Functions: Decomposition And Code Reuse, Part 3

- Global identifiers, scope and program design
- Declaring variables: where in your function/at what level in your program
- Boolean functions
- Breaking long functions into parts
- Common errors when defining functions
- Program design and defining functions
- Testing functions
- Benefits & drawbacks of defining functions

In Class Exercise, Functions

- Write a function called '`emphasize`' that takes a string as a parameter.
- This function returns a modified version of the string:
  - !!! will be added onto the end (three exclamation marks are added to the end of the existing string).
  - Recall: The concatenation operator is the 'plus' operator '+' and it can connect two strings.

James Tam

Declaring Variables: Stylistic Note

- Creating variables all at once at the start of a function.

```
def start():
    #Variables declared
    principle = -1
    rate = -1
    time = -1
    interest = -1
    amount = -1

    introduction()
    principle,rate,time = getInputs()
    interest, amount =
      calculate(principle,rate,time)
    display(principle,rate,time,
            interest,amount)

start()
```

Not syntactically required but a stylistic approach

Origins: many languages (e.g. C, C++, Java, Pascal) require variables to be declared with a specific type before they can be used:

```
fun ()
{
   //Variables declared
   Scanner in = null;
   int age = 0;

   in = new Scanner(System.in);
   age = in.nextInt()
   System.out.print("Age:");
}
```

James Tam

Global Scope (Again)

- Identifiers (constants or variables) that are declared within the body of a function have a local scope (the function).

```
def fun():
    num = 12
    # End of function fun
```

Scope of num is the function

- Identifiers (constants or variables) that are created outside the body of a function have a global scope (the program).

```
num = 12
def fun1():
   # Instructions

def fun2():
   # Instructions

# End of program
```

Scope of num is the entire program

James Tam

Global Scope: An Example

- **Name of the example program**: `8simple_global_example.py`
  - Learning objective: how global variables are accessible throughout a program.

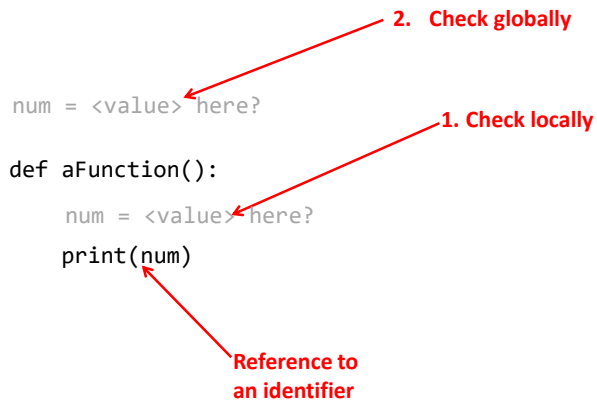
```
num1 = 10

def fun():
   print(num1)

def start():
   fun()
   print(num2)

num2 = 20

start()
```

```
10
20
```

James Tam

Scoping Rules: Globals

- When an identifier is referenced (variable or constant) then:
  1. First look in the local scope for the creation of the identifier: if found here then stop looking and use this identifier
  2. If nothing exists at the local level then look globally



James Tam

Global Variables: Python Specific Characteristic

- **Name of the example program**: `9detailedGlobalsVsLocals.py`
- Learning objective: Relationship between accessing global variables and creating locals.

```
num = 1
def fun():
    num = 2
    print(num)
def start():
    print(num)
    fun()
    print(num)

start()
```

2 Local created and displayed

1 Global

1 Global

James Tam

Python Globals: 'Read' But Not 'Write' Access

- By default global variables can be accessed globally (read access).
- Attempting to change the value of global variable will only create a new local variable by the same name (no write access to the global, a local is created).

```
num = 1

def fun():
    num = 2
    print(num)
```

Global num

Local num

- Prefacing the name of a variable with the keyword '`global`' in a function will indicate changes in the function will refer to the global variable rather than creating a local one.

```
global <variable name>
```

James Tam

Globals: Another Example ('Write' Access Via The "`Global`" Keyword)

- **Name of the example program**: `10modifyingGlobals.py`
- Learning objective: How global variables can be modified inside functions.

```
num = 1

def fun():
    global num
    num = 2
    print(num)

def start():
    print(num)
    fun()
    print(num)

start()
```

References to the name 'num' now affect the global variable, local variable not created inside function 'fun'

2 Global changed

1 Global

2 Global still changed after '`fun()`' is done

James Tam

Global Variables: General Characteristics

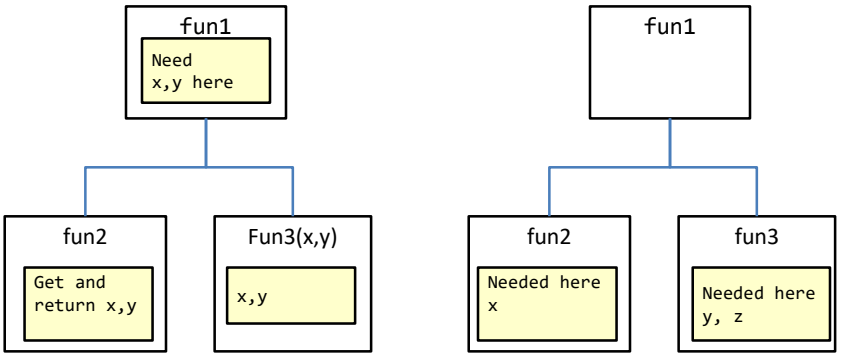
- You can access the contents of global variables anywhere in the program.
  - Python: this can occur even if the '`global`' keyword is not used.
- Benefits (why avoid global variables.)
  - Reason 1st: more efficient use of memory (covered in the last section)
  - **Reason #2 why the use of global variables is regarded as bad style[1]:** In most programming languages you can also modify global variables anywhere as well.
    - This is why the usage of global variables is regarded as bad programming style, they can be accidentally modified anywhere in the program.
      - Changes in one part of the program can introduce unexpected side effects in another part of the program.
  - **Reason #3:** pedagogical (creating variables locally forces you to apply important programming concepts such as parameter passing, function return values and scope).
    - Unless you have a compelling reason **you should NOT be using global variables** but instead you should pass variables as parameters/returning values.
      - Unless you are told otherwise using global variables can affect the style component of your assignment grade.
      - Global constants are acceptable and are commonly used.

1 **Reminder of reason #1**: it's an inefficient use of memory as variables should be allocated only as needed.

James Tam

What Level To Declare Variables

- Declare your variables as local to a function.
- When there are multiple levels of functions (a level is formed when one function calls another) then:
  - A variable should be created at the lowest level possible



James Tam

Recap #1 For The Upcoming Example: Format Specifiers

- **Format**:

```
print ("%<placeholder for type of info to display/code>"
  %<source of the info to display>)
```

- **Types of information** (that can be formatted via the format specifier):

Specifier	Information type	Example

%s	String	`print("%s" %(aStr))`
		`1 Example value: aStr = "axy"`
		`print ("%s" %("ab"))`

%d	Integer	`print("%d" %(aNum))`
		`1 Example value: aNum = 13`
		`print("%d" %(7))`

%f	Floating point	`print("%f" %(12.55))`

James Tam

Recap #2 For The Upcoming Example: Storing String Information

- Typically characters are encoded using ASCII
  - https://www.ascii-code.com/
- Each character is mapped to a numeric value
  - E.g., 'A' = 65, 'B' = 66, 'a' = 97, '2' = 50
- Values are sequential
  - e.g. `'0'=48, '1'=49, '2'=50… '9'=57,`

James Tam

New Term: Boolean Function

- Return a Boolean value (true/false): "Asks a question"
- Typically the Boolean function will 'ask the question' about a parameter(s) and return a True or False value.
- **Name of the example program**: `11booleanFunctionIsNum.py`
- Is it true that a single character string can be passed to the `int()` function (i.e. it's an integer)

`'0'=48, '1'=49, '2'=50… '9'=57,`

```
def start():
    aChar = "0"
    print(isDigit(aChar))
```

```
# Boolean function
def isDigit(aChar):
    digit = False
    if((aChar >= "0") and \
       (aChar <= "9")):
        digit = True
    return(digit)
```

```
# Testing the Boolean function (test driver case)
print("Testing '%s': " %(aChar), isDigit(aChar))
```

James Tam

New Term: Test Driver (Paraphrased From Many Sources)

- Program code that calls a function under different conditions.
- The conditions are typically simulated through the arguments passed to the function being tested.

```
aChar = "9"
print("Testing '%s': " %(aChar), isDigit(aChar))
aChar = "A"
print("Testing '%s': " %(aChar), isDigit(aChar))
```

- The results are often evaluated through the return value of the function being tested.

```
Testing '9':  True
Testing 'A':  False
```

James Tam

Example: How To Decompose A Long Function

- To decompose (break into parts) long functions examine the structure for sections e.g. loops (and their bodies), branches (and their bodies).
- Each of these sections may be a candidate to be moved into it's own separate function body:

Before

```
def fun1():
    while(BE1):
        if(BE2):
          #If body #1
        if(BE3):
          #If body #2
```

After

```
def fun3():
    #If body #2

def fun2():
    #If body #1

def fun1():
    while(BE1):
        if(BE2):
            fun2()
        if(BE3):
            fun3()
```

James Tam

The Starting Function Needs To Be Defined Before It Can Be Called

- **Correct** ☺

```
def fun():
    print("Works")

# Start
fun()
```

Function definition

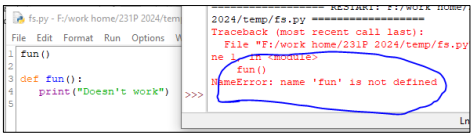
Function call

- **Incorrect** ☹

```
# Start
fun()

def fun():
    print("Doesn't work")
```

Function call

Function definition



Another Common Mistake

- Forgetting the brackets during the function call:

```
def fun():
    print("In fun")

# Start of program
print("Starting the program")
fun
```

James Tam

Another Common Mistake

- Forgetting the brackets during the function call:

```
def fun():
    print("In fun")

# Start of program
print("Program started")
fun()
```

Unlike many other languages the missing set of brackets do not produce a syntax/translation error (likely it will be logic error because the function isn't called)

James Tam

Another Common Problem: Indentation

- Recall: In Python indentation indicates that statements are part of the body of a function.
- (In other programming languages the indentation is not a mandatory part of the language but indenting is considered good style because it makes the program easier to read).
- Forgetting to indent:

```
def start():
print("start")

start()
```

James Tam

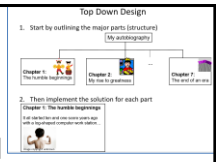
Another Common Problem: Indentation (2)

- Inconsistent indentation:

```
def start():
  print("first")
    # Error: Unless this is the body of branch or loop
    print("second")

start()
```

James Tam

Creating A Large Document



- Recall: When creating a large document you should plan out the parts before doing any actual writing.

Step 1: Outline all the parts (no writing)

Chapter 1
- Introduction
- Section 1.1
- Section 1.2
- Section 1.3
- Conclusion

Chapter 2
- Introduction
- Section 2.1
- Section 2.2
- Section 2.3
- Section 2.4
- Conclusion

Chapter 3
- Introduction
- Section 3.1
- Section 3.2
- Conclusion

Step 2: After all parts outlined, now commence writing one part at a time

Section 1.1
It all started seven
and two score
years ago…

James Tam

Creating A Large Program

- When writing a large program you should plan out the parts before doing any actual writing.

Step 1: Calculate interest (write empty 'skeleton' functions)

```
def getInformation():    def doCalculations():    def displayResults():
```

Step 2: All functions outlined, write function bodies one-at-a-time (test before writing next function)

```
def getInformation():
    principle = int(input())
    interest = int(input())
    time = int(input())
    return(principle,interest,time)
```

```
# Simple test: check inputs
# are properly read as input
# and returned to caller
p,r,t = getInformation()
print(p,r,t)
```

James Tam

Yet Another Problem: Creating 'Empty' Functions

```
def start():
```

```
start()
```

Problem: This statement appears to be a part of the body of the function but it is not indented???!!!

James Tam

Solution When Outlining Your Program By Starting With 'Empty' Functions

```
def fun():
    print()

# Program'sstart
fun()
```

A function must have at least one instruction in the body

Alternative (writing an empty function: 'pass' a python instruction that literally does nothing)

```
def fun():
    pass

# Program's start
fun()
```

James Tam

Testing Functions

- The correctness of a function should be verified. ("Does it do what it is supposed to do?")
- Typically this is done by calling the function, passing in predetermined parameters and checking the result i.e. via a test driver
- Example[1]: `12absoluteTest.py`

```
def absolute(number):
    if (number < 0):
        result = number * -1
    else:
        result = number
    return(result)
# Test cases/drivers
print(absolute(-13))
print(absolute(7))
```

Expected results:
13
7

1: In case you are wondering about the naming use. It's not testing for an absolute value but instead you're testing a program that implements an absolute value feature.

James Tam

How To Write Your Programs

- Reminder:
  - Outline the whole program with empty functions (stubs) that don't perform any tasks yet.
  - Implement the code for each function one at a time.
  - Test each function with a driver.
  - Fix any bugs, test again until you are reasonable sure the function is correct.
  - Only then should you move onto writing the next function.
- One approach:
  - Write/test each function in a file separate from the main program.
  - Only after you are sure of that function's correctness do you add its code into the file containing the main program.

James Tam

Why Employ Problem Decomposition And Modular Design (1)

- Drawback
  - Complexity – understanding and setting up inter-function communication may appear daunting at first.
  - Tracing the program may appear harder as execution appears to "jump" around between functions.
  - These are 'one time' costs: once you learn the basic principles of functions with one language then most languages will be similar.

Why Employ Problem Decomposition And Modular Design (2)

- Benefit
  - Solution is easier to visualize and create (decompose the problem so only one part of a time must be dealt with).
  - Easier to test the program:
    - Test one feature/function at a time
    - (Testing multiple features increases complexity)
  - Easier to maintain (if functions are independent changes in one function can have a minimal impact on other functions, if the code for a function is used multiple times then updates only have to be made once).
  - Less redundancy, smaller program size (especially if the function is used many times throughout the program).
  - Smaller programs size: if the function is called many times rather than repeating the same code, the function need only be defined once and then can be called many times.

James Tam

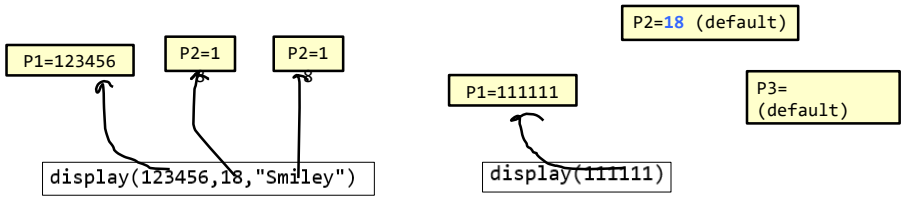
Extra Topics

- These are language specific implementations and will be covered (and tested) only there is time (i.e. we are ahead of schedule in the lectures).

James Tam

Default Arguments

- If some parameters are excluded in the function call then default values ("default arguments") can be specified.



- It can be employed when some parameters are not always known when the function is called.
  - Example: creating a new 'client' or 'patient' but the information is not all available when the person signs up.
- **New term, default argument**: if an argument is omitted then the default value is used for the missing value.

James Tam

Default Arguments: An Example

- **Name of the example program**: `13_default_parameters`
  - Learning objective: how define and call a function with default arguments.

```
def display(studentID,anAge=-1,aName="No name"):
    print(aName,studentID,anAge)

def start():
    display(123456,18,"Smiley")
    display(111111)
```

```
Smiley 123456 18
No name 111111 -1
```

James Tam

Syntax requirements For Default Arguments

- Default arguments cannot be followed by non-default arguments.
- Alternative wording: all optional arguments must be at end of the parameter list.
  - Example below: 1st argument (default) is followed by a non-default argument

```
def wrongDisplay(aName="No name",studentID,anAge=-1,):
    print(aName,studentID,anAge)
```

James Tam

Recall: In General Parameter Order Is Critical

```
def fun1(y,x):
    print(y,x)

def fun2(a,b):
    print(a,b)

#Starting execution point
def start():
    x = 1
    y = 2
    z = "hello"
    fun1(x,y)
    fun2(x,z)

start()
```

Order of the parameters (not the names) determines how parameters match

James Tam

New Term: Key Word Arguments

- The except is when the parameters being passed into a function are assigned values during the call.
  - In this case it's the **name of the parameters** that determine which parameters match up on the function call vs. the function definition.
  - In this case the **names must match**.
  - But order is not relevant.
- **Format (during function call):**

  <function name>(*<arg1>*=*<value>*, *<arg2>*=*<value>*, *<arg3>*=*<value>*…)
- **Example of using keyword arguments**
  - Function call (**default values** mandatory):

```
fun1(aNum=888,aStr="Lucky")
```

- **Example**
  - Function definition:

```
def fun1(aStr,aNum):
    print(aNum,aStr)
```

James Tam

Example Of Keyword Arguments

- **Name of the example program**: `14_keyword_arguments`
  - Learning objective: how to use keyword arguments (using the name of the arguments to specify how arguments are matched up in the function call vs. the function definition).

```
def fun1(aStr,aNum):
    print(aNum,aStr)

def fun2(localNum,localString):
    print(localNum,localString)

def start():
    print("fun1(), using keyword arguments: ", end="")
    fun1(aNum=888,aStr="Lucky")
    aNum=777
    print("fun2(), using normal positional arguments: ", end="")
    fun2(aNum,"Also lucky")
```

```
fun1(), using keyword arguments: 888 Lucky
fun2(), using normal positional arguments: 777 Also lucky
```

James Tam

If The Names Don’t Match: Won’t Work

```
#def fun3(aNum1,aStr1):
#    print(aNum1,aStr1)

#print("fun3(), using keyword arguments but names don't
#match: ")
#fun3(aNum=888,aStr="lucky")
```

Treated as keyword not positional arguments because **default values** are provided

James Tam

After This Section You Should Now Know

- What is global scope
- Consequences of employing global scope
- What are scoping rules when referring to an identifier
- Where variables should be declared in the body of a function
- A guideline for the level at which variables should be declared
- How/when to employ doc string documentation
- What is a Boolean function
- A technique for decomposing a long function into smaller functions
- Common errors when defining functions
- The basics of testing a function
- The benefits & drawbacks of defining functions
- New terms & concepts: Boolean function, test driver, default arguments, keyword parameters.

James Tam

Copyright Notification

- “Unless otherwise indicated, all images in this presentation are used with permission from Microsoft.”

James Tam