CptS 111, Spring 2023
Lect. #10, Feb. 13, 2023
Class Notes

Today's Agenda:

1. Introduction
2. Functions and function calls
   - built-in vs user-defined functions
   - non-void functions vs void functions
3. Parameters vs arguments
4. Void functions
5. Keyword arguments and default parameters

Ch. 5

1. Introduction

We write programs so we can use them with different inputs (e.g., loan amounts and interest rates) to get different outputs. The ability to vary values in a program is what makes them so powerful. We don't want to hardwire values in a program. For example,

```
loan_amt = 30000
int_rate = 6
num_years = 10
```

doesn't allow us to change the amount of the loan, and so on, without changing the entire program. And can you imagine how boring it would be if video games only had one way of playing them?

Until now, we've written programs (in labs and PAs) that use only sequential statements. For example:

In [1]:
```python
# Sequence of commands

hours = int(input('Enter the number of elapsed hours: '))
minutes = int(input('Enter the number of elapsed minutes: '))
seconds = int(input('Enter the number of elapsed seconds: '))
total_seconds = (hours * 3600) + (minutes * 60) + seconds
total_hours = total_seconds / 3600
print()
print('The total number of elapsed seconds: ', total_seconds)
print(f'The total number of elapsed hours: {total_hours:.2f}')
```

```
Enter the number of elapsed hours: 42
Enter the number of elapsed minutes: 1618
Enter the number of elapsed seconds: 3141

The total number of elapsed seconds:  251421
The total number of elapsed hours: 69.84
```

In Ch. 5, we'll discuss functions which allow us to code using a modular approach, not just by writing sequences of commands. We'll discuss the reasons for using functions in a later lecture.

2. Functions and Function Calls

Until now, you've only used Python's ***built-in*** functions, functions like `print()` and `input()` . However, we can write our own functions in Python which we call ***user-defined*** functions.

There are two types of built-in and user-defined functions:

1. void functions
2. non-void functions

Non-void functions return something; **void** functions don't. **Void** means empty so think of void functions as returning empty handed after they've been called.

The built-in function `print()` is an example of a void function, and the built-in function `input()` is an example of a non-void function. **Notice that we don't assign lvalues to void functions, but typically we do assign them to non-void functions**.

In [2]:
```python
# Non-void function input() vs void function print()
# Below, input() and print() are function calls to the built-in
# functions input() and print()

name = input('Enter your name: ') # non-void function call needs an lvalue
print(f'Hi, {name}!')              # lvalue can't be used with void func-
                                   # tion call
```

```
Enter your name: SpongeBob
Hi, SpongeBob!
```

Above, `input()` *and* `print()` *are function calls to two built-in functions defined in Python.*

3. Parameters vs Arguments

When we **define** a function, we specify its **parameters**. When we **call** a function, we use **arguments**, and these arguments are passed to the function and used as the values of the parameters. The parameter is in local space; the argument is in global space (we'll discuss these ideas in a later lecture). *Because a function parameter is in local space, it can share the same name as a variable in global space, i.e., global space is outside the function, and Python won't confuse the two.*

4. Void functions

The template for defining a void function is:

```
def <function_name>(<parameter_list>):  # function definition
    <function_body>
    return
```

where `def` is a reserved word (short for define). You don't actually have to include the `return` command in the definition of a void function, but **for CptS 111 you'll use `return` to indicate you've finished defining your function; it will make you think about whether or not something should be returned.** To call the function, we use a function call:

```
<function_name>(<argument_list>)  # function call
```

Again, note that we don't use an lvalue when we call a void function because a void function doesn't return anything.

1) The simplest void function has no parameters or arguments.

In [3]:
```python
# Void function w/ no parameters

def hello():                # Simplest void function with
    print('Hello, World!')  # no parameters in the defn
    return

hello()                     # Function call to void function w/ no arguments
```

```
Hello, World!
```

2) The next simplest void function requires parameters and arguments.

In [4]:
```python
# Void function w/ one parameter (a string)

def greet(name):
    print(f'Hello, {name}!')
    return

greet('EVE')    # We can use a string literal as the function call
print()         # argument

# Call function again

bot = 'WALL-E'
greet(bot)      # Or we can use a variable
```

```
Hello, EVE!

Hello, WALL-E!
```

In [5]:
```python
# Parameter names and arguments can be different

def greet(nom):
    print(f'Hello, {nom}!')
    return

name = 'Freddy Mercury'
greet(name)
```

```
Hello, Freddy Mercury!
```

In [6]:
```
# Or parameter names and arguments can be the same.  Why?

def greet(name):          # name is in local space
    print(f'Hello, {name}!')
    return

name = 'Freddy Mercury'
greet(name)               # name is in global space
```

```
Hello, Freddy Mercury!
```

5. Keyword Arguments (kwargs) and Default (Parameter) Values

When there are lots of parameters in a function definition (>3 or 4), it's common to use keywords in both the function definitions and function calls.

Let's first consider a built-in function with kwargs and default values, the function `print()` .

In [7]:
```
# Use built-in function print without kwargs

print('Go,', 'Cougs!')
```

```
Go, Cougs!
```

One of the kwargs in the `print()` function is `sep` . It's default value is a single space which is why a single space separates `Go,` and `Cougs!` in the previous example. However, we can change the default parameter value to whatever we like.

In [8]:
```
# Change sep from a default blank space to no space

print('Go,', 'Cougs!', sep='')
```

```
Go,Cougs!
```

In [9]:
```
# Change sep from a default blank space to some symbols

print('Go,', 'Cougs!', sep='**!*!*!**')
```

```
Go,**!*!*!**Cougs!
```

So let's see how we can include kwargs and default parameter values in our function definitions. For now, we'll consider a void function, but kwargs and default parameter values can be used in non-void functions, too.

In [10]:
```
# Void function wo/ kwargs

# First, define function

def checkout(price, tax, giftwrap):
    total = price * (1 + tax / 100) + giftwrap
    print(f'Your total charge is ${total:.2f}.')
    return

# Second, call function (note that it occurs AFTER
# the function definition)

checkout(23.75, 7.9, 5)
```

```
Your total charge is $30.63.
```

Now suppose the tax is usually 7.9% and giftwrapping for most presents is $5, e.g., it might be free if the price is more than 100 dollars or if the package is large. We can then use kwargs with default parameter values.

In [11]:
```
# Void function w/ default values for kwargs

def checkout(price, tax=7.9, giftwrap=5):
    total = price * (1 + tax / 100) + giftwrap
    print(f'Your total charge is ${total:.2f}.')
    return

# We can call the function and get the total without including
# the tax and giftwrap arguments

checkout(23.75)
```

```
Your total charge is $30.63.
```

In [12]:
```
# We can also use numbers, but Python will assume the arguments are
# given in the same order as the parameters unless keywords are used.

checkout(23.75, 6)  # Two values given; Python assumes 6 is the arg
                    # for tax
```

```
Your total charge is $30.18.
```

In [13]:
```
# Let's use the kwarg giftwrap

checkout(23.75, giftwrap=0)
```

```
Your total charge is $25.63.
```

In [14]:
```
# Let's switch the order of the kwargs; if kwargs are specified,
# their order doesn't matter

checkout(23.75, giftwrap=0, tax=6)
```

```
Your total charge is $25.18.
```

In [15]:
```
# Let's not use the keywords, but then we must ensure all the args
# are in the correct order

checkout(23.75, 6, 0)
```

```
Your total charge is $25.18.
```

Let's consider another example.

In [16]:
```
# Define kwarg w/ default value for user-defined function
# cost_of_meal() is a void function

def cost_of_meal(check_amt, tip=20):
    total_cost = check_amt * (1 + tip / 100)
    print(f'Your credit card will be charged ${total_cost:.2f}.')
    return
cost_of_meal(65)           # Using default tip of 20%
cost_of_meal(65, tip=15)  # Changing tip to 15%
```

```
Your credit card will be charged $78.00.
Your credit card will be charged $74.75.
```

In [17]:
```
# zCA 5.2.2: kwarg w/ default value
# Note use of built-in functions print(), int(), and input() together
# with user-defined non-void function number_of_pennies()
# 5 6, 4

def number_of_pennies(dollars, cents=0): # We'll cover non-void
    num_cents = (dollars * 100) + cents  # functions next week
    return num_cents

print(number_of_pennies(int(input()), int(input())))
print(number_of_pennies(int(input())))
```

```
5
6
506
4
400
```

In [18]:
```python
# zCA 5.2.3: kwargs w/ default values; split_check() is a non-void function
# 25 2, 100 2 0.075 0.21

def split_check(check, num_persons, tax_percentage=0.09, tip_percentage=0.15):
    total = check * (1 + tax_percentage + tip_percentage)
    return total / num_persons

bill = float(input())
people = int(input())

# Cost per diner at the default tax and tip percentages

print(f'Cost per diner: ${split_check(bill, people):.2f}')

bill = float(input())
people = int(input())
new_tax_percentage = float(input())
new_tip_percentage = float(input())

# Cost per diner at different tax and tip percentages
print(f'Cost per diner: ${split_check(bill, people, new_tax_percentage, new_tip_percentage):.2f}')
```

```
25
2
Cost per diner: $15.50
100
2
0.075
.21
Cost per diner: $64.25
```