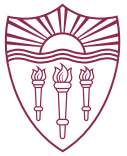
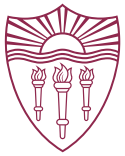


DSCI 510
PRINCIPLES OF PROGRAMMING FOR DATA SCIENCE

Itay Hen



THE FOR LOOP

for: the definite loop

- Quite often we have a list of items we'd like to operate on in order, such as the lines of a file
- Python has a construct that runs a block of code once on each element of a list; it's called a for loop
- Such a loop is a definite loop, because it executes the block of code a predefined number of times
- A definite loop iterates through a collection of objects, like the elements of a set, of a bag, of a list, the characters on a string, etc. (anything that is "iterable").

A Simple Definite Loop

```
for i in range(5, 0, -1):
    print(i)
print("Blastoff!")
```

```
5
4
3
2
1
Blastoff!
```

- range([start], stop, [step])
- start (optional): starting value of the sequence. Defaults to 0
- stop (required): The ending value of the sequence (exclusive). The sequence will run up to, but not include, this value
- step (optional): The increment between values in the sequence. Defaults to 1
- range returns an iterable object, which is often converted to a list when used in a loop

A Definite Loop over a list of Strings

```
friends = ['Joseph', 'Glenn', 'Sally']
for friend in friends:
    print(f'Happy New Year: {friend}')
print('Done!')
```

```
Happy New Year: Joseph
Happy New Year: Glenn
Happy New Year: Sally
Done!
```

- Definite loops (for loops) have explicit iteration variables that change each time through a loop.
- These iteration variable moves through the sequence or set
- The iteration variable in this for loop: friend
- It value changes from ‘Joseph’ to ‘Glenn’ to ‘Sally’

Another example:

```
>>> for char in "abcde":
...     print(char)
...
a
b
c
d
e
```

Find Largest Number in a List Containing Positive Numbers

numbers = [3, 41, 12, 9, 74, 15]

- How would you find the largest number?
  - Think of some variable to set initially
  - Go through the numbers, changing that variable if necessary
  - Check the variable's final value
- What would that variable be?

Code: Find Largest Number in a list of Positive Numbers

```
numbers = [3, 41, 12, 9, 74, 15]
largest_so_far = -1
for number in numbers:
    if number > largest_so_far:
        largest_so_far = number
print(f'Largest number was: {largest_so_far}')
```

Find Average of Numbers in a List

numbers = [14, 3, 2, 25, 8, 10]

- How would you find the average?
- Remember:
  - Think of some variable(s) to set initially
  - Go through the numbers, changing variable(s) if necessary
  - Check the variable(s) final value
- What would those variables be?

Accumulator Pattern: Find Average of Numbers in a List

```
numbers = [14, 3, 2, 25, 8, 10]
count = 0
sum_so_far = 0
for number in numbers:
    count += 1
    sum_so_far += number
average = sum_so_far / count
print(f"There were {count} numbers")
print(f"Their average was {average}")
```

```
There were 6 numbers
Their average was 10.333333333333334
```

Search Pattern Using a Boolean Variable: Search for 3

```
numbers = [14, 3, 2, 25, 8, 10]
found = False
print(f'Before: {found}')
for number in numbers:
    if number == 3:
        found = True
    print(f'Found: {number}')
print(f"After: {found}")
```

How could this be improved?

```
Before: False
Found: 14
Found: 3
Found: 2
Found: 25
Found: 8
Found: 10
After: True
```

Search Pattern Using a Boolean Variable: Search for 3

```
numbers = [14, 3, 2, 25, 8, 10]
found = False
print(f'Before: {found}')
for number in numbers:
    if number == 3:
        found = True
        break  # skip the rest of the list
    print(f'Found: {number}')
print(f"After: {found}")
```

```
Before: False
Found: 14
After: True
```

Find the smallest value of a list of arbitrary integers

```
numbers = [3, 41, 12, 9, 74, 15]
smallest_so_far = ???
for number in numbers:
    if number < smallest_so_far:
        smallest_so_far = number
print(f"Smallest number is: {smallest_so_far}")
```

What can we set smallest_so_far to ?

Maybe None
or perhaps float('inf')

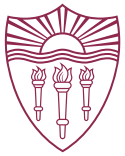
None

- None is a special object and a built-in constant that represents the absence of a value
  - Semantically, None means undefined.
  - But Syntactically, None is defined:
    - None can used in assignments and expressions
- None is its own data type in Python: NoneType. None is the only member of NoneType
- In a Boolean expression, None is treated as False (if my_var is None)
- A function without a return statement returns None
- None is often used as default value of function arguments when you want to allow the caller to omit that argument.
- When working with None, be careful not to use it in situations where you expect an actual value. Trying to perform operations on a None value can result in runtime errors, so it's important to handle None appropriately in your code.

Find the smallest value of a list of arbitrary integers

```
numbers = [3, 41, 12, 9, 74, 15]
smallest_so_far = None
for number in numbers:
    if smallest_so_far is None or number < smallest_so_far:
        smallest_so_far = number
print(f"Smallest number is: {smallest_so_far}")
```

```
Smallest number is: 3
```



FUNCTIONS

Functions

- Often you want to repeat blocks of code in your program
  - For example, you may need to convert temperature in Fahrenheit to Celsius multiple times
- It would be wasteful to have to include the same code multiple times
- A function may take *input(s)* and may provide *output(s)*

Examples of Functions

- We've already seen several *built-in* functions, e. g.,
  - `int('12'), input("> ")`
  - `type("string"), float(6)`
  - Do they take input(s)? Do they have output?
- There are many other built-in functions, for instance:
  - `max(), min()`

Python Functions

- There are two kinds of functions in Python.

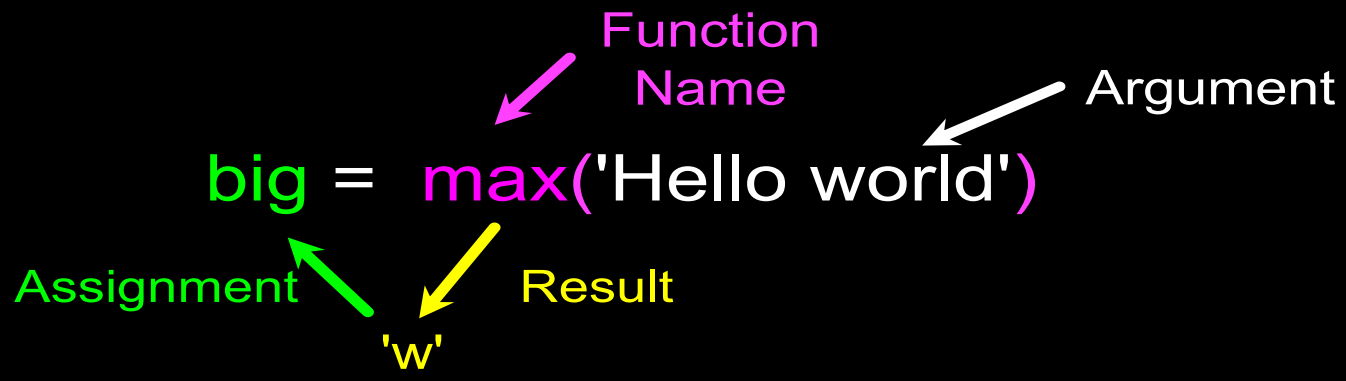
  - Built-in functions that are provided as part of Python - print(), input(), type(), float(), int() ...

  - Functions that we define ourselves and then use

- We treat function names as "new" reserved words (i.e., we avoid them as variable names)

Function Definition

- In Python a function is some reusable code that takes arguments(s) as input, does some computation, and then returns a result or results
- We define a function using the def reserved word
- We call/invoke the function by using the function name, parentheses, and arguments in an expression




```
>>> big = max('Hello world')
>>> print(big)
w
>>> tiny = min('Hello world')
>>> print(tiny)

>>>
```

Building our Own Functions

- We create a new function using the def keyword followed by parentheses, with optionally parameters inside the parenthesis
- We indent the body of the function
- This defines the function, but does not execute the body of the function

```
def print_lyrics():
    print("I'm a lumberjack, and I'm okay.")
    print('I sleep all night and I work all day.')
```

Arguments

- An argument is a value we pass into the function as its input when we call the function
- We use arguments so we can direct the function to do different kinds of work when we call it at different times
- We put the arguments in parentheses after the name of the function

```
big = max('Hello world')
```

Argument

Parameters

A parameter is a variable which we use in the function definition. It is a "handle" that allows the code in the function to access the arguments for a particular function invocation.

```
>>> def greet(lang):
...     if lang == 'es':
...         print('Hola')
...     elif lang == 'fr':
...         print('Bonjour')
...     else:
...         print('Hello')
...
>>> greet('en')
Hello
>>> greet('es')
Hola
>>> greet('fr')
Bonjour
>>>
```

Return Values

Often a function will take its arguments, do some computation, and return a value to be used as the value of the function call in the calling expression. The return keyword is used for this.

```
def greet():
    return "Hello"

print(greet(), "Glenn")
print(greet(), "Sally")
```

```
Hello Glenn
Hello Sally
```

Return Value

- A “fruitful” function is one that produces a result (or return value)
- The return statement ends the function execution and “sends back” the result of the function

```
>>> def greet(lang):
...     if lang == 'es':
...         return 'Hola'
...     elif lang == 'fr':
...         return 'Bonjour'
...     else:
...         return 'Hello'
...
>>> print(greet('en'),'Glenn')
Hello Glenn
>>> print(greet('es'),'Sally')
Hola Sally
>>> print(greet('fr'),'Michael')
Bonjour Michael
>>>
```

Multiple Parameters / Arguments

- We can define more than one parameter in the function definition
- We simply add more arguments when we call the function
- We match the number and order of arguments and parameters

```
def addtwo(a, b):
    added = a + b
    return added

x = addtwo(3, 5)
print(x)
```

```
8
```

Void (non-fruitful) Functions

- When a function does not return a value, we call it a "void" function
- Functions that return values are "fruitful" functions
- Void functions are "not fruitful"

Functions Must Be Defined Before Use

- Built-in functions are all predefined and may be used at any time
- Python has many modules that contain function definitions — you must *load* those modules before using their functions
  - See section 4.5 in the book for how to get many math functions
- You must define your own functions before you can use them

- Also, functions have their own type:

```
>>> type(print)
<class 'builtin_function_or_method'>
```

To function or not to function…

- Organize your code into “paragraphs” - capture a complete thought and “name it”
- Don’t repeat yourself - make it work once and then reuse it
- If something gets too long or complex, break it up into logical chunks and put those chunks in functions
- Make a library of common “tasks” that you do over and over - perhaps share this with your friends...

Variable Scope

Each variable defined in a program has a specific region of the program from which it can be accessed (or modified)

- Global variables: can be accessed anywhere
- Local variables: only in a specific block of code
- Function parameters: only within the function definition

Go to IDLE…
variable scope.py

Good Programming Style

- Changes to the global state (data) of the program from inside a function are called "side-effects"

- Minimize side-effects
  - Makes program more understandable and maintainable, since this keeps changes in the local scope

A word about *methods*

- A **method** is just a **function that belongs to an object** (i.e., tied to a specific class).
- In practice, a method is a function **defined inside a class** that you call using **dot notation**.
- Example:

```python
s = "hello"
print(s.upper())     # "HELLO"
print(s.rstrip("o")) # "hell"
```

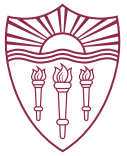
Here, `upper()` and `rstrip()` are **methods of the** `str` **class**.

So:

- Function = defined independently (`len(s)`)
- Method = function tied to an object (`s.upper()`)

- We will talk about this more when we discuss objects and classes.

Documenting Functions: Doc strings



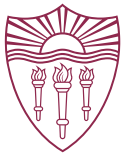
```python
def greet(name):
    """
    Greet the person passed as an argument.

    Parameters:
        name (str): The name of the person to greet.

    Returns:
        str: A greeting message including the person's name.
    """
    return f"Hello, {name}!"
```

```python
print(greet("Alice"))
# Output: Hello, Alice!
```

Documenting Functions: Doc strings



```
help(greet)
```

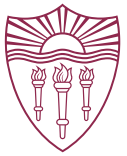
```
Help on function greet in module __main__:

greet(name)
    Greet the person passed as an argument.

    Parameters:
        name (str): The name of the person to greet.

    Returns:
        str: A greeting message including the person's name.
```

Keywords-only Parameters



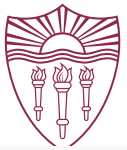
```
>>> def f(a, *, b):
...     return a // b, a % b
...
>>> f(4, 3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: f() takes 1 positional argument but 2 were given
>>>
>>>
>>> f(5, b=2)
(2, 1)
>>>
```

- Feature of Python 3
- To specify keyword-only arguments when defining a function, name them after the argument prefixed with *
- If you don't want to support variable positional arguments but still want keyword-only arguments, put a * by itself in the signature.

- Example:

```
def move(x, y, *, speed=1, direction="north"):
    print(f"Moving to ({x},{y}) at speed {speed} towards {direction}")
```

Keywords-only Parameters



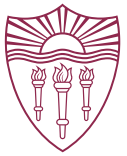
- Example:

```
def move(x, y, *, speed=1, direction="north"):
    print(f"Moving to ({x},{y}) at speed {speed} towards {direction}")
```

- x and y are **positional-or-keyword**
- speed and direction are **keyword-only**

```
>>> move(4,5,5,6)
Traceback (most recent call last):
  File "<python-input-9>", line 1, in <module>
    move(4,5,5,6)
    ~~~~^^^^^^^^^
TypeError: move() takes 2 positional arguments but 4 were given
>>> move(4,5,speed=4,direction="north")
Moving to (4,5) at speed 4 towards north
>>> move(4,5,direction="north",speed=4)
Moving to (4,5) at speed 4 towards north
>>> move(y=5,x=4,speed=4,direction="south")
Moving to (4,5) at speed 4 towards south
```

Positional-Only Parameters



```
>>> def f(a, b, /):
...     return a// b, a % b
...
>>> f(a=34, b=13)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: f() got some positional-only arguments passed as keyword arguments: 'a, b'
>>> f(34, 13)
(2, 8)
>>>
```

- To define a function requiring positional-only parameters, use a / in the parameter list
- All arguments to the **left** of / are positional-only

Higher-Order Functions

- A function that takes a function as an argument or returns a function as a result is called a `higher-order function'.
- len() as an argument

```
>>> fruits = ['strawberry', 'apple', 'cherry', 'fig', 'mango', 'banana', 'raspberry']
>>> sorted(fruits, key=len)
['fig', 'apple', 'mango', 'cherry', 'banana', 'raspberry', 'strawberry']
>>>
```

- User defined function as argument to the sorted() function

```
>>> def reversed(word):
...     return word[::-1]
...
>>> sorted(fruits, key=reversed)
['banana', 'apple', 'fig', 'mango', 'raspberry', 'strawberry', 'cherry']
>>>
```

Anonymous Functions

- The lambda keyword creates an anonymous (nameless) function within a Python expression.
- The body of a lambda function is limited to being a pure expression.
- Instead of

```
def add(x, y):
    return x + y
```

- Use:

```
add = lambda x, y: x + y
print(add(2, 3))  # Output: 5
```

Anonymous Functions

- The best use of anonymous functions is in the context of an argument list for a higher-order function.

```
>>> sorted(fruits, key=lambda word: word[::-1])
['banana', 'apple', 'fig', 'mango', 'raspberry', 'strawberry', 'cherry']
>>>
```

- It's a "throw-away function".