CS303E: Elements of Computers and Programming

Functions

Mike Scott
Department of Computer Science
University of Texas at Austin

Adapted from
Professor Bill Young's Slides

Last updated: June 21, 2023

Functions

- We have used several built in functions already:
  - print(), input(), int(), float(), range()
- List of Python built in functions

		Built-in Functions		
abs()	dict()	help()	min()	setattr()
all()	dir()	hex()	next()	slice()
any()	divmod()	id()	object()	sorted()
ascii()	enumerate()	input()	oct()	staticmethod()
bin()	eval()	int()	open()	str()
bool()	exec()	isinstance()	ord()	sum()
bytearray()	filter()	issubclass()	pow()	super()
bytes()	float()	iter()	print()	tuple()
callable()	format()	len()	property()	type()
chr()	frozenset()	list()	range()	vars()
classmethod()	getattr()	locals()	repr()	zip()
compile()	globals()	map()	reversed()	__import__()
complex()	hasattr()	max()	round()	
delattr()	hash()	memoryview()	set()	

Modules - More Functions

- In addition to the standard built in functions.
standard Python includes many modules
  - Modules are Python scripts (programs) that contain, typically, related functions that we can reuse in many Python programs and scripts
- When you download Python, you download the standard modules.
- Most of these modules are beyond the scope of this course.
- Two that we will use are the math module mathematical operations which don't have defined operators and the random module, with functions to generate *pseudo* random numbers.

Function	Description	Example
fabs(x)	Returns the absolute value of the argument.	fabs(-2) is 2
ceil(x)	Rounds x up to its nearest integer and returns this integer.	ceil(2.1) is 3 ceil(-2.1) is -2
floor(x)	Rounds x down to its nearest integer and returns this integer.	floor(2.1) is 2 floor(-2.1) is -3
exp(x)	Returns the exponential function of x (e^x).	exp(1) is 2.71828
log(x)	Returns the natural logarithm of x.	log(2.71828) is 1.0
log(x, base)	Returns the logarithm of x for the specified base.	log10(10, 10) is 1
sqrt(x)	Returns the square root of x.	sqrt(4.0) is 2
sin(x)	Returns the sine of x. x represents an angle in radians.	sin(3.14159 / 2) is 1 sin(3.14159) is 0
asin(x)	Returns the angle in radians for the inverse of sine.	asin(1.0) is 1.57 asin(0.5) is 0.523599
cos(x)	Returns the cosine of x. x represents an angle in radians.	cos(3.14159 / 2) is 0 cos(3.14159) is -1
acos(x)	Returns the angle in radians for the inverse of cosine.	acos(1.0) is 0 acos(0.5) is 1.0472
tan(x)	Returns the tangent of x. x represents an angle in radians.	tan(3.14159 / 4) is 1 tan(0.0) is 0
fmod(x, y)	Returns the remainder of x/y as double.	fmod(2.4, 1.3) is 1.1
degrees(x)	Converts angle x from radians to degrees	degrees(1.57) is 90
radians(x)	Converts angle x from degrees to radians	radians(90) is 1.57

Importing Modules

- To use non standard functions, ones that are part of a module, we call the function with the name of the module, a period *spoken "dot"*, and the name of the function. math.sqrt(1000)

```
>>> math.sqrt(1000)
Traceback (most recent call last):
  File "<input>", line 1, in <module>
NameError: name 'math' is not defined
```

- must also import the module

```
>>> import math
>>> math.sqrt(1000)
31.622776601683793
```

- In a program or script, imports at the top of the file.

The random Module

- Several useful functions are defined in the random module:
- **`randint(a, b):`** generate a random integer between a and b, inclusively.
- **`randrange(a, b):`** generate a random integer between a and b-1, inclusively.
- **`random():`** generate a float in the range [0 . . . 1).
- How would we simulate flipping a coin with two sides?

Examples of Calls to random Functions

```
>>> import random
>>> random.randint(1, 2)
2
>>> random.randint(1, 2)
2
>>> random.randint(1, 2)
2
>>> random.randint(1, 2)
1
>>> random.randint(1, 2)
1
>>> random.randint(1, 6)
6
>>> random.randint(1, 6)
4
>>> random.randint(1, 6)
3
>>> random.randrange(1, 2)
1
>>> random.randrange(1, 2)
1
>>> random.randrange(1, 2)
1
>>> random.randrange(1, 2)
>>> |
```

```
>>> random.randrange(1, 3)
1
>>> random.randrange(1, 3)
2
>>> random.random()
0.8773265491912745
>>> random.random()
0.6165742684164001
>>> random.random()
0.9273524701896365
>>> random.random()
0.13852627933299988
>>> random.random()
0.664132281949973
>>> for i in range(0, 10):
...     print(random.randint(1, 100))
...
63
51
43
87
60
51
33
26
```


Importing Modules

- Typing the name of the module every time can be tedious
  - A lot of programming languages and IDEs have features to reduce the amount of typing we have to do
- Can import specific or all functions from a module:

```
>>> from random import randint
>>> randint(1, 100)
78
>>> randint(1, 10)
8
>>> random()
Traceback (most recent call last):
  File "<input>", line 1, in <module>
TypeError: 'module' object is not callable
>>> from random import *
>>> random()
0.06999097275883659
```

The * is a wildcard, meaning all.

- **Any downside to always importing all?**

Three Common Data Types

Three data types we will use in many of our early Python programs are:

`int`: signed integers (whole numbers)

- Computations are exact and *of unlimited size*
- Examples: 4, -17, 0

`float`: signed real numbers (numbers with decimal points) Large

- range, but fixed precision
- Computations are approximate, not exact Examples:
- 3.2, -9.0, 3.5e7

`str`: represents text (a string)

- We use it for input and output We'll see
- more uses later Examples: "Hello, World!",
- 'abc'

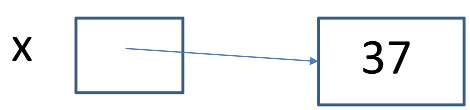
These are all *immutable.* The value cannot be altered.

Immutable

- It may appear some values are mutable
  - they are not
  - rather variables are mutable and can be bound (refer to) different values
- Note, how the id of x (*similar to its address)* has changed

```
>>> x = 37
>>> x
37
>>> id(x)
140711339416352
>>> x = x + 10
>>> x
47
>>> id(x)
140711339416672
```


x = 37



x = x + 10

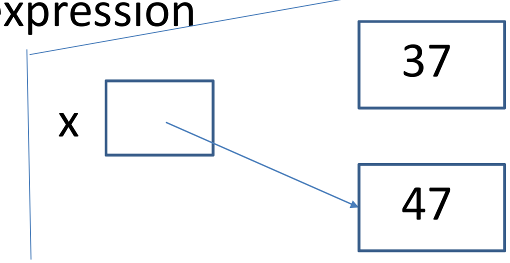
\# substitute in the value x is referring to

x = 37 + 10

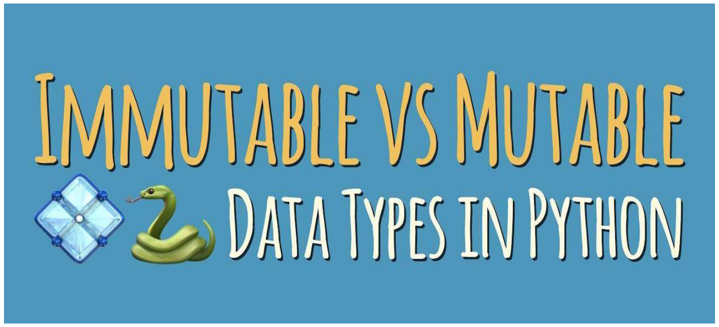
\# evaluate the expression

x = 47

\# so now …



Mutable vs. Immutable



An **immutable** value is one that cannot be changed by the programmer after you create it; e.g., numbers, strings, etc.

A **mutable** values is one that can be changed; e.g., sets, lists, etc.

What Immutable Means

- An **immutable** object is one that cannot be changed by the programmer after you create it; e.g., numbers, strings, etc.
- It also means that *there is typically only one copy of the object in memory.*
- Whenever the system encounters a new reference to 17, say, it creates a pointer (references) to the already stored value 17.
- Every reference to 17 is actually a pointer to the *only* copy of 17 in memory. Ditto for "abc".
- If you do something to the object that yields a new value (e.g., uppercase a string), you're actually creating a new object, not changing the existing one.

Function

We've seen lots of system-defined functions; now it's time to define our own., like main.

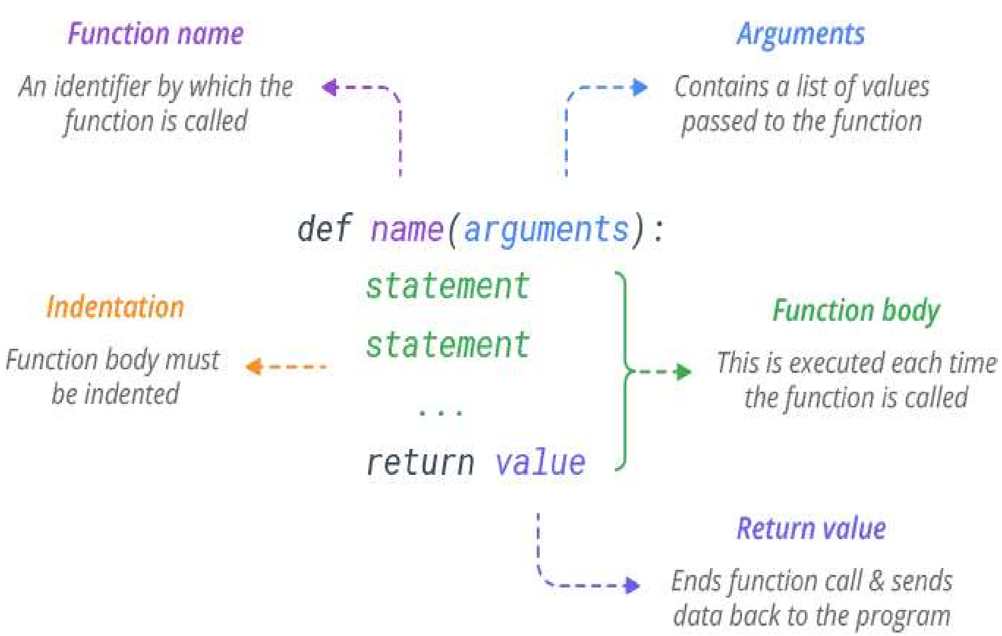
General form:

```
def functionName( list of parameters ):
                                          #
    header statement(s)                   # body
```

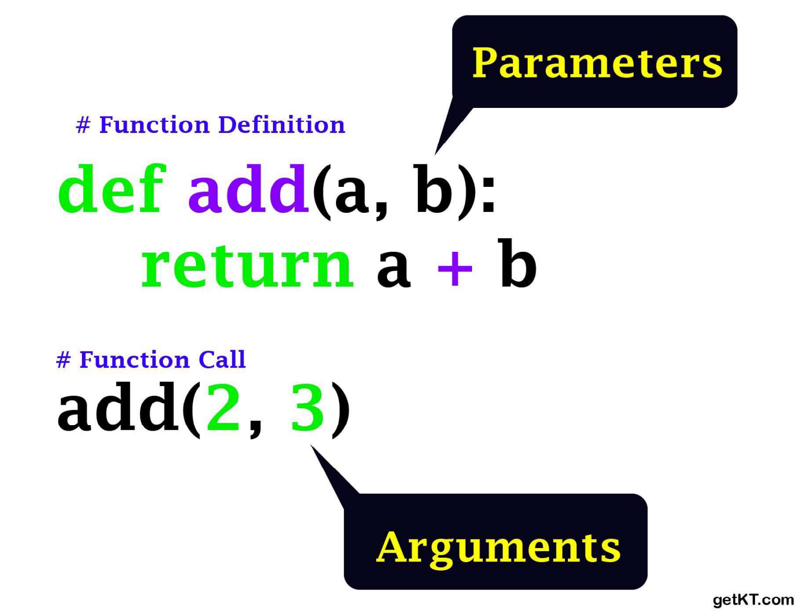
Meaning: a function definition defines a block of code that performs a specific task. It can reference any of the variables in the list of parameters. It may or may not return a value.

The parameters are **formal parameters**; they hold arguments (refer to the same values) passed to the function later when the function is *called.*

Functions



Calling a Function



Function Example

Suppose you want to sum the integers 1 to n.

In file function_examples.py:

```
# Return the sum of values from 1 to n.
# This is an example of a cumulative sum algorithm.
def sum_to_n(n):
    total = 0
    for i in range(1, n + 1):
        total += i
    return total
```

Notice this defines a *function* to perform the task, but *won't perform the task until the function is called from else where.*
We still have to call/invoke the function with specific arguments.

```
def main():
    print(sum_to_n(1))
    print(sum_to_n(1000))
```

```
1
500500

Process finished with exit code
```


Some Observations

```
def sum_to_n(n)          # function header
    ....                 # function body
```

Here n is a *formal parameter*. It is used in the definition as a place holder for an *actual parameter* (e.g., 10 or 1000) in any specific call.

sum_to_n(n) *returns* an int value, meaning that a call to sum_to_n can be used anyplace an int expression can be used.

```
x = sum_to_n(30)
print(x)
print('Even' if sum_to_n(5) % 2 == 0 else 'Odd')
for i in range(1, 30):
    print(i, sum_to_n(i))
```

Note, with functions the argument is the input.
We occasionally ask the user for input in the function.

Functional Abstraction

Once we've defined sum_to_n, we can use it almost as if were a primitive in the language without worry about the details of the definition.

We need to know what it does,
but don't care anymore how it does it!

This is called **information hiding** and / or **functional abstraction**.

And that is **POWERFUL!**

Another Way to Add Integers 1 to N

Suppose later we discover that we could have coded sumToN more efficiently (as discovered by the 8-year old C.F. Gauss in 1785):

```
# Efficient implementation of summing the values
# from 1 to n. We assume n >= 1
def sum_to_n(n):
    return (n + 1) * n // 2
```

Because we defined sum_to_n as a function, we can just swap in this definition without changing any other code. If we'd done the implementation in-line, we'd have had to go find every instance and change it.

Return Statements

When you execute a `return` statement, you return to the calling environment. Your functions may or may not explicitly return a value.

General forms:

```
return
return expression
```

A `return` that doesn't return a value actually returns the constant None. ***Use return without a value sparingly.***

Every function has an *implicit* return at the end.

```
# Demonstrate the implicit return in functions even
# if no return written.
def print_x(x):
    print(x)
```

```
print(print_x(73))
```

→

```
73
None
```


Some More Function Examples

Suppose we want to multiply the integers from 1 to n:

```
# Return the result of multiply the values from
# 1 to n. This is the factorial function. We assume n >= 0
def multiply_to_n(n):
    result = 1
    for i in range(2, n + 1):
        result *= result
```

Convert Fahrenheit to Celsius AND Celsius to Fahrenheit :

```
# Convert degrees fahrenheit to degrees celsius.
def fahrenheit_to_celsius(degrees_f):
    return 5 / 9 * (degrees_f - 32)


# Convert degrees celsius to degrees fahrenheit.
def celsius_to_fahrenheit(degrees_c):
    return 1.8 * degrees_c + 32
```


Fahr to Celsius Table

In slideset 1, we showed the C version of a program to print a table of Fahrenheit to Celsius values. Here's a Python version:

In file fahr_to_celsius_table.py:

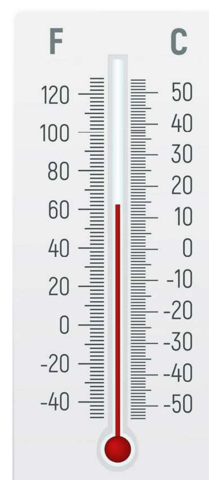
```
from function_examples import fahrenheit_to_celsius


# Print the table.
def main():
    lower_temp = -50
    upper_temp = 250
    step = 10
    # If the loop variable has meaning beyond a simple
    # counter, okay to name it something other than i, k, j.
    for degrees_f in range(lower_temp, upper_temp + 1, step):
        degrees_c = fahrenheit_to_celsius(degrees_f)
        print(format(degrees_f, "3d"), '\t',
              format(degrees_c, "5.1f"))


main()
```

Running the Temperature Program

-50	-45.6
-40	-40.0
-30	-34.4
-20	-28.9
-10	-23.3
0	-17.8
10	-12.2
20	-6.7
30	-1.1
40	4.4
50	10.0
60	15.6
70	21.1
80	26.7
90	32.2
100	37.8
110	43.3
120	48.9



Exercise: Do a similar problem converting Celsius to Fahrenheit.

A Bigger Example: Print First 100 Primes

Suppose you want to print out a table of the first 100 primes, 10 per line.

You could sit down and write this program from scratch, without using functions. But it would be a complicated mess (see section 5.8).

Better to use **functional abstraction**: find parts of the algorithm that can be coded separately and "packaged" as functions.

2	3	5	7	11	13	17	19	23	29
31	37	41	43	47	53	59	61	67	71
73	79	83	89	97	101	103	107	109	113
127	131	137	139	149	151	157	163	167	173
179	181	191	193	197	199	211	223	227	229
233	239	241	251	257	263	269	271	277	281
283	293	307	311	313	317	331	337	347	349
353	359	367	373	379	383	389	397	401	409
419	421	431	433	439	443	449	457	461	463
467	479	487	491	499	503	509	521	523	541

Print First 100 Primes: Algorithm

Here's some Python-like pseudocode to print 100 primes:

```
def print100Primes():
    primeCount = 0
    num = 0
    while (primeCount < 100):
        if (we've already printed 10 on the current line):
            go to a newline
        nextPrime = ( the next prime > num)
        print nextPrime on the current line
        num = nextPrime
        primeCount += 1
```

Note that most of this is just straightforward Python programming! The only "new" part is how to find the next prime. So we'll make that a *function*.

Top Down Development

So let's *assume* we can define a function:

```
# Return the first prime larger than n.
def get_next_prime(n):
```

in such a way that it returns the first prime larger than num.

Is that even possible?

Is there always a "next" prime larger than num?

Yes! There are an infinite number of primes. So if we keep testing successive numbers starting at num + 1, we'll eventually find the next prime. *That may not be the most efficient way!*

Value of Functional Abstraction

Notice we're following a "divide and conquer" approach: Reduce the solution of our bigger problem into one or more subproblems which we can tackle independently.

It's also an instance of "information hiding." We don't want to think about how to find the next prime, while we're worrying about printing 100 primes. Put that off! Think about one thing at a time. ***Structural decomposition.***

Next Step

Now solve the original problem, *assuming* we can write get_next_prime(n)

In file function_examples.py:

```
# Print a table of the first n primes
# 10 per line. We expect n >= 1
def print_prime_table(n):
    current_num = 1
    for i in range(1, n + 1):
        current_num = get_next_prime(current_num)
        print(format(current_num, '5d'), end=' ')
        # go to next line after every ten primes
        if i % 10 == 0:
            print()
    print()
```


Looking Ahead

Here's what the output should look like.

```
    2     3     5     7    11    13    17    19    23    29
   31    37    41    43    47    53    59    61    67    71
   73    79    83    89    97   101   103   107   109   113
  127   131   137   139   149   151   157   163   167   173
  179   181   191   193   197   199   211   223   227   229
  233   239   241   251   257   263   269   271   277   281
  283   293   307   311   313   317   331   337   347   349
  353   359   367   373   379   383   389   397   401   409
  419   421   431   433   439   443   449   457   461   463
  467   479   487   491   499   503   509   521   523   541
```

Of course, we couldn't do this if we really hadn't defined get_next_prime. So let's see what that looks like.

How to Find the Next Prime

The next prime (> num) can be found as indicated in the following pseudocode:

```
def get_next_prime(num):
    if num< 2:
        return 2 as the answer
    else:
        guess = num+ 1
        while (guess is not prime)
            guess += 1
        return guess as the answer
```

Again we solved one problem by assuming the solution to another problem: deciding whether a number is prime.

Can you think of ways to improve this algorithm?

Here's the Implementation

Note that we're assuming we can write:

```
# We assume n >= 2. Return True if n is prime,
# False otherwise.
def is_prime(n):
```

```
# Return the first prime larger than n.
def get_next_prime(n):
    if n < 2:
        return 2
    guess = n + 1
    while not is_prime(guess):
        guess += 1
    return guess
```

This works (assuming we have defined is_prime), but it's got an inefficiency. How can we make it more efficient?

Find Next Prime: A Better Version

When looking for the next prime, we don't have to test every number, just the odd numbers (after 2).

```
# Return the first prime larger than n.
def get_next_prime(n):
    if n < 2:
        return 2
    # We know n >= 2 and that no even integers
    # greater than 2 are prime. So go to the next
    # odd number and only check odd numbers.
    guess = n + 1 if n % 2 == 0 else n + 2

    # OR maybe more clearly
    # guess = n + 1
    # if guess % 2 == 0:
    #     guess = guess + 1

    while not is_prime(guess):
        guess += 2
    return guess
```

Now all that remains is to write is_prime.

Is a Number Prime?

We already solved a version of this in a previous lecture. Let's rewrite that code as a Boolean-valued function:

```
# We assume n >= 2. Return True if n is prime,
# False otherwise.
def is_prime(n):
    # Special case for 2, the only even prime.
    if n == 2:
        return True
    # Check if there are any odd divisors
    # up to the square root of the number.
    prime = n % 2 != 0
    divisor = 3
    limit = math.sqrt(n)
    while divisor <= limit and prime:
        prime = n % divisor != 0
        divisor += 2
    return prime
```


Sidetrack - Boolean "Zen"

- Did you notice this line of code in the is_prime method?

```
return prime
```

- prime is a boolean that holds the value True of False, so we simply return than value in that variable
- avoid the following: it is unnecessarily verbose

```
# YUCK!!!!
if prime == True:
    return True
else:
    return False
```


One More Example

Suppose we want to find and print k primes, starting from a given number:

In file function_examples.py:

```
# Print the first num primes after the
# value start. One prime per line.
def print_num_primes_staring_from(num, start):
    if num == 0:
        print("Request was for 0 primes")
    else:
        print('First', num, 'primes after', start, '.')
        current = start
        for i in range(num):
            current = get_next_prime(current)
            print((i + 1), current)
```

Notice that we can use functions we've defined such as get_next_prime and is_prime (almost) *as if* they were Python primitives.

Positional Arguments

This function has four formal parameters:

```
# Demo of positional arguments.
def some_function(x1, x2, x3, x4):
```

Any call to this function should have exactly four actual arguments, which are matched to the corresponding formal parameters:

```
some_function(5, 12, 5, 13)
x = 3
y = -5
some_function(x, y + 2, x * y, 12)
```

This is called using **positional** arguments.

Keyword Arguments

It is also possible to use the formal parameters as **keywords.**

```
# Demo of positional arguments.
def some_function(x1, x2, x3, x4):
    print('In some_function')
    print(x1, x2, x3, x4)
```

These two calls are equivalent:

```
some_function(5, 12, -7, 13)
some_function(x3=-7, x1=5, x4=13, x2=12)
```

```
In some_function
5 12 -7 13
In some_function
5 12 -7 13
```

Keyword Arguments

You can list the keyword arguments in any order, but all must still be specified.

```
some_function(x3=12, x1=12)
```

```
Traceback (most recent call last):
  File "C:/Users/scottm/PycharmProjects/AssignmentSolutions/SlidesCode/function_
    main()
  File "C:/Users/scottm/PycharmProjects/AssignmentSolutions/SlidesCode/function_
    some_function(x3=12, x1=12)
TypeError: some_function() missing 2 required positional arguments: 'x2' and 'x4'
```


Keyword Arguments

And even possible to mix keyword arguments with positional arguments.

The positional arguments must come first followed by the keyword.

```
some_function(5, 12, x4=13, x3=-7)
```

```
def some_function(x1, x2, x3, x4):
```

Default Parameters

You can also specify **default arguments** for a function. If you don't specify a corresponding actual argument, the default is used.

```
# Demonstrate a default argument for a parameter.
def print_rectangle_area(width=1.0, height=2.0):
    area = width * height
    print('A rectangle with a width of', width,
          'and a height of', height,
          'has an area equal to', area)
```

```
print_rectangle_area()  # uses default arguments
print_rectangle_area(4.5, 7.6)  # uses positional arguments
print_rectangle_area(height=20.5, width=5.2)  # uses keyword arguments
print_rectangle_area(4.5)  # default height
print_rectangle_area(height=10.0)  # default width
print_rectangle_area(width=5.25)  # default height
```

Do any of the built in functions we have been using have default arguments?

Using Defaults

```
A rectangle with a width of 1.0 and a height of 2.0 has an area equal to 2.0
A rectangle with a width of 4.5 and a height of 7.6 has an area equal to 34.199
A rectangle with a width of 5.2 and a height of 20.5 has an area equal to 106.6
A rectangle with a width of 4.5 and a height of 2.0 has an area equal to 9.0
A rectangle with a width of 1.0 and a height of 10.0 has an area equal to 10.0
A rectangle with a width of 5.25 and a height of 2.0 has an area equal to 10.5
```

You can mix default and non-default arguments, but must define the non-default arguments first.

```
def email(address, message=''):
```


Passing by Reference

All values in Python are objects, including numbers, strings, etc.

When you pass an argument to a function, you're actually passing a **reference** to the object, not the object itself.

There are two kinds of objects in Python:

mutable: you can change them in your program.

immutable: you can't change them in your program.

If you pass a reference to a mutable object, it can be changed by your function. If you pass a reference to an immutable object, it can't be changed by your function.

What is a Data Type?

A **data type** is a categorization of values.

Data Type	Description	Example
int	integer. An immutable number of unlimited magnitude	42
float	A real number. An immutable floating point number, system defined precision	3.1415927
str	string. An immutable sequence of characters	'Wikipedia'
bool	boolean. An immutable truth value	True, False
tuple	Immutable sequence of mixed types.	(4.0, 'UT', True)
list	Mutable sequence of mixed types.	[12, 3, 12, 7, 6]
set	Mutable, unordered collection, no duplicates	[12, 6, 3]
dict	dictionary a.k.a. maps, A mutable group of (key, value pairs)	{'k1': 2.5, 'k2': 5}

Others we likely won't use in 303e: complex, bytes, frozenset

Passing an Immutable Object

Consider the following code:

```
def increment_x(x):
    x += 1
    print('Value of x in the function increment_x =', x)


def reverse_list(lst):
    lst.reverse()
    print('list in the function reverse_list =', lst)
```

```
print()
x = 3
print('x before function call:', x)
increment_x(x)
print('x after function call: ', x)
print()

lst = [2, 3, 5, 7, 11]
print('list before function call:', lst)
reverse_list(lst)
print('list after function call: ', lst)
```

Passing Immutable and Mutable Objects - Output

```
x before function call: 3
Value of x in the function increment_x = 4
x after function call:  3

list before function call: [2, 3, 5, 7, 11]
list in the function reverse_list = [11, 7, 5, 3, 2]
list after function call:  [11, 7, 5, 3, 2]
```

Notice that the immutable integer parameter to increment_x was unchanged, while the mutable list parameter to reverse_list was changed.

Variables are mutable. They can be made to refer to different objects (values), but some objects (values) such as ints, floats, and Strings in Python are immutable.

Scope of Variables

Variables defined in a Python program have an associated **scope**, meaning the portion of the program in which they are defined.

A **global variable** is defined outside of a function and is visible after it is defined. *Use of global variables is generally considered bad programming practice. Not allowed per our 303e program hygiene guidelines.*

A **local variable** is defined within a function and is visible from the definition until the end of the function.

A local definition overrides a global definition.

Overriding

A local definition (locally) overrides the global definition.

```
x = 1                  # x is global

def func():
    x = 2              # this x is local
    print(x)           # will print 2

func()
print(x)               # will print 1
```

Running the program:

```
> python funcy.py
2

1
```

Returning Multiple Values - Useful

The Python `return` statement can also return multiple values. In fact it returns a *tuple* of values.

```
def multipleValues ( x, y ):
  return x + 1, y + 1

print( "Values returned are: ",  multipleValues ( 4, 5.2 ))

x1, x2 = multipleValues( 4, 5.2 )
print( "x1: ", x1, "\tx2: ", x2 )
```

```
Values returned are:  (5, 6.2)
x1:  5  x2:  6.2
```

You can operate on this using tuple functions, which we'll cover later in the semester, or assign them to variables.