Every Boilermaker Engineer Codes: 101
Entry-Level Programming in Python

Lecture 04

Dr. John H. Cole
<jhcole@purdue.edu>

PURDUE
UNIVERSITY®

COLLEGE OF ENGINEERING

Spring 2021

Part I

Functions

Functions

Function a reusable block of code that performs a specific task

- used to divide a large program into several small tasks (divide and conquer)
- their code is executed whenever the function is called
- called from sequential, conditional (`if`) and repetition (`for`, `while`) control structures

You have already used several functions (e.g. `input()`, `print()`, and `range()`).

Parts of a Function Definition

Function Header the first line of a function definition

- starts with the keyword `def`
- assigns a name
- includes comma separated parameters in parentheses
- ends with a colon :
- e.g. `def function_name(parameter1, parameter2):`

Function Body indented block of code after the function header

- executes every time the function is called

Function Syntax

Editor - function_syntax.py

```
1  statement_1
2
3  def func_name(param):
4      statement_2
5
6      statement_3
7
8  func_name(arg_1)
9  func_name(arg_2)
10 func_name(arg_3)
11
12 statement_4
```

- line 3 is the function header
- lines 4 and 6 are the function body
- lines 8, 9 and 10 call the function

`statement_1` always executes first

`statement_2` executes each time `func_name` is called

`statement_3` executes each time `func_name` is called

`statement_4` always executes last

Function Benefits

Functions make development easier by

- increasing code readability
- enabling code reuse
- facilitating collaboration
- easing testing and debugging

Functions Naming

- use descriptive function names (e.g. `calculate_tax`, or `display_report`, but **not** `ct1` or `disrep`)
- first character must be a letter or underscore
- remaining characters must be letters, numbers or underscores
- names are case sensitive (Avg != avg)
- cannot use spaces
- cannot use keywords

False	assert	del	for	in	or	while
None	break	elif	from	is	pass	with
True	class	else	global	lambda	raise	yield
and	continue	except	if	nonlocal	return	
as	def	finally	import	not	try	

Calling a Function

Editor - grail_1.py

```
1 def proclaim():
2     print('  I am Arthur')
3     print('  King of the Britons')
4
5 print('Who goes there?')
6 proclaim()
```

Terminal

```
$ python grail_1.py
Who goes there?
  I am Arthur
  King of the Britons
```

- The function named `proclaim` is defined on lines 1-3.
- The function body does not execute until it is called on line 6.

Calling a Function Before It's Defined

Editor - grail_2.py

```
1 print('Who goes there?')
2 proclaim()
3 def proclaim():
4     print('  I am Arthur')
5     print('  King of the Britons')
```

- The function must be defined before it is called.

Terminal

```
$ python grail_2.py
Who goes there?
Traceback (most recent call last):
  File "grail_2.py", line 2, in <module>
    proclaim()
NameError: name 'proclaim' is not defined
```

Calling a Function in a Function

Editor - grail_3.py

```
1  def main():
2      print('Who goes there?')
3      proclaim()
4      print("You're using coconuts!")
5
6  def proclaim():
7      print('  I am Arthur')
8      print('  King of the Britons')
9
10 main()
```

- you can define as many functions as you need
- functions can call other functions

Using a Main Function

Terminal

```
$ python grail_3.py
Who goes there?
  I am Arthur
  King of the Britons
You're using coconuts!
```

- often an entire program is wrapped in a `main` function
- defines the *mainline logic* of the program
- calls other functions when they are needed
- called to start the program

Arguments and Parameters

Argument data passed into a function in a function call

Parameter a variable in a function that receives an argument

Editor - grail_4.py

```
1 def proclaim(name):
2     print(f'  I am {name}')
3     print('  King of the Britons')
4 proclaim('Arthur')
```

- `'Arthur'` is passed as an argument to the parameter `name`.

Terminal

```
$ python grail_4.py
  I am Arthur
  King of the Britons
```

Multiple Arguments and Parameters by Position

- functions can have multiple parameters
- separate parameters by commas in function definition
- separate arguments by commas in function call
- arguments are passed to parameters *by position*

Editor - grail_5.py

```
1 def proclaim(name, title):
2     print(f'  I am {name}')
3     print(f'  {title} of the Britons')
4
5 proclaim('Arthur', 'King')
```

Multiple Arguments and Parameters by Keyword

- function parameters become keywords in a function call
- arguments specify parameters by keyword (e.g. `name = 'Arthur'`)
- when using keywords, position doesn't matter

Editor - grail_6.py

```
1 def proclaim(name, title):
2     print(f'  I am {name}')
3     print(f'  {title} of the Britons')
4
5 proclaim(name = 'Arthur', title = 'King')
6 proclaim(title = 'King', name = 'Arthur')
```

- lines 5 & 6 are equivalent

Mixing Positional and Keyword Arguments

- positional arguments must appear first
- keyword arguments must appear last

Editor - grail_7.py

```
1  def proclaim(name, title, noun):
2      print(f'  I am {name}')
3      print(f'  {title} of the {noun}')
4
5  proclaim('Arthur', 'King', 'Britons')
6  proclaim('Arthur', 'King', noun='Britons')
7  proclaim('Arthur', title='King', noun='Britons')
8  proclaim('Arthur', noun='Britons', title='King')
9  proclaim(name='Arthur', title='King', noun='Britons')
10 #proclaim(name='Arthur', title='King', 'Britons')
```

Default Parameters

- parameters can use default values when arguments are not provided

Editor - grail_9.py

```
1 def proclaim(name = 'Arthur', title = 'King',
2               noun = 'Britons'):
3     print(f'  I am {name}')
4     print(f'  {title} of the {noun}')
5
6 proclaim()
7 proclaim(title='Knight', noun='Round Table')
8 proclaim('Sir Robin')
9 proclaim('Sir Robin', 'Knight', 'Round Table')
```

Returning Values

Two types of function:

Void executes body statements then terminates (e.g. `print`)

Value-Returning executes body statements then returns a value to the calling statement (e.g. `input`)

Void Functions

- no return value is specified
- defaults to returning None
- called for their side effects

Terminal

```
$ python
>>> a = print('Hi')
Hi
>>> a == None
True
>>> type(a)
<class 'NoneType'>
```

Editor - grail_8.py

```
1 def proclaim(name):
2     print('  I am', name)
3     print('  King of the Britons')
```

Value-Returning Functions

- use the keyword `return` followed by a value
- `return` stops executing the function body
- returned values are available to calling statement

Editor - power.py

```
1 def pow(a, b):
2     return a**b
3     print("won't print")
4
5 c = pow(2, 3)
6 print('2^3 =', c)
7 print('3^2 =', pow(3, 2))
8 print('8^8 =', pow(c, c))
```

Terminal

```
$ python power.py
2^3 = 8
3^2 = 9
8^8 = 16777216
```

- argument and parameter variable names don't need to match

Returning Multiple Values

Editor - roots.py

```
1 def quadratic_root(a, b, c):
2     # calculate discriminant d
3     d = b**2 - 4*a*c
4     # calculate the roots
5     r1 = (-b + d**(1/2))/(2*a)
6     r2 = (-b - d**(1/2))/(2*a)
7     return r1, r2
8 p, q = quadratic_root(4, 20, 24)
9 print(f'roots at {p} and {q}')
```

Terminal

```
$ python roots.py
roots at -2.0 and -3.0
```

- can return multiple values using multiple assignment
- can return a list with anything we need in it

Variable Scope

Scope the part of a program in which a variable may be accessed

Global Variable a variable assigned a value outside all functions

Local Variable a variable assigned a value inside a function

Local Variables are Local

- local variables are only accessible within the function that created them

Editor - scope_1.py

```
1 def func():
2     a = 'spam'
3     print(a)
4 func()
5 print(a)
```

Terminal

```
$ python scope_1.py
spam
Traceback (most recent call last):
  File "scope_1.py", line 5, in <module>
    print(a)
NameError: name 'a' is not defined
```

Local Variables Must Be Defined

- local variables must be defined before they are used

Editor - scope_2.py

```
1 def func():
2     print(a)
3     a = 'spam'
4 func()
```

Terminal

```
$ python scope_2.py
Traceback (most recent call last):
  File "scope_2.py", line 4, in <module>
    func()
  File "scope_2.py", line 2, in func
    print(a)
UnboundLocalError: local variable 'a' referenced
before assignment
```

Parameters Are Local Variables

Editor - scope_3.py

```
1 def fun(a):
2     print('enter fun', a)
3     a += 1
4     print('exit fun', a)
5
6 a = 1
7 print('before fun', a)
8 fun(a)
9 print('after fun', a)
```

- changes made to parameter values do not change the arguments value
- known as pass by assignment
- arguments provide one way communication into a function

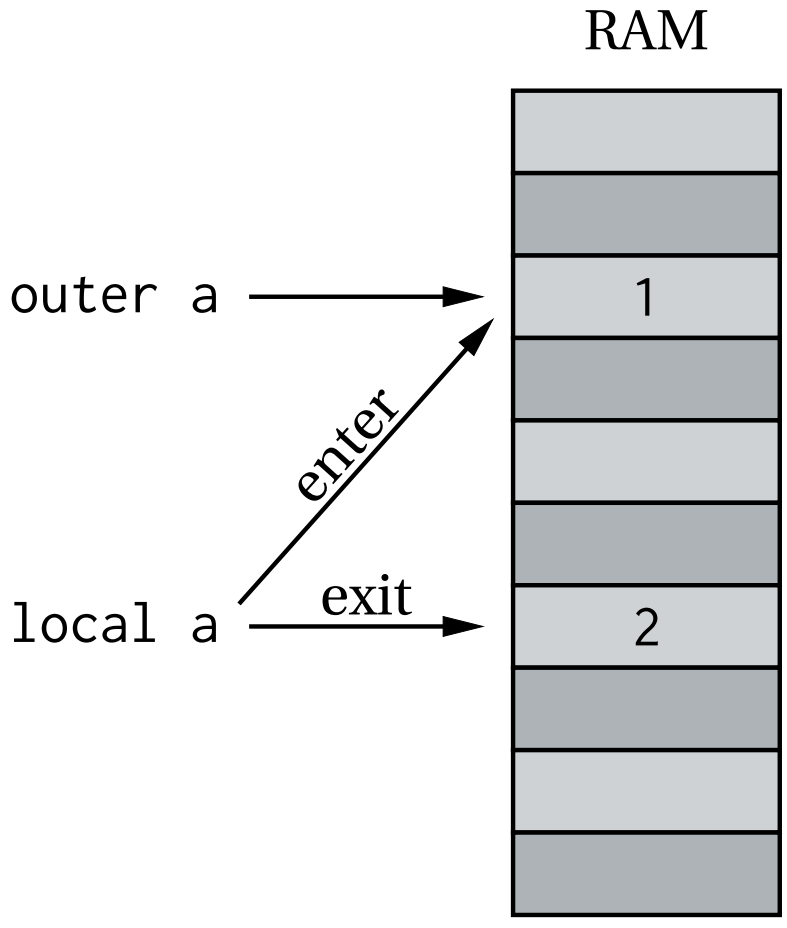
Terminal

```
$ python scope_3.py
before fun 1
enter fun 1
exit fun 2
after fun 1
```

Parameters Are Local Variables

Editor - scope_3.py

```
1 def fun(a):
2     print('enter fun', a)
3     a += 1
4     print('exit fun', a)
5
6 a = 1
7 print('before fun', a)
8 fun(a)
9 print('after fun', a)
```



Functions Only See Their Own Local Variables

Editor - scope_4.py

```
1 def fun1(a):
2     print(a, end='')
3 def fun2(a):
4     print(a, end='')
5 a = 'and spam'
6 fun1('Spam, Spam, ')
7 fun2('Spam, egg ')
8 print(a)
```

- different functions can use variables with the same name
- the functions cannot see each others local variables

Terminal

```
$ python scope_4.py
Spam, Spam, Spam, egg and Spam
```

Global Variables are Global

- global variables are accessible everywhere within the program
- functions can only assign to global variables if they are declared as global in the function

Terminal

```
$ python scope_5.py
Spam bacon Spam and Spam
```

Editor - scope_5.py

```
1  a = 'Spam '
2  def fun1():
3      print(a, end='')
4  def fun2():
5      a = 'bacon '
6      print(a, end='')
7  def fun3():
8      global a
9      a = 'and Spam\n'
10 fun1() # Spam
11 fun2() # bacon
12 fun1() # Spam
13 fun3() #
14 fun1() # and Spam
```

When to Use Global Variables

Avoid Global Variables

- global variables make a program harder to understand
- the variable could be changed anywhere in your program which makes debugging difficult
- functions using global variables are harder to reuse in another program

Global Constants are OK

- global constants are never changed in your program
- keep globals constant by not redeclaring them in functions (i.e. never use the keyword `global`)

Part II

Your Turn

Practice Exercise

Complete the function named `calc_avg` in the program below. The function accepts a list of numbers (`data`) and should calculate the average of the numbers in the list. Then your function should `return` the calculated average. Once its complete, run the code to see the result.

Editor - practice.py

```
1 def calc_avg(values):
2     # write your code here
3
4 data = [3, 18, 7, 1373]
5 print(f'The average is: {calc_avg(data)}')
```