ECS 32A - Functions

Aaron Kaloti

UC Davis - Summer Session #1 2020

UC DAVIS
COMPUTER SCIENCE

Motivation: Code Reuse

Example: Rectangle Area/Perimeter

```
length = 8
width = 5
print("The area is:", length * width)
print("The perimeter is:", length + length + width + width)

length += 2
print("=== After increasing length by 2 ===")
print("The area is:", length * width)
print("The perimeter is:", length + length + width + width)

width += 100
print("=== After increasing width by 100 ===")
print("The area is:", length * width)
print("The perimeter is:", length + length + width + width)
```

- Want to reduce number of repeated lines.
  - Loops are not ideal, due to non-duplicate lines / no pattern of change in the length/width.

Motivation: Code Reuse

Example: Rectangle Area/Perimeter

```
def print_area_and_perimeter(length,width):
    print("The area is:", length * width)
    print("The perimeter is:", length + length + width + width)

length = 8
width = 5
print_area_and_perimeter(length,width)

length += 2
print("=== After increasing length by 2 ===")
print_area_and_perimeter(length,width)

width += 100
print("=== After increasing width by 100 ===")
print_area_and_perimeter(length,width)
```

Terminology

Function

- **function**: named sequence of statements that performs a computation.

Caller

- The **caller** of the function need not be aware of the contents of the function; they just expect the function to do what the function claims it will do.

Example: `print()`

- When calling `print()`, we don't care how it's implemented.

```
print("Hello, World!")
```

Argument/Parameter

- **argument/parameter**: something **passed** as input to a function.
  - Example: Above, we pass the string "Hello, World!" to `print()`. This string is the argument.

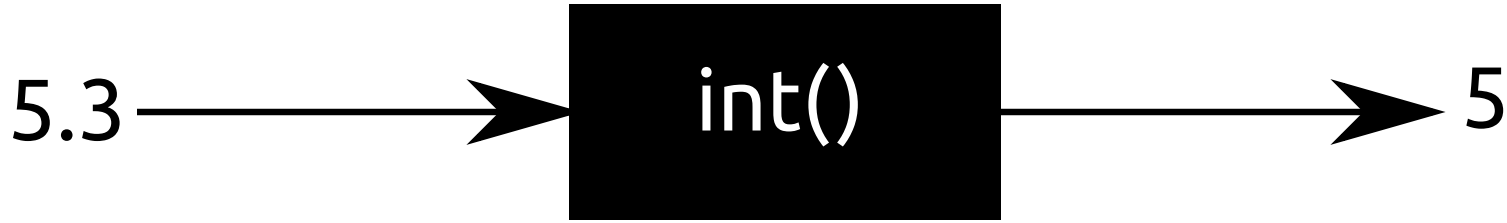
Built-In Functions

- Built-in functions such as `print()` needn't be defined by you.

`int()`

```
>>> int(5.3)
5
>>> int("2")
2
```

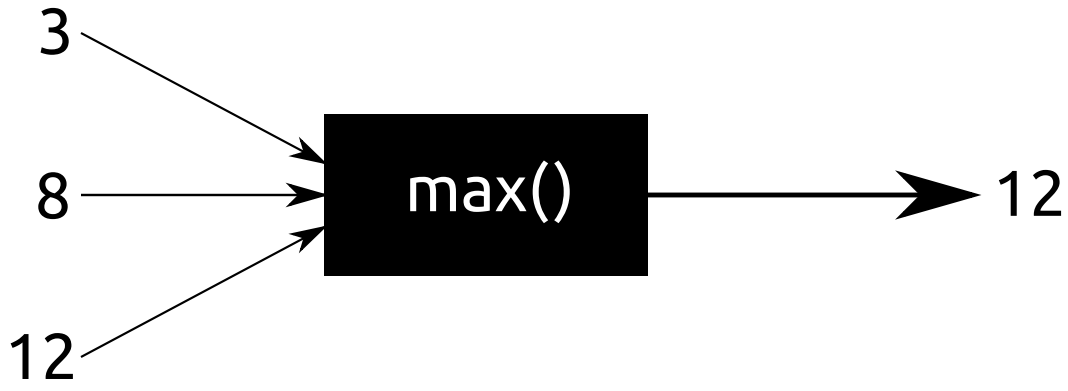
Viewing `int()` as a Black Box



Built-In Functions

`max()` and `min()`

```
>>> max(3,8,12)
12
>>> max(15,6)
15
>>> min(6,6,0,5)
0
>>> min(4,-8,3)
-8
```



Built-In Functions

Others

- `float()`
- `type()`
- `round()`
- `input()`
- and many more[1]

1. List of built-in functions: https://docs.python.org/3/library/functions.html

Defining a Function

Example

```
def hello_world():
    print("Hello, World!")

hello_world()
```

```
Hello, World!
```

Defining a Function

Example

- Can change the name.

```
def abcdefg():
    print("Hello, World!")

abcdefg()
```

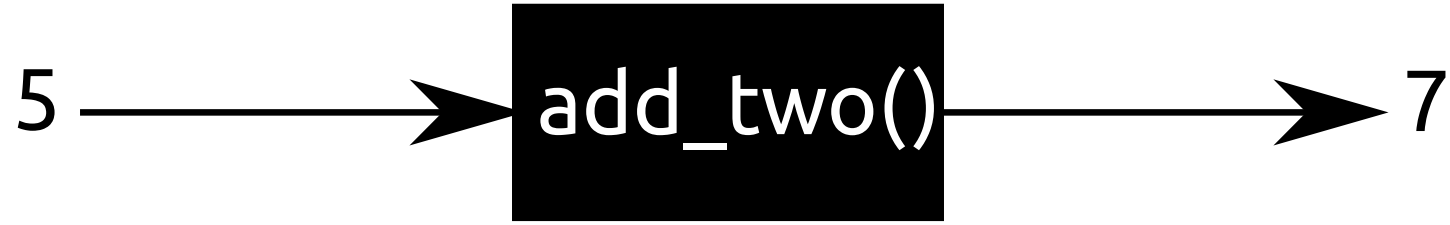
```
Hello, World!
```

Example: Add Two

Prompt

- Write a function that takes as argument an integer and returns the result of adding two to that integer.

Visualization



Solution

```
def add_two(x):
    return x + 2
```

Example: Product of Three

Prompt

- Write a function that takes three integers as arguments and returns the product of these three integers.

Solution

```
def product_three(a, b, c):
    prod = a * b * c
    return prod
```

Example: Median of Three

Prompt

- Write a function that takes three integers as arguments and returns the median of these three integers.

Solution

```
def median_three(a, b, c):
    # if @a is the median
    if b <= a <= c or c <= a <= b:
        return a
    # if @b is the median
    if a <= b <= c or c <= b <= a:
        return b
    # if @c is the median
    if a <= c <= b or b <= c <= a:
        return c
```

Can Call Function Multiple Times

Example

```
def hello_world():
    print("Hello, World!")

hello_world()
hello_world()
hello_world()
```

```
Hello, World!
Hello, World!
Hello, World!
```

Motivation: Code Reuse

Example: Rectangle Area/Perimeter (Recap)

```
def print_area_and_perimeter(length,width):
    print("The area is:", length * width)
    print("The perimeter is:", length + length + width + width)

length = 8
width = 5
print_area_and_perimeter(length,width)

length += 2
print("=== After increasing length by 2 ===")
print_area_and_perimeter(length,width)

width += 100
print("=== After increasing width by 100 ===")
print_area_and_perimeter(length,width)
```

Example: Prompt and Sum

Prompt

- Write a function `prompt_and_sum()` that takes two arguments `n` and `limit`. The function should prompt the user to enter `n` integers and return the sum of all of the integers entered that were not greater than `limit`.
  - Example: if the call `prompt_and_sum(5,8)` is made and the user enters 3, 7, 20, -1, and 9, then the returned value should be $3 + 7 + -1 = 9$.

Solution

```
def prompt_and_sum(n, limit):
    s = 0
    for i in range(n):  # range(0, n): i -> 0,1,..., n - 2, n - 1
        val = int(input("Enter: "))
        # if @val is not greater than @limit
        # if not val > limit:
        if val <= limit:
            s += val
    return s
```

Example: Count Letter 'a'

Prompt

- In the previous two sets of slides, we wrote a program that asks the user for a string and prints the number of occurrences of the lowercase "a" letter. Convert this program into a function that takes the string as argument and returns the number of occurrences.

Solution

```
def count_a(s):
    counter = 0
    for c in s:
        if c == "a":
            counter += 1
    return counter
```

None

- The value `None`, which has its own special type, is used to indicate the *absence of a value* in necessary cases, such as with a function that returns nothing.

```
>>> type(None)
<class 'NoneType'>
>>> None + 2
Traceback (most recent call last):
  File "<pyshell#39>", line 1, in <module>
    None + 2
TypeError: unsupported operand type(s) for +: 'NoneType' and 'int'
>>> None + "None"
Traceback (most recent call last):
  File "<pyshell#40>", line 1, in <module>
    None + "None"
TypeError: unsupported operand type(s) for +: 'NoneType' and 'str'
```

Void Functions

- In other words, if a function does not return a value, the default **return value** is `None`.

```
def do_nothing():
    a = 3
    b = a
    c = b

z = do_nothing()
print("z is", z)
```

do-nothing.py

Output:

```
z is None
```

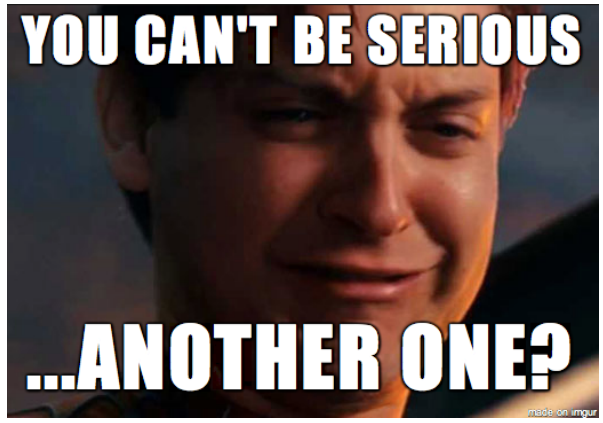
- Your book uses the term **void functions**[1] for these functions.
- In contrast, your book uses the term **fruitful functions**[2] to refer to functions that *do* return a value (i.e. that return a value that is not `None`).

1. This term will make a lot more sense once you learn C, C++, or Java.
2. On the other hand, this term is completely made up among a few Python texts, as far as I can tell.

Can Call Functions Within Loops

Example

```
while True:  # until end of time
    reboot_spider_man()
```



Example: Using Multiple Functions

Prompt

- Write a function that continously prompts the user to enter an integer until the user enters "Done". The function should then return the average of the integers entered by the user. Whenever the user is prompted to enter an integer, the function should prompt the user again if they do not enter a valid integer or "Done"; the function should use a "helper function" to do this part.
    - Example: If the user enters 5, 15, and "Done", then the function should return 10.

Example: Using Multiple Functions

Solution (Continued on Next Slide)

```
def is_valid_input(x):
    '''
    Returns True if x is valid integer or "Done";
    otherwise, returns False.
    '''
    # Check if @x is "Done".
    if x == "Done":
        return True
    # We know @x is not "Done"; check if @x is valid integer.
    # Try to do what is in the "try block". If a crash occurs
    # while doing so, then immediately move to the "except block".
    # If no crash occurs in the "try block", then the
    # "except block" is ignored.
    try:
        val = int(x)
        return True
    except:
        # int(x) conversion failed -> @x is not an integer.
        return False
...
```

Example: Using Multiple Functions

Solution

```
...
def get_valid_input():
    '''
    Keeps asking for user input until they enter valid integer
    or "Done", and then returns user input.
    '''
    inp = ""
    while not is_valid_input(inp):
        # Ask for user input.
        inp = input("Enter: ")
    return inp

def get_avg():
    s = ""
    sum = 0
    n = 0
    while s != "Done":
        s = get_valid_input()
        # After the above call, @s is either valid integer or "Done".
        if s != "Done":
            sum += int(s)
            n += 1
    return sum / n
```

Example: Triangles

- Note: I've moved this example to be later in the slides, since continuing it will make more sense after the brief discussion on boolean functions.

Example: Find a Character

- In the previous set of slides, we wrote a program that prompts the user to enter a string and a character and checked where the given character occurred in the given string. Convert this program into a function that takes the string and target character as arguments and returns the index of the first occurrence (or -1 if there's no occurrence).

Example: Something Special

- Write a function called `sum_special` that takes as argument two integers and returns the sum of the two integers. Each of the two given integers must be between -4 and 4. *However*, each integer will be given as a string, with the negative sign preceding it, if appropriate. Here are some examples of how your program should work.

```
>>> sum_special("four","three")
7
>>> sum_special("zero","two")
2
>>> sum_special("zero","-two")
-2
>>> sum_special("-three","-one")
-4
>>> sum_special("-four","-four")
-8
>>> sum_special("-four","four")
0
```

- Make sure to not duplicate the same code twice while doing this; use another function when appropriate.

Example: Replace

- In the previous deck of slides, we had no examples of where string mutability is useful. Now, we get to those examples.

Prompt

- Write a function that takes as argument a string and returns a string that is identical to the given string, except that all instances of "a" are replaced by "X".
  - For example, if "abca" is passed to the function, then the function should return "XbcX".

Example: Reverse Abc

Prompt

- Write a function called `reverse_abc` that takes as argument a string and returns a copy of that string with *all* instances of "abc" reversed.
  - Examples:
    - `reverse_abc("abc")` should return "cba".
    - `reverse_abc("xxabcxxabc")` should return "xxcbaxxcba".
    - `reverse_abc("abbc")` should return "abbc".

Boolean Function

- A **boolean function** is a function that only returns a boolean value, i.e. `True` or `False`.

```
def is_less_than_5(val):
    if val < 5:
        return True
    else:
        return False
```

is-less-than-5.py

- Sample output:

```
>>> is_less_than_5(8)
False
>>> is_less_than_5(3)
True
```

- This is not a boolean function.

```
def blah(val):
    if val < 5:
        return "abc"
    else:
        return "def"
```

Boolean Function

```
def is_less_than_5(val):
    if val < 5:
        return True
    else:
        return False
```

is-less-than-5.py

- You may tend to see boolean functions used like this:

```
val = int(input("Enter val: "))
if is_less_than_5(val):
    print("{} is less than 5!".format(val))
```

is-less-than-5-in-condition.py

```
==== RESTART: ... ====
Enter val: 5
>>>
==== RESTART: ... ====
Enter val: -3
-3 is less than 5!
```

- A boolean function can also be used with `not` as well:

```
val = int(input("Enter val: "))
if not is_less_than_5(val):
    print("{} is NOT less than 5!".format(val))
```

Example: Triangles

Prompt

Exercise 5.3. *If you are given three sticks, you may or may not be able to arrange them in a triangle. For example, if one of the sticks is 12 inches long and the other two are one inch long, you will not be able to get the short sticks to meet in the middle. For any three lengths, there is a simple test to see if it is possible to form a triangle:*

> *If any of the three lengths is greater than the sum of the other two, then you cannot form a triangle. Otherwise, you can. (If the sum of two lengths equals the third, they form what is called a "degenerate" triangle.)*

1. *Write a function named* `is_triangle` *that takes three integers as arguments, and that prints either "Yes" or "No", depending on whether you can or cannot form a triangle from sticks with the given lengths.*

2. *Write a function that prompts the user to input three stick lengths, converts them to integers, and uses* `is_triangle` *to check whether sticks with the given lengths can form a triangle.*

Example: Triangle

Solution: `is_triangle()`

```
# Assumes @a, @b, and @c are positive.
def is_triangle(a, b, c):
    # If the first length is greater than the sum of the other two.
    if a > b + c:
        print("No")
    # If the second length exceeds the sum of the other two.
    elif b > a + c:
        print("No")
    # If the third length exceeds the sum of the other two.
    elif c > a + b:
        print("No")
    else:
        print("Yes")
```

- We'll get to the other function(s) on Tuesday (7/14).

Example: Triangles (Revisited)

Prompt

- Rewrite `is_triangle()` to be a boolean function.
- Implement `try_triangle()` for the second part.

Example: New Password

- In the previous set of slides, we wrote a program that prompts the user to enter a new password and tells the user if this password fulfills the following requirements:
  - At least one capital letter.
  - At least two digits.
  - At least one non-alphanumeric character (e.g. a dollar sign).
  - At least eight letters.
- Modify this program so that it *keeps* prompting the user to enter a new password until they enter a valid one.
  - *Hint*: You wil find functions helpful.

Dead Code

- **WARNING: What you are about to see may be very disturbing.**
- What is the value of `a` after this program ends?

```
def foo():
    return 5
    j = -30
    while j < 8:
        i = 14
        if j <= -100:
            while j < -100:
                j += 100
            continue
        while i > j and j < i:
            if i == 11:
                break
            i += 20
        while j < 30:
            if j % i == 8:
                for k in range(5,8):
                    j = i - k
            j += 1
        i *= j

a = foo()
```

- `return` ends the function *immediately*, so `a` will have the value 5.

`print()` vs. `return`

```
def hello_world():
    print("Hello, World!")

a = hello_world()
```

- What will be the value of `a` after this program finishes?

`print()` vs. `return`

```
def hello_world():
    print("Hello, World!")

a = hello_world()
print("a is", a)
```

print-vs-return.py

- What will be the value of `a` after this program finishes?
- Let's add the above `print()` statement so that we can find out.

Output:

```
Hello, World!
a is None
```

`print()` vs. `return`

- To again emphasize the distinction,
  - `print()`: send a message to the screen
    - example: `print(16)` sends the message "16" to the screen
  - `return`: immediately ends the current function and sends the value back
    - example: `return 8` sends 8 back
- Notice that there is no mention of `print()` sending any value back. That is because it does not. If `print()` sent a value back, then why does `c` have no value when we do this?

```
>>> c = print(8)
8
>>> c
>>> print(c)
None
```

`print()` vs. `return`

- On the other end, `return` does not send a value to the screen. The program below prints nothing to the screen.

```
def foo():
    return 5

a = foo()
```

- However, the program below *does*.

```
def foo():
    return 5

a = foo()
print(a)
```

- Notice the key difference (i.e. the use of `print()`). If your program does not call `print()`, nothing will be printed in the screen. That is a hard fact as far as the coverage in this class is concerned.

`print()` vs. `return`

- Part of the confusion between these two concepts comes from the Interpreter mode. Suppose we have the below two functions.

```
def foo():
    print(8)
def goo():
    return 8
```

print-vs-return3.py

- Calling either of the above on the interpreter, as shown below, results in 8 being printed, so it *seems* that `print` and `return` are the same thing.

```
>>> foo()
8
>>> goo()
8
```

- But try to assign the `return` value to a variable:

```
>>> a = foo()
8
>>> b = goo()
>>> print(a)
None
>>> print(b)
8
```

`print()` vs. `return`

- Why, then, *does* 8 get printed to the screen on Interpreter mode?

```
def foo():
    print(8)

def goo():
    return 8
```

print-vs-return3.py

```
>>> foo()
8
>>> goo()
8
```

- The reason is that Interpreter mode works differently than a script in one key way: standalone values are printed to the screen.

`print()` vs. `return`

- In the Interpreter output below, all of the commands are standalone values -- or expressions that *evaluate* to standalone values -- that are consequently printed to the screen, even without use of `print()`.

```
>>> print(8)
8
>>> print(15)
15
>>> a = 20
>>> print(a)
20
>>> print(5 == 7)
False
```

```
>>> 8
8
>>> 15
15
>>> a = 20
>>> a
20
>>> 5 == 7
False
```

- If I put the above four commands into a program, the output would be absolutely nothing, because `print()` is never called.

```
8
15
a = 20
a
5 == 7
```

print-vs-return2.py

Output:

```
[nothing]
```

`print()` vs. `return`

- As discussed earlier in this lecture, if a function does not *explicitly* return anything, then that function returns `None`.

```
def foo():
    print(13)
```

- `foo()` does not return any value. It prints 13, but it does not return *anything*, so the result of the statement `a = foo()` is that `a` becomes `None`.

`print()` vs. `return`

- When should you use one over the other?
- It's not the right question; they are unrelated tools for unrelated purposes, and you can use neither, one, or even both. You might think it is akin to asking when you should use `for` loops vs. `while` loops, but it is truly more akin to asking when you should use `for` loops vs. conditional statements; `print()` and `return` have their own uses.

```
def foo():
    print(8)

a = foo()
print("a =", a)
```

print-vs-return5.py

```
8
a = None
```

```
def foo():
    print(8)
    return 8

a = foo()
print("a =", a)
```

print-vs-return4.py

```
8
a = 8
```

`print()` vs. `return`

- To recap the differences:

1. **When a function prints a value, that says *nothing* about what the function *returns*. Printing a value and returning a value are two different concepts.**

```
def foo():
    print(14)
    return 10
```

```
>>> x = foo()
14
>>> x
10
```

2. **Printing a value does not necessarily involve a function; returning a value *does*.**

```
>>> print(15)
15
>>> return 15
SyntaxError: 'return' outside function
```

3. **Printing a value does not *end* a function. Returning a value *does*. A function can print messages as many times as it likes.**

4. **The value returned by a function may never even be printed to the screen.**

`print()` vs. `return`

- One final note: when you are told to implement a function, the directions must *always* be clear about whether you are to print a certain value(s), return a certain value, or both. If the directions do not specify this, then those directions are underspecified, and you could ask for clarification, but usually, the default is to assume the function is meant to `return` the value that it generated. Otherwise, what would be the point?
    - Recall the `count_a` function that we previously implemented. If that function *printed* the count instead of returning it, then we -- the caller of `count_a()`, would be *unable* to use that value (the count of lowercase a's) that `count_a` generated. Sure, the value would get printed to the screen (which we may not even have wanted...), but what if we wanted to compare that value to another value of some sort? `count_a()` would be useless.

Exercise: Game

- Write a program that asks the user for the intial state of a game and then continues to ask the user for commands (i.e. "left", "right", or "end") that tell the game where to move the user's "character". The state of the game is a string consisting of x's, with one of those characters instead being an "A"; this is the user's character. For example, if the user enters "xxA" as the initial state and then enters "left", the state should become "xAx"; if the user enters "left" again, the state should become "Axx"; and then if the user enters "left" yet again, the state should not change. Entering "right" causes the opposite behavior. Entering "end" should end the function.

Exercise: Game

- Here is an example of how your function should behave.

```
>>> game()
Enter initial state: xxAxx
Current state: xxAxx
Enter command: left
Current state: xAxxx
Enter command: right
Current state: xxAxx
Enter command: right
Current state: xxxAx
Enter command: right
Current state: xxxxA
Enter command: right
Current state: xxxxA
Enter command: left
Current state: xxxAx
...
```

```
Enter command: left
Current state: xxAxx
Enter command: right
Current state: xxxAx
Enter command: left
Current state: xxAxx
Enter command: left
Current state: xAxxx
Enter command: left
Current state: Axxxx
Enter command: left
Current state: Axxxx
Enter command: right
Current state: xAxxx
Enter command: end
```

- You should make a function to handle leftward movement and a function to handle rightward movement.

Modules

- **module**: set of variables and functions that can be `import`ed into your program for you to use. They are provided like built-in functions (like `max()`, `print()`, `int()`, etc.), but in the case of a module, you must manually `import` the module before using it.

`math` Module

Interpreter

```
>>> import math
>>> math.ceil(8.1)
9
>>> math.ceil(8.5)
9
```

In a Program

```
import math
print(math.ceil(8.1))
print(math.ceil(8.5))
```

Modules

`math` Module

- Other variables/functions from the `math` module[1]:

```
>>> import math
>>> math.factorial(4)
24
>>> math.sqrt(9)
3.0
>>> math.cos(45)
0.5253219888177297
>>> math.pi
3.141592653589793
```

1. More can be found here: https://docs.python.org/3/library/math.html

Modules

`random` Module

- Provides functions to help with dealing with random[1] numbers.

```
>>> import random
>>> random.randint(2,5)
5
>>> random.randint(2,5)
2
>>> random.randint(2,5)
2
>>> random.randint(2,5)
4
>>> random.randint(2,5)
2
>>> random.randint(2,5)
4
>>> random.randint(2,5)
3
>>> random.randint(2,5)
3
```

1. Whenever you see "random" in a programming context, that means "pseudorandom". It is impossible to generate *truly* random numbers, but it *is* possible to generate a distribution of numbers such that the probability of generating any given number is *extremely* close to being equal to the probability of generating another given number.

Modules

`random` Module

Seeding

- *Seeding* allows you to generate the same stream of random numbers, in case you need to repeat a stream for whatever reason. If I ever happened to give an assignment using the `random` module, I would have the autograder perform seeding.

```
>>> random.seed(15)
>>> random.randint(100,1000000)
219231
>>> random.randint(100,1000000)
12320
>>> random.seed(15)
>>> random.randint(100,1000000)
219231
>>> random.randint(100,1000000)
12320
>>> random.seed(18)
>>> random.randint(100,1000000)
190169
```

- See here (https://docs.python.org/3/library/random.html) for more information about the `random` module, including functions meant to facilitate sampling from a variety of distributions (or from a set).

Modules

`sys` Module: `exit()`

- `exit()` from `sys` module immediately ends program.

Example

Code

```
import sys
print("AAA")
sys.exit()
print("BBB")
```

Output

```
AAA
```

There is a "built-in" `exit()` that is supposed to only be used in the Interpreter mode. If you call it, Python IDLE will ask if you want to close the entire Python IDLE window. I say "built-in" in quotes because technically, this `exit()` is *not* "built-in" and is in fact from the `site` module, and the `site` module is automatically imported. Source: https://docs.python.org/3/library/constants.html#exit

Modules

`sys` Module: `exit()`

- Shouldn't[1] use in functions; still ends program (not just the function).

Example

Code

```
import sys

def foo():
    print("AAA")
    sys.exit()
    print("BBB")

foo()
```

Output

```
AAA
```

1. Unless you're sure you want to.

Modules

Example: os Module

Modules

Importing Something Specific

- Importing an entire module -- or multiple entire modules, as will be done in a real-world Python program -- is importing a lot of variables and functions, many of which you are unlikely to use.

Example

```
>>> from math import sqrt
>>> sqrt(9)
3.0
```

- Didn't import `cos()`...

```
>>> cos(45)
Traceback (most recent call last):
  File "<pyshell#3>", line 1, in <module>
    cos(45)
NameError: name 'cos' is not defined
>>> math.cos(45)
Traceback (most recent call last):
  File "<pyshell#4>", line 1, in <module>
    math.cos(45)
NameError: name 'math' is not defined
```

Local Variables

Global Variables

Appendix: Why Functions

- I won't talk about these more specifically, but here are our primary textbooks' reasons for why we should use functions, according to pages 24-25, on section 3.11:
  - Creating a new function gives you an opportunity to name a group of statements, which makes your program easier to read and debug.
  - Functions can make a program smaller by eliminating repetitive code. Later, if you make a change, you only have to make it in one place.
  - Dividing a long program into functions allows you to debug the parts one at a time and then assemble them into a working whole.
  - Well-designed functions are often useful for many programs. Once you write and debug one, you can reuse it.

Appendix: More on Scope

- As is consistent with the previous lecture, the rules on scope apply even with functions that have strings as arguments.

```
def foo(string):
    string = "abc"

chars = "abcde"
foo(chars)
print(chars)
```

- Once we get to lists, this will get more complicated, but if you understand the rules of scope and string immutability well, you'll find no inconsistencies.