DAY-3

Functions

You've already seen and used functions such as print and abs. But Python has many more functions, and defining your own functions is a big part of python programming.

In this lesson you will learn more about using and defining functions.

You saw the abs function in the previous tutorial, but what if you've forgotten what it does?

The help() function is possibly the most important Python function you can learn. If you can remember how to use help(), you hold the key to understanding most other function.

Here is an example, run this code on your editor:

```
help(round)
```

Output:
```
Help on built-in function round in module builtins:

round(...)
    round(number[, ndigits]) -> number

    Round a number to a given precision in decimal digits (default 0 digits).
    This returns an int when called with one argument, otherwise the
    same type as the number. ndigits may be negative.
```

help() displays two things:

1. the header of that function round(number[, ndigits]). In this case, this tells us that round() takes an argument we can describe as number. Additionally, we can optionally give a separate argument which could be described as ndigits.
2. A brief English description of what the function does.

Common pitfall: when you're looking up a function, remember to pass in the name of the function itself, and not the result of calling that function.
What happens if we invoke help on a *call* to the function abs()? Unhide the output of the cell below to see.

```
help(round(-2.01))
```

Run this code on you editor & you need not understand all the line of output.

Python evaluates an expression like this from the inside out. First it calculates the value of round(-2.01), then it provides help on the output of that expression.

(And it turns out to have a lot to say about integers! After we talk later about objects, methods, and attributes in Python, the voluminous help output above will make more sense.)

3.1. Round a number.py

```
def round_to_two_places(num):
    return round(num, 2)
print(round_to_two_places(3.14159))
```

Output: 3.14

round is a very simple function with a short docstring. help shines even more when dealing with more complex, configurable functions like print. Don't worry if the following output looks inscrutable... for now, just see if you can pick anything new out from this help.

3.2. help.py

```
help(print)
```

Output:

```
Help on built-in function print in module builtins:

print(...)
    print(value, ..., sep=' ', end='\n', file=sys.stdout, flush=False)

    Prints the values to a stream, or to sys.stdout by default.
    Optional keyword arguments:
    file:  a file-like object (stream); defaults to the current sys.stdout.
    sep:   string inserted between values, default a space.
    end:   string appended after the last value, default a newline.
    flush: whether to forcibly flush the stream.
```

If you were looking for it, you might learn that print can take an argument called sep, and that this describes what we put between all the other arguments when we print them.

Defining functions

Builtin functions are great, but we can only get so far with them before we need to start defining our own functions. Below is a simple example.

3.3.minimum difference1.py

```
def least_difference(a, b, c):
    diff1 = abs(a - b)
    diff2 = abs(b - c)
```

```
    diff3 = abs(a - c)
    return min(diff1, diff2, diff3)
print(
    least_difference(1, 10, 100),
    least_difference(1, 10, 10),
    least_difference(5, 6, 7))
```

Output:

```
9 0 1
```

This creates a function called least_difference, which takes three arguments, a, b, and c. Functions start with a header introduced by the def keyword. The indented block of code following the : is run when the function is called.
return is another keyword uniquely associated with functions. When Python encounters a return statement, it exits the function immediately, and passes the value on the right hand side to the calling context.

Or maybe the help() function can tell us something about it.

```
help(least_difference)
```

Output:

```
Help on function least_difference in module __main__:

least_difference(a, b, c)
```

Python isn't smart enough to read my code and turn it into a nice English description. However, when I write a function, I can provide a description in what's called the **docstring**.

Docstrings

3.4.minimum difference2.py

```
def least_difference(a, b, c):
    """Return the smallest difference between any two numbers
    among a, b and c.

    >>> least_difference(1, 5, -5)
    4
    """
    diff1 = abs(a - b)
    diff2 = abs(b - c)
    diff3 = abs(a - c)
    return min(diff1, diff2, diff3)
```

The docstring is a triple-quoted string (which may span multiple lines) that comes immediately after the header of a function. When we call help() on a function, it shows the docstring.

```
help(least_difference)
```

```
Help on function least_difference in module __main__:

least_difference(a, b, c)
    Return the smallest difference between any two numbers
    among a, b and c.

    >>> least_difference(1, 5, -5)
    4
```

Aside: example calls The last two lines of the docstring are an example function call and result. (The >>> is a reference to the command prompt used in Python interactive shells.) Python doesn't run the example call - it's just there for the benefit of the reader. The convention of including 1 or more example calls in a function's docstring is far from universally observed, but it can be very effective at helping someone understand your function.

Good programmers use docstrings unless they expect to throw away the code soon after it's used (which is rare). So, you should start writing docstrings too.

```
def least_difference(a, b, c):
    """Return the smallest difference between any two numbers
    among a, b and c.
    """
    diff1 = abs(a - b)
    diff2 = abs(b - c)
    diff3 = abs(a - c)
    min(diff1, diff2, diff3)

print(
    least_difference(1, 10, 100),
    least_difference(1, 10, 10),
    least_difference(5, 6, 7),
)
```

Output: None None None

Python allows us to define such functions. The result of calling them is the special value None. (This is similar to the concept of "null" in other languages.)
Without a return statement, least_difference is completely pointless, but a function with side effects may do something useful without returning anything. We've already seen two examples of this: print() and help() don't return anything. We only call them for their side effects (putting some text on the screen). Other examples of useful side effects include writing to a file, or modifying an input.

```
mystery = print()
print(mystery)
```

Output: None

Default arguments

When we called help(print), we saw that the print function has several optional arguments. For example, we can specify a value for sep to put some special string in between our printed arguments:

```
print(1, 2, 3, sep=' < ')
```

Output: 1 < 2 < 3

But if we don't specify a value, sep is treated as having a default value of ' ' (a single space).

```
print(1, 2, 3)
```

Output: 1 2 3

Adding optional arguments with default values to the functions we define turns out to be pretty easy:

3.5. Hello Function.py

```
def greet(who="Tithi"):
    print("Hello,", who)

greet()
greet(who="Paul")
# (In this case, we don't need to specify the name of the argument, because it's
unambiguous.)
greet("Python!")
```

Output: Hello, Tithi
Hello, Paul
Hello, Python!

Functions Applied to Functions

Here's something that's powerful, though it can feel very abstract at first. You can supply functions as arguments to other functions. Some example may make this clearer

3.6.Function in Function.py

```
def mult_by_five(x):
    return 5 * x

def call(fn, arg):
```

```
    """Call fn on arg"""
    return fn(arg)

def squared_call(fn, arg):
    """Call fn on the result of calling fn on arg"""
    return fn(fn(arg))

print(
    call(mult_by_five, 1),
    squared_call(mult_by_five, 1),
    sep='\n', # '\n' is the newline character - it starts a new line
)
```

Output:
5
25

Functions that operate on other functions are called "Higher order functions." You probably won't write your own for a little while. But there are higher order functions built into Python that you might find useful to call.
Here's an interesting example using the max function.
By default, max returns the largest of its arguments. But if we pass in a function using the optional key argument, it returns the argument x that maximizes key(x) (aka the 'argmax').

3.7. Mod of Number.py

```
def mod_5(x):
    """Return the remainder of x after dividing by 5"""
    return x % 5

print(
    'Which number is biggest?',
    max(100, 51, 14),
    'Which number is the biggest modulo 5?',
    max(100, 51, 14, key=mod_5),
    sep='\n',
)
```

Output: Which number is biggest?
100
Which number is the biggest modulo 5?
14

the candy-sharing friends Alice, Bob and Carol tried to split candies evenly. For the sake of their friendship, any candies left over would be smashed. For example, if they collectively bring home 91 candies, they'll take 30 each and smash 1.
Below is a simple function that will calculate the number of candies to smash for *any* number of total candies.

3.8. Number of smash candies.py

```
def to_smash(total_candies, n_friends=3):
   return total_candies % n_friends

print(to_smash(92))
```

Output: 2

```
# # Which of the two variables above has the smallest absolute value?

x = -10
y = 5
a = abs(x)
b = abs(y)
smallest_abs = min(a, b)
print(smallest_abs)
```

Output: 5

Simply write the function's name followed by (), placing any required arguments within the brackets. For example,

3.9. Call functions.py

```
def my_function():
   print("Hello From My Function!")

def my_function_with_args(username, greeting):
   print("Hello, %s , From My Function!, I wish you %s"%(username, greeting))

def sum_two_numbers(a, b):
   return a + b
my_function()
my_function_with_args("John Doe", "a great year!")

x = sum_two_numbers(1,2)
print(x)
```

Output:
Hello From My Function!
Hello, John Doe , From My Function!, I wish you a great year!
3

In this exercise you'll use an existing function, and while adding your own to create a fully functional program.

1. Add a function named list_benefits() that returns the following list of strings: "More organized code", "More readable code", "Easier code reuse", "Allowing programmers to share and connect code together"
2. Add a function named build_sentence(info) which receives a single argument containing a string and returns a sentence starting with the given string and ending with the string " is a benefit of functions!"
3. Run and see all the functions work together!

3.10.

```
# Modify this function to return a list of strings as defined above
def list_benefits():
    pass

# Modify this function to concatenate to each benefit - " is a benefit of functions!"
def build_sentence(benefit):
    pass

def name_the_benefits_of_functions():
    list_of_benefits = list_benefits()
    for benefit in list_of_benefits:
        print(build_sentence(benefit))

name_the_benefits_of_functions()
```

Output:
More organized code is a benefit of functions!
More readable code is a benefit of functions!
Easier code reuse is a benefit of functions!
Allowing programmers to share and connect code together is a benefit of functions!

**Solution In Exercise.py file

----------THANK YOU----------