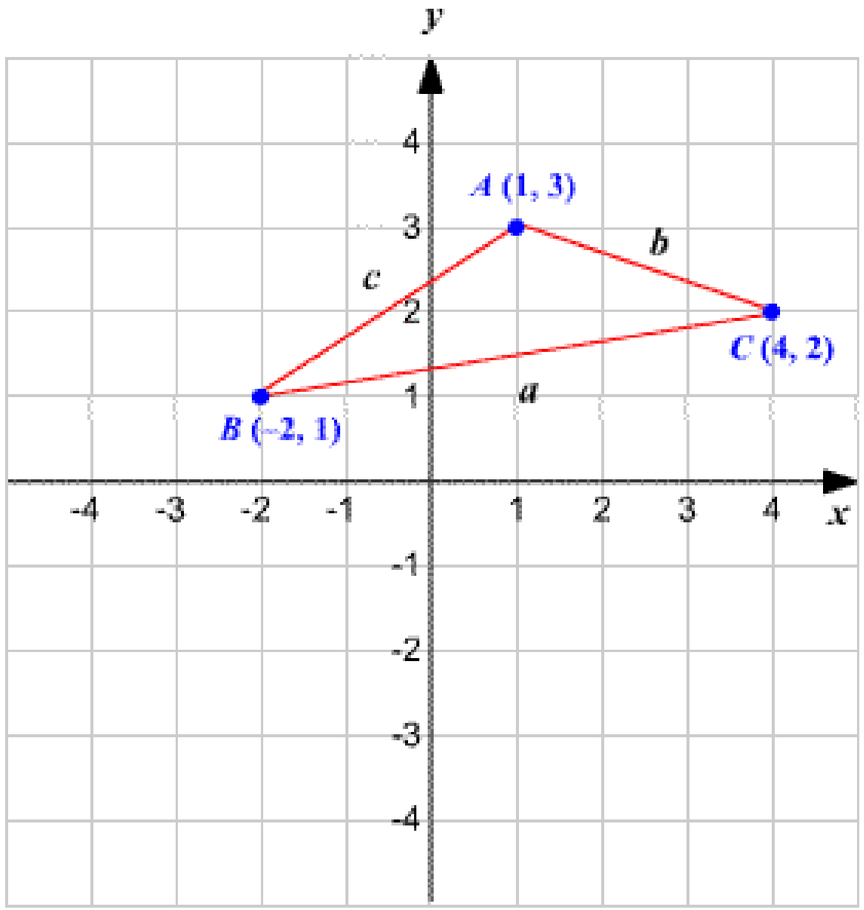
TECHNION

Azrieli Continuing Education and External Studies Division

Module 5.5: Functions

Why do we need functions?

Let’s say we have a triangle…



- We’ll save the points using packing/unpacking, that we learned last lesson.

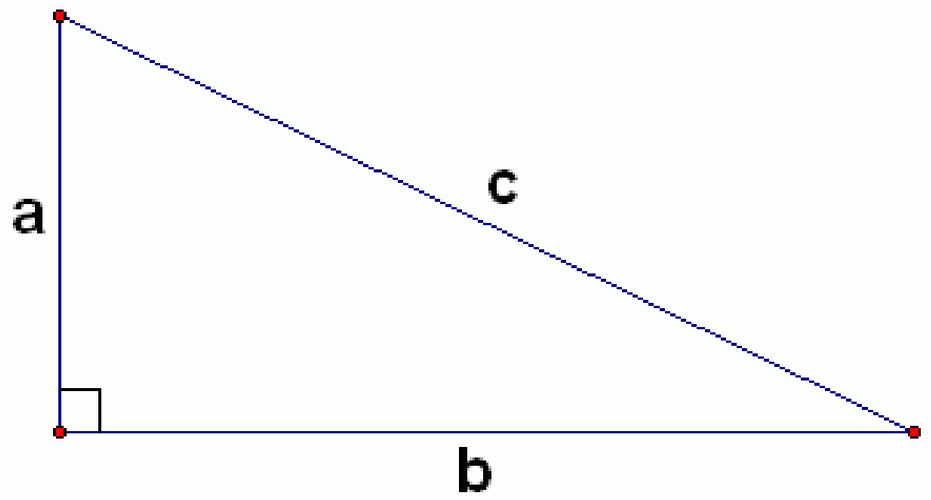
```
In [1]: ax, ay = 1, 3

In [2]: bx, by = -2, 1

In [3]: cx, cy = 4, 2
```

I want to know the triangle's perimeter…

- We'll do this by using the Pythagorean Theorem.



$$a^2 + b^2 = c^2$$

Let's code!

```
In [4]: side1 = ((ax - bx) ** 2 + (ay - by) ** 2) ** 0.5

In [5]: side2 = ((bx - cx) ** 2 + (by - cy) ** 2) ** 0.5

In [6]: side3 = ((ax - cx) ** 2 + (ay - cy) ** 2) ** 0.5

In [7]: perimeter = side1 + side2 + side3

In [8]: print(perimeter)
12.850591465930588
```

What’s the problem with this?

- We wrote the exact same code 3 times.

Let's code!

```
In [4]: side1 = ((ax - bx) ** 2 + (ay - by) ** 2) ** 0.5

In [5]: side2 = ((bx - cx) ** 2 + (by - cy) ** 2) ** 0.5

In [6]: side3 = ((ax - cx) ** 2 + (ay - cy) ** 2) ** 0.5

In [7]: perimeter = side1 + side2 + side3

In [8]: print(perimeter)
12.850591465930588
```

What’s the problem with this?

- 1. We wrote the exact same code 3 times.

- 2. The computation isn’t easy or readable. It would be easier if it had a ***name***.

- This is what *functions* are for. They take a set of commands (some code), and give it a name, so that you can reuse them easily.

Basic Function Creating

- Steps:

1. Use the ***def*** keyword
2. Function name
3. Brackets
4. Colon
5. Indented function body

```
def say_hi():
    print('Hi!')
```

- This defines (*but does not execute*) the function.

Executing a Function

- Just like always!

- Steps:

1. Function name
2. Brackets

```
In [10]: say_hi()
Hi!
```

If we don’t call the function, it will not run…

```
1  print('Starting...')
2
3  def say_hi():
4      print('Hi!')
5
6  print('Finished!')
7
```

```
C:\Python>python functions1.py
Starting...
Finished!

C:\Python>_
```

If we don’t call the function, it will not run…

```
1  print('Starting...')
2
3  def say_hi():
4      print('Hi!')
5
6  say_hi()
7
8  print('Finished!')
9
```

```
C:\Python>python functions1.py
Starting...
Hi!
Finished!

C:\Python>
```

Arguments & Parameters

- A function can be added to arguments as inputs. These can also be called *parameters*.

- For example, *input* receives the argument *prompt* which is the question to ask the user.
- The function *print* receives inputs to print to the screen.

- These arguments are written inside the parentheses, when the function is called.

- **<u>Dictionary Corner:</u>**
- ***Argument*** – the input given to the function when *calling/executing* the function.
- ***Parameter*** – the input given to the function when *defining* the function.
- These are essentially the same, but used in different contexts.

Single Parameter Example

```
def say_hi(language):
    if language == 'es':
        print('Hola')
    elif language == 'fr':
        print('Bonjour')
    elif language == 'ar':
        print('Salaam')
    elif language == 'en':
        print('Hello')
    else:
        print('Unknown language...')
```

Execution:

```
In [12]: say_hi('ar')
Salaam

In [13]: say_hi('en')
Hello

In [14]: say_hi('zz')
Unknown language...
```

Does this function *return* anything?

- This function, much like the function *print*, prints to the screen but doesn't ***return*** anything.

- In order to return a value, we should use the built-in keyword ***return***.

```
def say_hi():
    return 'Hi!'
```

- The return statement ends the function execution and "sends back" the result of the function.

Using *return*

- Again, once the code hits ***return***, the function execution ends, and the value is given back to whatever called the function.

```
def say_hi(language):
    if language == 'es':
        return 'Hola'
    elif language == 'fr':
        return 'Bonjour'
    elif language == 'ar':
        return 'Salaam'
    elif language == 'en':
        return 'Hello'

greeting = say_hi('ar') + ', Hugh!'
print(greeting)
```

```
C:\Python>python functions2.py
Salaam, Hugh!
```

Functions – Step-by-Step

Let's look at this code.
What will happen?

- The first line to be executed will be:
  a = 25
- Then, is_even(a) will be called.
- The code will jump to the function.
- A new variable named *num* will be created, with the value 25.
- The value n % 2 == 0 will be calculated and returned (False)
- b will now be *False*
- b will be printed. The program will print *False*.

```
def is_even(num):
    return num % 2 == 0

a = 25
b = is_even(a)

print(b)
```

Multiple Parameters

- In order to define multiple parameters, we put them in the brackets of the function definition, divided by a comma:

```
def add(a, b):
    return a + b
```

- Execution:

```
In [16]: add(15, 9)
Out[16]: 24
```

Multiple Return Values

- You can also ***return*** multiple values.
- Let's take, for example, the built in function ***divmod*** – that calculates the integer division value of two numbers, and the remainder:

```
In [17]: divmod(8, 3)
Out[17]: (2, 2)
```

- This function gives a tuple, with two values!

Unpacking Multiple Return Values

```
In [17]: divmod(8, 3)
Out[17]: (2, 2)

In [18]: div, mod = divmod(8, 3)

In [19]: div
Out[19]: 2

In [20]: mod
Out[20]: 2
```

Let's Create Our Own *divmod* Function

```
def my_divmod(x, y):
    div = int(x / y)
    mod = x % y
    return div, mod
```

So, in order for us to return multiple values, we write them both after the ***return*** keyword, divided by a comma. This packs them into a tuple!

```
In [22]: my_divmod(11, 4)
Out[22]: (2, 3)
```

None

- There is a special object in Python named ***None***.

```
In [24]: a = None

In [25]: type(a)
Out[25]: NoneType
```

- It is the only object of type *NoneType*.

- It is customary to use **None** whenever no value should be returned from a function.

No Function Return Value

- Remember when we said that the function ***print*** has no output?
- In fact, its output is **None**:

```
In [26]: output = print('hello')
hello

In [27]: type(output)
Out[27]: NoneType
```

Using None As A Return Value

```
def sqrt(num):
    if num < 0:
        return None
    return num ** 0.5
```

```
In [29]: sqrt(16)
Out[29]: 4.0

In [30]: sqrt(-1)

In [31]:
```

Using No Return Value

- Not writing anything after the ***return*** keyword is the same as writing **None** – it also returns None.

```
def sqrt(num):
    if num < 0:
        return
    return num ** 0.5
```

```
In [31]: type(sqrt(-2))
Out[31]: NoneType
```

Using No *return* Keyword

- In fact, not using ***return*** at all, also returns **None**!

```
def sqrt(num):
    if num >= 0:
        return num ** 0.5
```

```
In [35]: sqrt(9)
Out[35]: 3.0

In [36]: sqrt(-4)

In [37]: type(sqrt(-4))
Out[37]: NoneType
```

To function or not to function...

- Organize your code into logical chunks – every step of the solution should be a function, with a name!
- Don’t repeat yourself - make it work once and then reuse it.
- If something gets too long or complex, then break it up into different chunks and put those chunks in functions.

Variables with The Same Name

- What if I create a variable, and change it inside the function?

- No worries! In every function, all variables are saved in a new area of memory, called a ***scope***. This scope can hold different variable values, without changing the variable values of the outer scope (whoever called the function).

Scope Example

```
def change_n(n):
    n = 3


n = 5
change_n(n)
print(n)
```

```
C:\Python>python functions6.py
5
```

Looking For Variables in Outer Scope

- If a variable is called in a function, but there is no such variable in the local scope, Python looks for the variable in the outer scope – called the ***global*** scope.

```
n = 5

def foo():
    print(n)

foo()
```

```
C:\Python>python functions6.py
5
```

Constants

- Because of this attribute of functions, we have ***constants***.

- Constants are similar to variables, but instead of varying (changing) throughout the script, they are supposed to stay the same.

- Constants are *always* written at the top, or beginning, of the script.
- Constants are written in uppercase letters.

- From now, everything you write in your script that is an unchanging value, should be saved as a constant at the beginning of your script.

- This will help you manage your code in the future – condensing all values in one known place in your script.

Constants - Example

```
USER_NAME_PROMPT = 'What is your name?: '
GREETING = 'Hello {}!'

def say_hello(name):
    print(GREETING.format(name))

user_name = input(USER_NAME_PROMPT)
say_hello(user_name)
```

Comment Lines

- An easy way to explain your code to a reader, is using comment lines.
- Comment lines begin with a hashtag (#) and should be above the line being explained.

```
def calculate_distance(ax, ay, bx, by):
    # Uses Pythagoras Theorem
    return ((ax - bx) ** 2 + (ay - by) ** 2) ** 0.5
```

Docstring

- A ***docstring*** is a documentation of a function – explaining exactly what the function does.

- Every function should have a docstring.

- The docstring is defined by using a multi-line triple quotes string, beneath the function definition line, before the function body.

- Every docstring has 3 parts:

1. Explanation of what the function does.
2. Explanation of every parameter the function receives, and their expected type.
3. Explanation of what the function returns, and the return type.

Docstring - Example

```
def calculate_distance(ax, ay, bx, by):
    """
    This function calculates the distance between two points
    on a graph, using the Pythagoras Theorem.

    :param ax: First point, x axis value
    :type ax: int
    :param ay: First point, y axis value
    :type ay: int
    :param bx: Second point, x axis value
    :type bx: int
    :param by: Second point, y axis value
    :type by: int

    :return: The distance between the two points
    :rtype: float
    """
    return ((ax - bx) ** 2 + (ay - by) ** 2) ** 0.5
```

Docstring in Help / ?

- This docstring now appears whenever help / ? are called:

```
In [39]: help(calculate_distance)
Help on function calculate_distance in module __main__:

calculate_distance(ax, ay, bx, by)
    This function calculates the distance between two points
    on a graph, using the Pythagoras Theorem.

    :param ax: First point, x axis value
    :type ax: int
    :param ay: First point, y axis value
    :type ay: int
    :param bx: Second point, x axis value
    :type bx: int
    :param by: Second point, y axis value
    :type by: int

    :return: The distance between the two points
    :rtype: float
```

Why We Need Docstrings

- Let’s remember how we learned to use the ***input*** function.

- What did we need to know?

1. What the function does
2. What the function receives
3. What the function returns

- Did we need to know exactly how it does this? No!

- Our ***docstrings*** let others use our functions, without having to understand everything about them.

- From now on, all functions should have docstrings.

Can A Function Call Another Function?

- Of course!
- This can help us divide our code into logical chunks!

So why do we need functions?

- Exit the Presentation and open the file entitled triangle_perimeter_functions.py, listed in the TrainerPS under this Presentation.
- Look at the file – isn't it beautiful?

triangle_perimeter_functions.py

Recursion

- Recursion is a way of solving a problem.
- Recursion happens when, during the execution of a function, the function calls itself.

Recursion Creates a Loop!

- What will happen if we execute this function?

```
def what_does_this_do(number):
    return what_does_this_do(number + 1)
```

```
In [41]: what_does_this_do(1)
---------------------------------------------------------------------------
RecursionError                            Traceback (most recent call last)
<ipython-input-41-cc36a19ac7f5> in <module>
----> 1 what_does_this_do(1)

<ipython-input-40-ba08d872490b> in what_does_this_do(number)
      1 def what_does_this_do(number):
----> 2     return what_does_this_do(number + 1)

RecursionError: maximum recursion depth exceeded
```

Solving Factorial Using Recursion

- A ***factorial*** of a number (which is symbolized as the ! sign) is a multiplication of all whole numbers leading up to the number, including itself.

- n! = 1 * 2 * 3 * … * n-1 * n

- 3! = 1 * 2 * 3

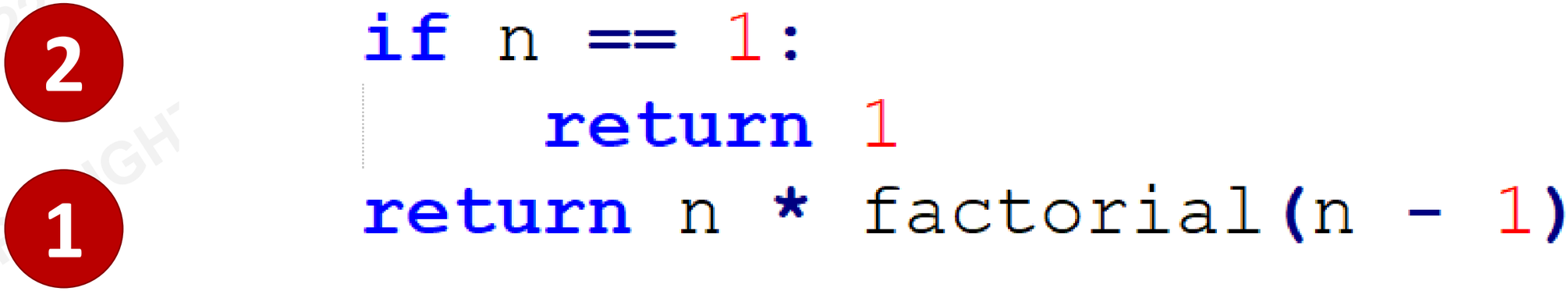
- 6! = 1 * 2 * 3 * 4 * 5 * 6

- We will use ***recursion*** to solve this!

Solving Factorial Using Recursion (cont.)

- Every recursion needs:

1. **A loop in function execution** (a function calling itself)
2. **A stopping rule** (when to stop the loop)



Default Arguments

- Sometimes we will want a function to receive a default value.

- This is very useful when one parameter is almost always the same value, but you still want to give the function more possibilities.

- This is done by using the “=“ sign after the parameter name.

```
def say_hi(language='en'):
```

- This means that when language is not defined (no input is given to the function), language will be ‘en’. But if language is defined, it will change to the chosen input.

Default Arguments (cont.)

- There is no space surrounding the = operator in default arguments.

- The chosen default argument should always be the most used option!
- This is so that we will actually enjoy not having to input the argument.

- Default arguments are always **at the end** of the argument list:

- **GOOD:** `def foo(a=1, b=2, c=3)`
- `def foo(a, b, c=3)`
- `def foo(a, b=10, c=100)`
- **BAD:** `def foo(a=1, b, c)`
- `def foo(a=5, b, c=10)`

Default Arguments - Example

- Let’s take this function:

```
def print_name(name='Hugh', repeat=3)
    for i in range(repeat):
        print(name)
```

- What will this print when executing:
- print_name()
- print_name(“Mohammed”)
- print_name(“Saoud”, 5)
- print_name(repeat=5)
- print_name(repeat=10, name=“Abdullah”)
- print_name(5)

Main Function

- All re-usable code should be in functions.
- That leaves only the code that ***activates*** those functions – runs them one after the other.
- These lines of code should be in their own function, named ***main***. This function should be at the end of the code.
- This means that all of our code is now either constants, or in functions.
- The main should be so simple, and so readable, even my mom would understand what it does!
- Good news: This function is the only one that doesn't need a docstring! This is because it shouldn't do anything new – it should only activate other functions.

Last thing…

- At the *end* of our code, we will write this:

```
if __name__ == "__main__":
    main()
```

- This executes your main function.

- Its OK that you don't understand why we need this condition. Just do it. We'll understand in the future!

Example

- Exit the Presentation and open the file entitles secret_treasure.py, located below the presentation in the TrainerPS.

- It has a very easy ***main()*** function, that is easy to read and understand.
- The rest of the code is also inside functions.
- It ends with our ***if __name__ == '__main__':*** statement.

secret_treasure.py

Summary

- Creating Basic Functions
- Executing Functions
- Defining Parameters and Giving Arguments (single/multiple)
- Returning Values (single/multiple)
- Scopes
- Constants
- Docstrings
- Recursion
- Default Arguments
- Main Function and if __name__ == “__main__”