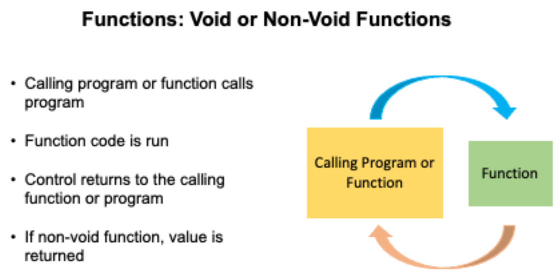
CptS 111, Spring 2023
Lect. #11, Feb. 22, 2023
Class Notes

Today's Agenda:

1. PA #4
2. Look at the flow of functions
3. Reiterate the upside-down nature of coding functions
4. Non-void functions
5. Reasons for using functions
6. Docstrings (and the `help()` function again)

Ch. 5 (cont.)

2. Function Flow


Functions: Void or Non-Void Functions
• Calling program or function calls program
• Function code is run
• Control returns to the calling function or program
• If non-void function, value is returned
Calling Program or Function
Function

3. Program Order

Recall that Python is an interpreted language. Each line of code is interpreted in sequential order starting from the top of a program. This means that all objects must be defined before they can be used. Because of this, it may seem as though we're writing programs upside down when we define functions first. Let's look at an example to see what I mean.

In [1]:
```
# Program that calls function to calculate the cube of a value.

def calc_cube(num):
    print(f'{num} cubed is {num ** 3}.')
    return

number = int(input('Enter the integer to be cubed: '))
calc_cube(number)
```

```
Enter the integer to be cubed: 42
42 cubed is 74088.
```

This is what I mean by upside down. We might think that we should define the function after we prompt for the number we want to cube and then call the function because that's the order we see in the output, i.e., first we prompt for the number, then we call the function, and then the function prints the value of the number that's been cubed. However, let's see what happens if we try to write a program right-side up.

In [2]:
```python
# Program with incorrect order of statements

number = int(input('Enter the integer to be squared: '))
calc_square(number)

def calc_square(num):
    print(f'{num} squared is {num ** 2}.')
    return
```

```
Enter the integer to be squared: 42

---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Input In [2], in <cell line: 4>()
      1 # Program with incorrect order of statements
      3 number = int(input('Enter the integer to be squared: '))
----> 4 calc_square(number)
      6 def calc_square(num):
      7     print(f'{num} squared is {num ** 2}.')

NameError: name 'calc_square' is not defined
```

As we see, Python can't find `calc_square` because it hasn't been defined yet. The Python interpreter interprets the definition of a function first, but it doesn't actually use the function until it's called. In Python, we typically write all functions for a program in one area at or near the top of our code.

4. Non-void Functions

Recall there are two types of functions:

1. void functions
2. non-void functions

   Non-void functions return a value; void functions don't.

Recall the template for a void function:

```
def <function_name>(<parameter_list>): # function header
    <function_body>
    return                             # return is empty, i.e., void
```

where the use of `return` isn't necessary, but it is helpful in CptS 111 to let us know we have completed a function definition.

The template for a non-void function is:

```
def <function_name>(<parameter_list>):
    <function_body>
    return <value>   # ALWAYS RETURN A VALUE FROM A NON-VOID FUNCTION!!!
```

The difference between a void function and a non-void function is that we ***always*** return a value or values from a non-void function.

Note the following:

- **Non-void function calls are assigned to lvalues, i.e., the value returned by the function is assigned to the lvalue.**
- ***If more than one value is returned by a function, we need to separate them by commas and assign the same number of lvalues as the number of values returned.***
- Non-void functions can be called in `print()` statements without an lvalue assignment.
- Non-void functions can be called in equations without an lvalue assignment.
- Non-void and void functions can be called by other functions.
- Non-void functions can be nested, e.g., int(input()).

Also note that `return` isn't a function; it's a special command that tells Python to return the value following it so it can be assigned to the lvalue associated with the function call or else used in the print statement or math expression. As we'll see below, we can actually use Python expressions following the return statement.

Consider the following examples:

In [3]:
```python
# Lvalue assigned to function call to a non-void function;
# we can use functions and operations in the same line as return, but
# note that it isn't obvious what we're returning here

def calc_bat_avg(hits, at_bats):
    return int(1000 * hits / at_bats)

bat_avg = calc_bat_avg(128, 410)  # Use lvalue with non-void function!
print(bat_avg)
```

```
312
```

In [4]:
```
# Non-void function called in print() function;
# code is more readable if values are calculated before return stmt

def calc_bat_avg(hits, at_bats):
    avg = int(1000 * hits / at_bats)
    return avg

print('Your batting average:', calc_bat_avg(128, 410))
```

```
Your batting average: 312
```

In [5]:
```
# Non-void function called twice in math expression

def calc_bat_avg(hits, at_bats):
    avg = int(1000 * hits / at_bats)
    return avg

avg_bat_avg = int((calc_bat_avg(128, 410) + calc_bat_avg(101, 405)) / 2)
print('Average batting average:', avg_bat_avg)
```

```
Average batting average: 280
```

In [6]:
```
# Function called by another function and returning more than one value
# Note that there are the same number of items returned as there are lvalues
# assigned to the calling function

def get_input():
    loan_amt = float(input('Enter loan amount: '))
    int_rate = float(input('Enter interest rate [%]: '))
    yrs_to_pay = int(input('Enter number of years to pay loan: '))
    int_rate_frac = int_rate / 100
    num_months = yrs_to_pay * 12
    return loan_amt, int_rate_frac, num_months

def calc_pymt():
    loan_amt, int_rate, num_months = get_input()
    numerator = loan_amt * int_rate / 12
    denominator = 1 - (1 + int_rate / 12) ** -num_months
    return numerator / denominator

print(f'Your monthly payment will be ${calc_pymt():.2f}.')
```

```
Enter loan amount: 10000
Enter interest rate [%]: 7
Enter number of years to pay loan: 10
Your monthly payment will be $116.11.
```

In [7]:
```
# Nested functions: innermost function called first

integer = int(float(input('Enter a number: ')))
print(integer)
```

```
Enter a number: 1.618
1
```

We can also nest user-defined functions!

In [8]:
```
# Nested user-defined functions
# get_symbol() returns a symbol, and this symbol is passed to
# print_symbol()

def get_symbol():
    symbol = input('Enter a symbol: ')
    return symbol

def print_symbol(symbol):
    print(50 * symbol)
    return

print_symbol(get_symbol()) # void function calls non-void function
```

```
Enter a symbol: @
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
```

5. Why use functions?

The book mentions a number of reasons why functions are useful, including:

1. It's a way of organizing programs so that they're easier to write and also to read at a later time, i.e., they improve the readability of a program so maintaining it is easier.
2. It allows a modular approach to programming, e.g., if you write a function to perform part of a program, you can change the function to improve it without having to change the rest of the program.

3. If a function is useful, we can reuse it in a program or in lots of different programs and avoid redundancy.

6. Docstrings (and the `help()` Function Again)

***Docstrings** follow the function definition header, i.e., they're the first statement in the body of a function,* and are used to briefly explain the purpose of a function. **Unlike comments (which use #'s), docstrings are only used with functions.**

To create a docstring, we set it off by a pair of three triple quotes, either single or double. The book suggests that the docstring be placed on a single line, but you may have your own preference. My preference has evolved to placing the quotes above and below the docstring.

For example,

In [9]:
```
# Using a docstring

def calc_bat_avg(hits, at_bats):  # function header
    '''
    Calculates a hitter's batting average using number of hits and at bats.
    '''
    avg = int(1000 * hits / at_bats)
    return avg

print('Your batting average:', calc_bat_avg(128, 410))
```

```
Your batting average: 312
```

Recall how we used the `help()` function previously with built-in functions, e.g.,

In [10]:
```
help(print)
```

```
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.
```

We can also use the `help()` function with functions we've defined, e.g.,

In [11]:
```
help(calc_bat_avg)
```

```
Help on function calc_bat_avg in module __main__:

calc_bat_avg(hits, at_bats)
    Calculates a hitter's batting average using number of hits and at bats.
```

Pretty cool! As we can see from the above, the `help()` function prints the function name with its parameters and then prints whatever we've written in the docstring.