print(id(a))
b = “If all we have is a hammer, everything looks like a nail”
print(id(b))
```

Addresses in memory

- What happens if we do this:

```
a = 5
print(id(a)) # 4399780000
b = 5
print(id(b)) # 4399780000
```



- Are a and b bound to the same int object or are they bound to different int objects each containing a 5?

Python preloads and ***interns*** some objects

- Versions of Python differ, but some preload integers in the range [-5, 256]
- Type this in at the prompt of the interactive interpreter, and tell me what we get:

```
>>> nums = list(range(-10, 300))
>>> for i in nums:
...     print(i, id(i), id(i+1)-id(i))
```

Aliases and interning

- When two variables refer to the same address, we call them aliases
- Two Python variables that have been assigned the integer 1 both store the same address to the same integer object in memory
- **Aliases are variables that refer to the same object**
- Python optimizes memory and binds variable names to the same object when they are assigned the same value
- This is perfectly fine because the objects are ***IMMUTABLE***!

Variables

- ***Store references***: a variable is "bound to an object" – it stores an address (often called a reference) to an object in memory
- ***Are dynamic***: if we change the value assigned to one of the aliases, the variable is simply bound to a different object, and the original object is still in memory safely unchanged
- (A variable ca store references to many kinds of things over the course of a program, but shouldn't)
- (That's because a good variable name tells us what it points to all the time)

Memory tables and variable tables

- Let's represent computer memory using a **memory table**
- A memory table tells us what is stored where
- In Python we refer to data using variables
- Every variable is bound to an address
- We must visit that address to find out what the variable "stores"
- We can use a **variable table** to visualize what each variable refers to, i.e., what address it stores.

Let's establish some “table manners” for 1510

1. A variable name can refer to different addresses at different times (variables are dynamic)
2. An **integer value** is represented as 1510
3. An **address in memory** is represented as @1510

When the interpreter revs up

When Python starts up, the memory table looks something like this (the memory addresses are made up):

- We've already seen that some integers are stored in some addresses.
- For example, the integer 2 is stored at the memory address 4399779904.
- Remember: when we assign a value to a variable, we are really associating (binding) the variable to an address

Memory table	
Address	**Value**
4399779840	0
4399779872	1
4399779904	2
4399779936	3
4399779968	4
4399780000	5

Now let's create some integer objects

Let's execute this code:

```
a = 2
b = 4
```

Variable table	
Variable	**Value**
a	@4399779904
b	@4399779968

Memory table	
Address	**Value**
4399779840	0
4399779872	1
4399779904	2
4399779936	3
4399779968	4
4399780000	5

We can do this

```
>>> id(a)
439977904
>>> id(2)
439977904
```

```
>>> b = a
>>> id (b)
439977904
```

Variable table	
Variable	**Value**
a	@4399779904
b	@4399779968

Memory table	
Address	**Value**
4399779840	0
4399779872	1
4399779904	2
4399779936	3
4399779968	4
4399780000	5

What does this represent?

```
? = ???
? = ???
```

Variable table	
Variable	**Value**
n	@4399781184
greet	@4399783962

Memory table	
Address	**Value**
4399781184	42
4399781216	43
…	…
4399783962	“hello”
…	…
…	…

Try this

```
m = 10

def f():
    m = 12
    n = m
    # Fill in the
    # memory
    # diagram at
    # this
    # point

f()
print(m)
```

Globals	
Variable	**Value**

Locals	
Variable	**Value**

Memory table	
Address	**Value**
4399780160	10
4399780192	11
4399780224	12
4399780256	13
4399780288	14
4399780320	15

INTRO TO DATA STRUCTURES

Data structures

- We often process or manage more than one object at a time
- We like to arrange these objects into useful groups with useful relationships between the objects
- We need to be able to access and modify the group AND the individual objects in the group too!
- We assemble these useful groups of objects using *data structures*
- Data structures give us:
  1. A collection of values/objects
  2. A relationship between the values/objects
  3. Functions or operations that can be applied to the data.

Containers

- A **container** is a kind of data structure used to group related values together
- A container is a kind of compound (multi-part) type
- A container stores references (addresses) of other objects
- A *list* is a container created by surrounding a sequence of variables or literals with square brackets [ ]

LISTS

The Python List

- This line of code creates a new list variable my_list that "contains" two items, an integer and a string:

```
my_list = [10, 'abc']
```

- An item in a list is called an *element*
- A list is a *sequence*:
  - There is a beginning (front) and an end (back)
  - The contained elements are in order, and have a position aka index
  - The indices starts at zero (like strings)
- This is how we can create an empty list:

```
my_empty_list = []
```

About lists in Python…

- Lists are useful for reducing the number of variables in a program (a single list can store an entire collection of *related data*)
- Individual list elements can be accessed using their *index*

```
numbers = [1, 2, 3, 4]
print(numbers[3]) # this prints 4
```

- A list's index must be an integer
- The index cannot be anything else
- Trying to use something that is not an integer will produce a runtime error and the program will terminate.

Lists are mutable

- Lists are mutable, meaning that a programmer can *add and remove elements* from a list as needed
- List elements can also be updated with new values by performing an assignment to an index (position) in the list
- Lists have some helpful methods in Python like:
    1. list_name.append(value)
    2. list_name.pop(index)
    3. list_name.remove(value)

Creating a list

```
>>> no_colours = []
>>> some_colours = ['red', 'yellow', 'green']
>>> numbers = [1, 2, 3]
>>> grades = [90.0, 85.5, 78.5]
```

1. Assignment operator
2. Square brackets
3. (Possibly empty) comma-separated list of elements.

Printing a list

```
>>> colours = ['red', 'yellow', 'blue']
>>> print(colours)
['red', 'yellow', 'blue']
>>> print(colours[0])
red
>>> print(colours[3])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

Changing a list

```
>>> grades = [90.0, 85.5, 78.5]
>>> a_grade = grades[0]
>>> print(a_grade)
90.0
>>> grades[0] = 49.9
>>> print(a_grade)
90.0
>>> print(grades)
[49.9, 85.5, 78.5]
```

Appending to a list

```
>>> students = []
>>> students
[]
>>> students.append('Felix')
>>> students
['Felix']
>>> students.append('Lexie')
>>> students
['Felix', 'Lexie']
>>> students[2] = 'Evon' # Whoops nope, that's too crazy for Python!
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list assignment index out of range
```

Inserting elements into a list

```
>>> students.insert(0, 'Aamir')
>>> students
['Aamir', 'Felix', 'Lexie']
>>> students.insert(1, 'Josh')
>>> students
['Aamir', 'Josh', 'Cody', 'Lexie']
>>> students.insert(4, 'Kim')
>>> students
['Aamir', 'Josh', 'Cody', 'Lexie', 'Kim']
>>> students.insert(10, 'Alex')
>>> students
['Aamir', 'Josh', 'Cody', 'Lexie', 'Kim', 'Alex']
```

Removing elements from a list

```
>>> colours
['red', 'yellow', 'green']
>>> colours.append('blue')
>>> colours
['red', 'yellow', 'green', 'blue']
>>> colours.append('purple')
>>> colours
['red', 'yellow', 'green', 'blue', 'purple']
>>> a_colour = colours[4]
>>> a_colour
'purple'
>>> del colours[4]
>>> a_colour
'purple'
>>> colours
['red', 'yellow', 'green', 'blue']
>>> del colours[3]
>>> colours
['red', 'yellow', 'green']
```

Popping elements from a list

```
>>> colours
['red', 'yellow', 'green', 'periwinkle']
>>> popped = colours.pop()
>>> popped
'periwinkle'
>>> colours
['red', 'yellow', 'green']
>>> popped = colours.pop()
>>> popped
'green'
>>> colours
['red', 'yellow']
```

```
>>> popped = colours.pop(0)
>>> popped
'red'
>>> colours
['yellow']
```

When do I pop and when do I del?

- If you don't want to use an element delete it from the list (and memory!) and move on
- If you want use an element after you remove it from a list, pop it off the list and assign the return value to a variable

Removing an element by value

```
>>> colours = ['red']
>>> colours
['red']
>>> colours.append('red')
>>> colours
['red', 'red']
>>> colours.remove('red')
>>> colours
['red']
>>> colours.remove('red')
>>> colours
[]
>>> colours.remove('red')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: list.remove(x): x not in list
>>> colours = ['red']
>>> colours
['red']
>>> colours.remove('blue')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: list.remove(x): x not in list
>>>
```

We can organize our list

- We can sort
- There are dozens (hundreds?) of useful sort algorithms
- We will learn some later this term

•Python knows how to sort!

- Use the list's sort( ) method to efficiently sort a list

Sorting list of ints, floats, and strings

```
>>> numbers = [6, 2, 9]
>>> numbers
[6, 2, 9]
>>> numbers.sort()
>>> numbers
[2, 6, 9]
>>> numbers = [0.0, -33.33, -33.33]
>>> numbers.sort()
>>> numbers
[-33.33, -33.33, 0.0]
>>> colours = ['red', 'yellow', 'blue']
>>> colours
['red', 'yellow', 'blue']
>>> colours.sort()
>>> colours
['blue', 'red', 'yellow']
```

Python can generate a separate sorted copy

```
>>> numbers = [2, 99, -33]
>>> number = numbers[0]
>>> numbers
[2, 99, -33]
>>> number
2
>>> sorted_numbers = sorted(numbers) # global sorted function make a copy
>>> numbers
[2, 99, -33]
>>> number
2
>>> sorted_numbers
[-33, 2, 99]
```

More list behaviours

- We can find the length of a list by passing a list to the global `len( )` function
- We can concatenate lists using the `+` operator
- We can reverse a list by invoking the `reverse( )` method on a list
- We can make lists of lists (of lists of lists of …)
- Look for the Python.org documentation about the list.

Be careful: functions vs methods

- Some built-in global functions operate on sequences like lists and strings
- Examples include len( ), sorted( )

- Methods are special functions built into the definitions of data types like lists and strings
- Examples include sort( ), append( ), clear( ), insert( ), pop( ), remove( ), and reverse( )
- *Question*: How can we differentiate a function from a method?

Lists are not homogenous

- Lists can contain mixed types of objects
- For example, x = [1, 2.5, 'abc'] creates a new list x that contains an integer, a floating-point number, and a string
- In some languages, this is considered ***heresy***
- Challenge: print out the addresses (using the id function) of some elements in a list. Where are they located in relation to one another?

Do you remember int(), float(), and str()

- There is a list( ) function too!
- Accepts an 'iterable' argument (a string, list, or tuple)
- Returns a new list object:

```
>>> letters = list('Python > JavaScript')
>>> letters
['P', 'y', 't', 'h', 'o', 'n', ' ', '>', ' ',
'J', 'a', 'v', 'a', 'S', 'c', 'r', 'i', 'p',
't']
```

Recall that lists are mutable

- We can add and remove elements
- We call this in-place modification
- We don't have to create a new object, the list grows and shrinks for us
- In-place modification affects any variable that refers to the list:

```
my_teams = ['Canadiens', 'Sabres', 'Nordiques']
your_teams = my_teams  # Create an alias for the list
my_teams[1] = 'Oilers'  # Modify list element
print('My teams are:', my_teams)  # What gets printed?
print('Your teams are:', your_teams)  # What gets printed?
```

MEMBERSHIP OPERATOR

Membership operators

- A common task is to determine if a container contains a specific value
- Python has membership operators in and not in
- These operators return true if the left operand matches the value of some element in a container

```
>>> colours = ['red', 'white', 'cerulean']
>>> print('cerulean' in colours)
>>> print('magenta' not in colours)
```

When to use membership operators

- Can be used with sequence types (we have studied str and list so far):
- We can check if a list contains something
- We can also use to determine whether a string is a **substring,** or a matching subset of characters, in a larger string:

```
request_str = 'GET index.html HTTP/1.1'
if '/1.1' in request_str:
    print('HTTP protocol 1.1')
if 'HTTPS' not in request_str:
    print('Unsecured connection')
```

SLICING LISTS

"Slicing" a list

- We can access a single element in a list using its index and square brackets
- 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 is easy as slicing a pie

```
>>> letters = list('Hello world')
```

- Omitting the first index starts at the beginning

```
>>> print(letters[ :5])
['H', 'e', 'l', 'l', 'o']
```

- Omitting the second index ends at the end

```
>>> print(letters[6:])
['w', 'o', 'r', 'l', 'd']
```

REPETITION WITH THE FOR LOOP

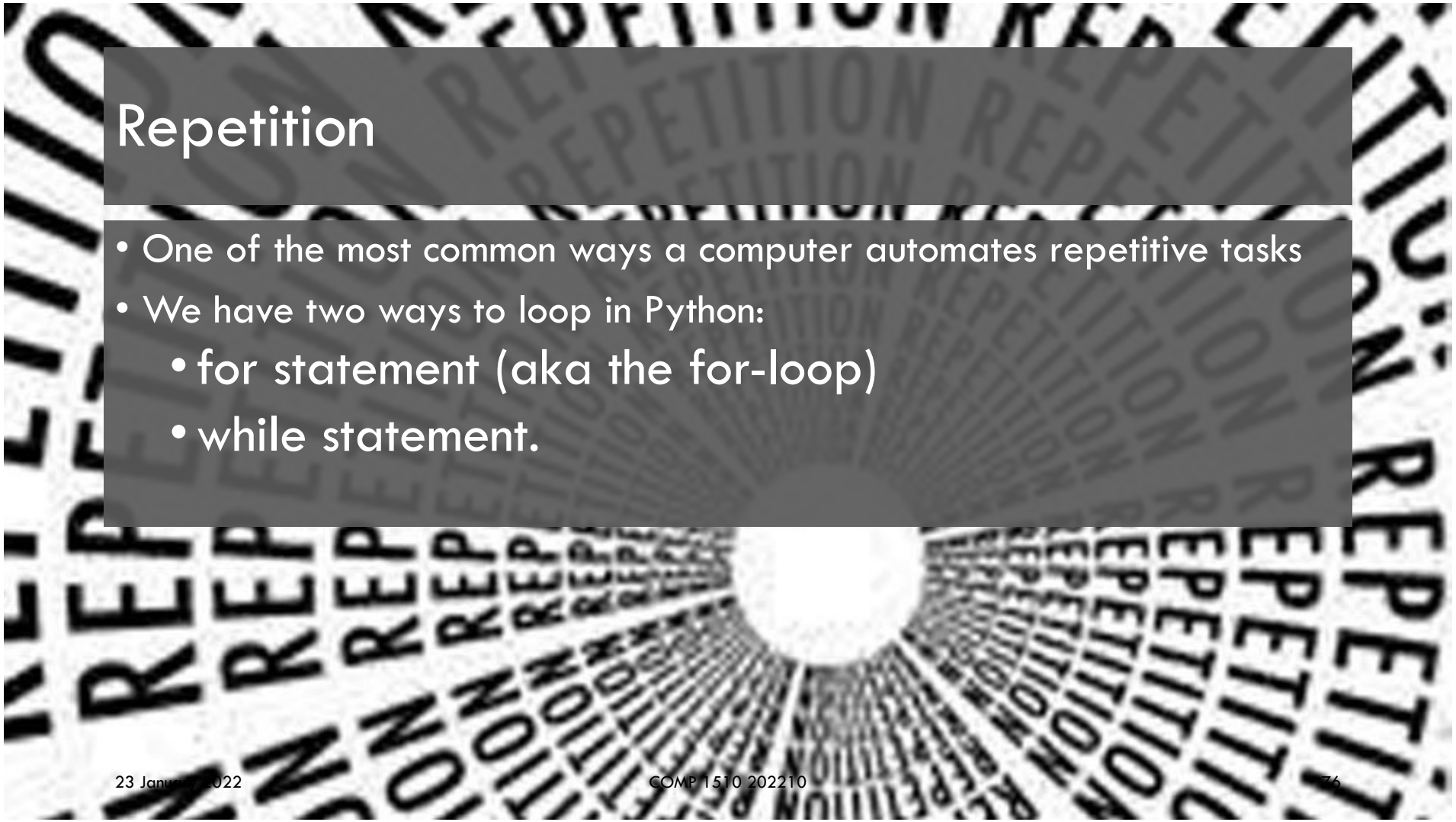
Control structure review

- We've learned about:
    1. Sequence
    2. Selection
    3. Indirection (functions)

- The only control structure left is:
    4. Repetition (aka looping)

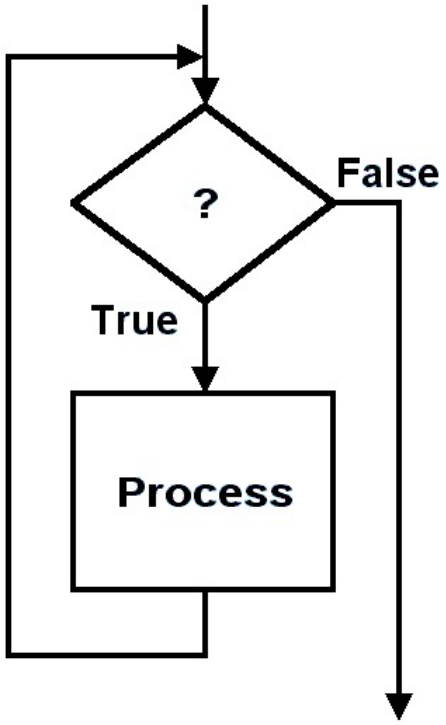
Repetition

- One of the most common ways a computer automates repetitive tasks
- We have two ways to loop in Python:
  - for statement (aka the for-loop)
  - while statement.



Repetition aka iteration

- When we want to repeat a command
- Counting
- **Looping**
- While X is true, do Y
- For each A in the group called B, do C
- Do S while T is less than W
- Then we continue with the sequence of commands…



Introducing the for-loop

```
winner_list = [ 'andy', 'bart', 'cinnamon' ]
for winner in winner_list:
    print(winner)
```

```
andy
bart
cinnamon
```

We say: “For every winner in winner_list, print the winner”

Rules for the for-loop

- The **loop body** is repeated once for each element in the list
- The loop body is **indented,** and can be any number of lines
- There is no maximum loop size or list size
- We can use any name we want for the **repeating variable,** but it's best to use a **singular name**
- During each loop, the repeating variable is **bound** for that loop to the address of the next element in the list
- When the loop reaches the end of the list, the loop **ends gracefully**
- **Don't forget the colon :**

RANGE FUNCTION

What's a range object, exactly?

- 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.*

More about the range data type

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

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

Range example code

```
>>> r = range(0, 20, 2)
>>> r
range(0, 20, 2)
>>> 11 in r
False
>>> 10 in r
True
>>> r.index(10)
5
>>> r[5]
10
>>> r[:5]
range(0, 10, 2)
>>> r[-1]
18
```

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))
[]
```

We can find the min and the max easily

```
>>> digits = list(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 > JavaScript')
>>> sum(letters) # <-- <-- Will this work?
```

We know about three kinds of sequences now

- These are **not the same thing:**
  1. str (the string)
  2. list
  3. range
  4. (there is a fourth we will look at soon called the tuple)
- They are stored differently and are treated differently
- **Only <u>some</u> sequence operations are common to all**

https://docs.python.org/3/library/stdtypes.html#sequence-types-list-tuple-range

Common sequence operations*

Operation	Result
`x in s`	`True` if an item of *s* is equal to *x*, else `False`
`x not in s`	`False` if an item of *s* is equal to *x*, else `True`
`s + t`	the concatenation of *s* and *t*
`s * n` or `n * s`	equivalent to adding *s* to itself *n* times
`s[i]`	*i*th item of *s*, origin 0
`s[i:j]`	slice of *s* from *i* to *j*
`s[i:j:k]`	slice of *s* from *i* to *j* with step *k*
`len(s)`	length of *s*
`min(s)`	smallest item of *s*
`max(s)`	largest item of *s*
`s.index(x[, i[, j]])`	index of the first occurrence of *x* in *s* (at or after index *i* and before index *j*)
`s.count(x)`	total number of occurrences of *x* in *s*

* Note that the range don't support sequence concatenation or repetition

https://docs.python.org/3/library/stdtypes.html#mutable-sequence-types

Mutable sequence ops

We've seen most of these

The ones we haven't seen include some interesting methods like copy which creates a *shallow copy*

A *shallow copy* does **NOT** duplicate the objects in the sequence…

Operation	Result
`s[i] = x`	item *i* of *s* is replaced by *x*
`s[i:j] = t`	slice of *s* from *i* to *j* is replaced by the contents of the iterable *t*
`del s[i:j]`	same as `s[i:j] = []`
`s[i:j:k] = t`	the elements of `s[i:j:k]` are replaced by those of *t*
`del s[i:j:k]`	removes the elements of `s[i:j:k]` from the list
`s.append(x)`	appends *x* to the end of the sequence (same as `s[len(s):len(s)] = [x]`)
`s.clear()`	removes all items from *s* (same as `del s[:]`)
`s.copy()`	creates a shallow copy of *s* (same as `s[:]`)
`s.extend(t)` or `s += t`	extends *s* with the contents of *t* (for the most part the same as `s[len(s):len(s)] = t`)
`s *= n`	updates *s* with its contents repeated *n* times
`s.insert(i, x)`	inserts *x* into *s* at the index given by *i* (same as `s[i:i] = [x]`)
`s.pop([i])`	retrieves the item at *i* and also removes it from *s*
`s.remove(x)`	remove the first item from *s* where `s[i]` is equal to *x*
`s.reverse()`	reverses the items of *s* in place

That's it for week 04!

That wasn't so bad, was it?