Functional Decomposition: Part 1

- Defining new functions, calling functions you have defined
- Declaring variables that are local to a function
- Scope: local vs. global
- Function specific style requirements (rules of thumb for good style)

Built In Python Functions

- Python comes with many functions that are a built in part of the language e.g., ‘`print()`’, ‘`input()`’
  - They either come ‘automatically’ or you can access that module/library with an `import`.
- (If a program needs to perform a common task e.g., finding the absolute value of a number, then you should first check if the function has already been implemented).
- For a list of all prewritten Python functions.
  - https://docs.python.org/3/library/functions.html

Writing Your Own Functions: Why Do It?

- **First reason, you have no choice**: the code hasn't been implemented for this feature yet.
- Example: you can't just look up the prebuilt functions in python and have one of them do all the work for one of your assignments.

Writing Your Own Functions: Why Do It?

- **Second reason, you need to know this**: it's not only done all the time in real life but it's a key component of this course.
- (Exert from the university calendar description):
  - "Introduction to problem solving, analysis and design of small-scale computational systems and **implementation using a procedural programming language**. "
  - **All this means that it is expected that all students who have successfully finished this course will be able to properly implement a non-trivial program not only using functional decomposition but also apply important related concepts such as: parameters, return values and scope.**
- New terminology:
  - Function, procedure, method
  - For now you can think of them as largely interchangeable although you will learn the difference between a function and method towards the end of this course.
    - Most languages don't distinguish procedures from functions.

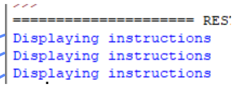
Examinations

- You have to know the terms ‘function’/’procedure’ (and eventually ‘method’).
- But you don’t have to memorize the first two reasons (just covered) for using functional decomposition for the exam.
- For the exam: You do need to know the other reasons (#3 – 7) that come from functional decomposition that will immediately follow in these notes.

Writing Your Own Functions: Why Do It?

- **Third reason, reuse/efficiency**: Once the function definition is complete (and tested reasonably) it can be called (reused) many times.

```
def displayInstructions():
    print("Displaying instructions")

# Main body of code (starting execution point)
displayInstructions()
displayInstructions()
displayInstructions()
```



```
====================== RES
Displaying instructions
Displaying instructions
Displaying instructions
```

- Think about how many times prewritten functions such as `input` and `print` have be used.

Writing Your Own Functions: Why Do It?

- **Fourth reason, easier maintenance**: (related to the previous benefit: write once, use many times): when program maintenance (changes to code) is needed.
- If the same code is written over and over again in different parts of the program then each location must be changed.
- Implementing that same code in one function requires only changes to the code in that function.

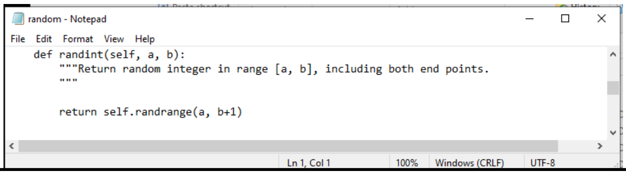
```
def myFunction():
    #Code to modify
```

```
#My program, no functions
#Code to modify


#Code to modify

#Code to modify
```

- This may result in a smaller program with fewer/no redudancies as well.

Writing Your Own Functions: Why Do It?

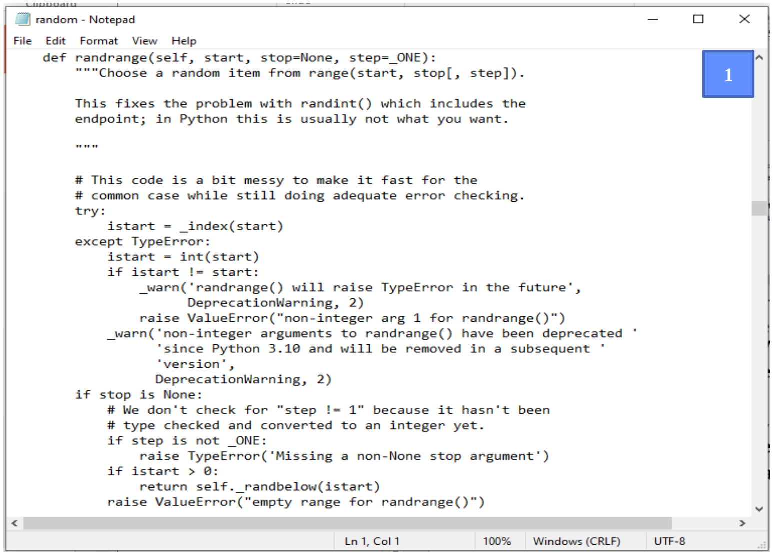
- **Fifth reason, decoupling of your code**:
- New terminology, decoupling: a fancy term for a simple concept.
- In this case it means you can simply use a function without worrying about the ‘internal’ details of how it was written.
- You simply need things such as: how to call it, what operations the function implements, what are it’s return values etc.
- This is the actual code from the `randint()` function.
  - You just have to know how to call it not know all the intimate details of how every line works.



```
    def randint(self, a, b):
        """Return random integer in range [a, b], including both end points.
        """

        return self.randrange(a, b+1)
```

Writing Your Own Functions: Why Do It?

- More Of The Random Library/Module

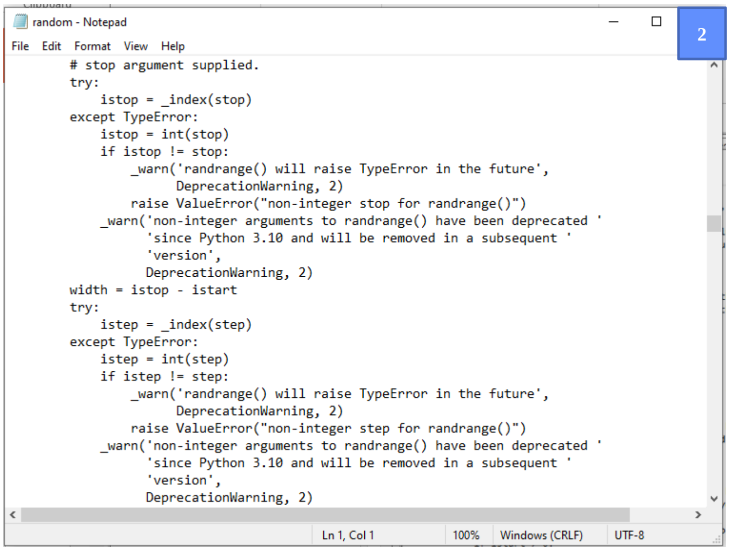


```
    def randrange(self, start, stop=None, step=_ONE):
        """Choose a random item from range(start, stop[, step]).

        This fixes the problem with randint() which includes the
        endpoint; in Python this is usually not what you want.

        """

        # This code is a bit messy to make it fast for the
        # common case while still doing adequate error checking.
        try:
            istart = _index(start)
        except TypeError:
            istart = int(start)
            if istart != start:
                _warn('randrange() will raise TypeError in the future',
                      DeprecationWarning, 2)
                raise ValueError("non-integer arg 1 for randrange()")
            _warn('non-integer arguments to randrange() have been deprecated '
                  'since Python 3.10 and will be removed in a subsequent '
                  'version',
                  DeprecationWarning, 2)
        if stop is None:
            # We don't check for "step != 1" because it hasn't been
            # type checked and converted to an integer yet.
            if step is not _ONE:
                raise TypeError('Missing a non-None stop argument')
            if istart > 0:
                return self._randbelow(istart)
            raise ValueError("empty range for randrange()")
```

James Tam

Writing Your Own Functions: Why Do It?

- More Of The Random Library/Module

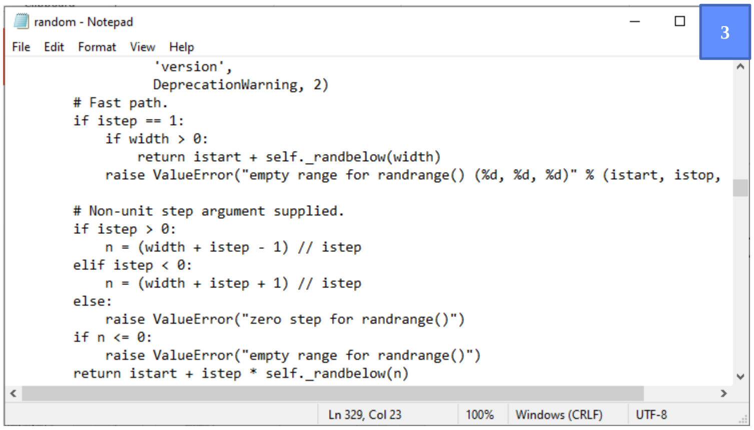


```
        # stop argument supplied.
        try:
            istop = _index(stop)
        except TypeError:
            istop = int(stop)
            if istop != stop:
                _warn('randrange() will raise TypeError in the future',
                      DeprecationWarning, 2)
                raise ValueError("non-integer stop for randrange()")
            _warn('non-integer arguments to randrange() have been deprecated '
                  'since Python 3.10 and will be removed in a subsequent '
                  'version',
                  DeprecationWarning, 2)
        width = istop - istart
        try:
            istep = _index(step)
        except TypeError:
            istep = int(step)
            if istep != step:
                _warn('randrange() will raise TypeError in the future',
                      DeprecationWarning, 2)
                raise ValueError("non-integer step for randrange()")
            _warn('non-integer arguments to randrange() have been deprecated '
                  'since Python 3.10 and will be removed in a subsequent '
                  'version',
                  DeprecationWarning, 2)
```

James Tam

Writing Your Own Functions: Why Do It?

- More Of The Random Library/Module



```
                  'version',
                  DeprecationWarning, 2)
        # Fast path.
        if istep == 1:
            if width > 0:
                return istart + self._randbelow(width)
            raise ValueError("empty range for randrange() (%d, %d, %d)" % (istart, istop,

        # Non-unit step argument supplied.
        if istep > 0:
            n = (width + istep - 1) // istep
        elif istep < 0:
            n = (width + istep + 1) // istep
        else:
            raise ValueError("zero step for randrange()")
        if n <= 0:
            raise ValueError("empty range for randrange()")
        return istart + istep * self._randbelow(n)
```

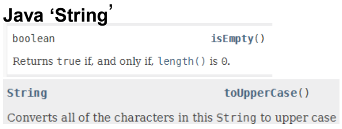
Writing Your Own Functions: Why Do It?

- **Sixth reason, it simplifies things.**
  - Increased readability: Allows you to focus on one part of a program at a time (thus reduced complexity).
  - Program design/implementation is easier:
    - Sometimes you will have to write a program for a large and/or complex problem.
    - One technique employed in this type of situation is the top-down approach to design (coming later in the functional decomposition notes)
      - The main advantage is that it reduces the complexity of the problem because you only have to work on it a portion at a time.

Functional decomposition goes hand-in-hand with good programming style and proper documentation.

- If you apply good style introduced in this section (e.g. each function implements a single well-defined task – more on this later) it helps make it clear which function you should be looking at when you want to use pre-written code.
- Proper documentation indicates how a function should and should not be used.

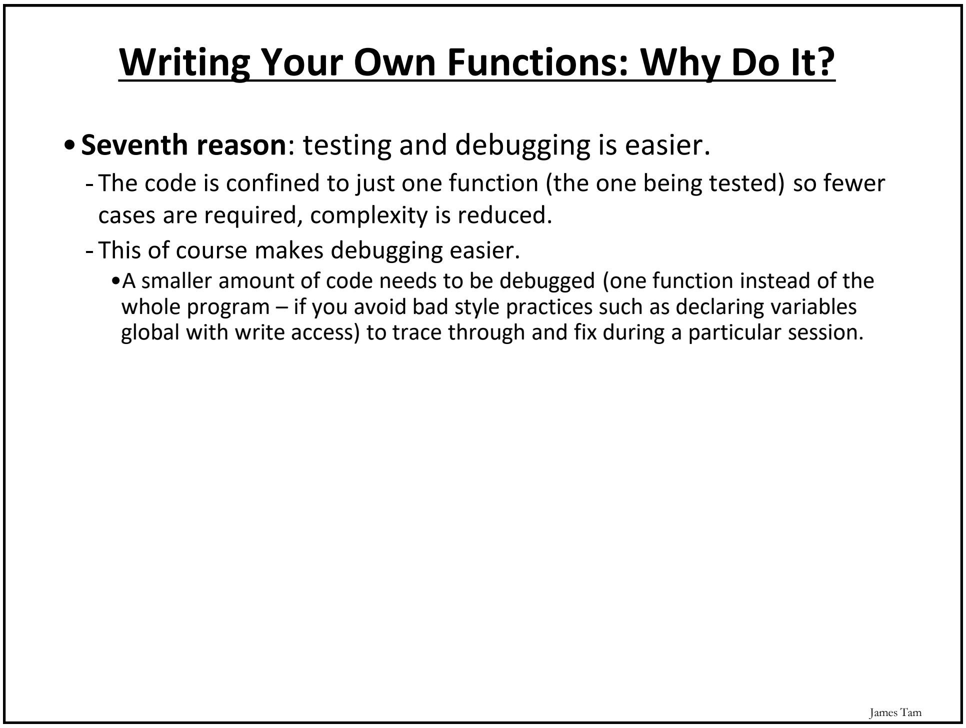
Java 'String'



boolean	isEmpty()
Returns true if, and only if, length() is 0.	
String	toUpperCase()
Converts all of the characters in this String to upper case	

Example function (you could write)

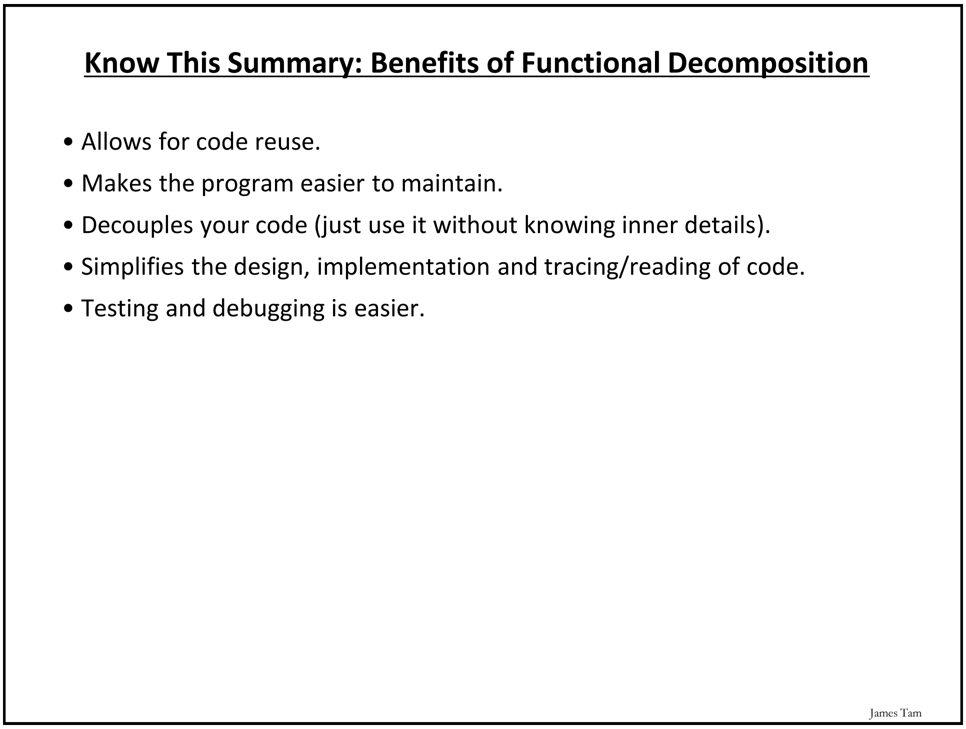
```
divide(float,float)
  Parameters: two floating point numbers
  Returns: a float (quotient of the numbers)
  Assumptions: 2nd parameter not zero.
```



Writing Your Own Functions: Why Do It?

- **Seventh reason**: testing and debugging is easier.
  - The code is confined to just one function (the one being tested) so fewer cases are required, complexity is reduced.
  - This of course makes debugging easier.
    - A smaller amount of code needs to be debugged (one function instead of the whole program – if you avoid bad style practices such as declaring variables global with write access) to trace through and fix during a particular session.

James Tam

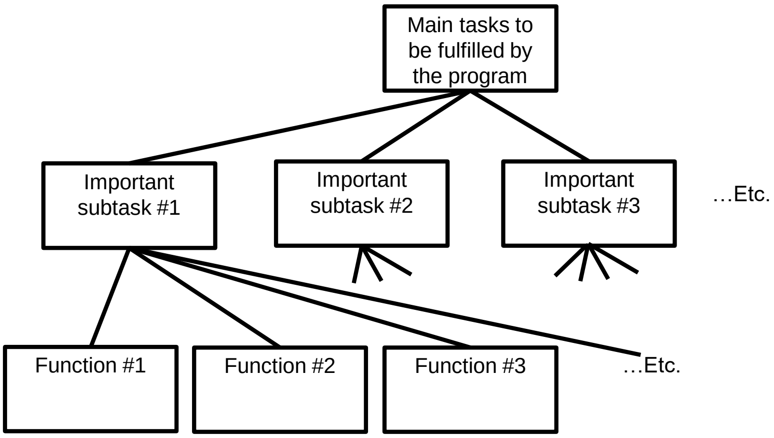


Know This Summary: Benefits of Functional Decomposition

- Allows for code reuse.
- Makes the program easier to maintain.
- Decouples your code (just use it without knowing inner details).
- Simplifies the design, implementation and tracing/reading of code.
- Testing and debugging is easier.

James Tam

Simplying A Problem With Functional Decomposition

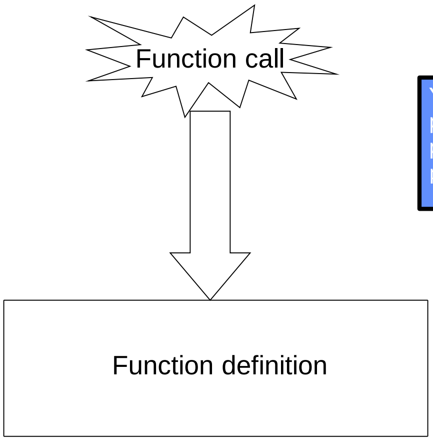


Similar to creating a document: don't start coding until you are done decomposing the structure.

Things Needed In Order To Use Functions

- Function call (you've done this before)
  - Actually running (executing) the function.
  - You have already done this second part many times because up to this point you have been using functions that have already been defined by someone else e.g., `print(), input()`
- Function definition (this is what you will learn)
  - Instructions that indicate what the function will do when it runs.
  - Before this section: you have used built-in python functions (with their instructions already written by someone else).
  - In this section: you will learn how to write the instructions inside a function body which execute when that function runs.

Functions (Basic Case: No parameters/Inputs)



You've already called prebuilt functions and passed no arguments e.g. `print(), input()`

James Tam

Defining A Function

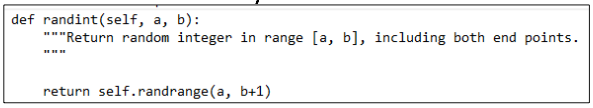
- **Format:**

```
def <function name>1():
    body2
```

- **Example:**

```
def displayInstructions():
    print ("Displaying instructions on how to use the
      program")
```

- You don't need to define prebuilt functions because some else has defined the code for you.



```
def randint(self, a, b):
    """Return random integer in range [a, b], including both end points.
    """

    return self.randrange(a, b+1)
```

[1] Functions should be named according to the rules for naming variables (all lower case alphabetic, separate multiple words via camel case or by using an underscore).

[2] Body = the instruction or group of instructions that execute when the function executes (when called).

The rule in Python for specifying the body is to use indentation.

James Tam

Calling A Function

- **Format:**

  `<function name>()`

- **Example**:

  `displayInstructions()`

- As you mentioned you have already learned how to call a prewritten function e.g. `print(), int(), input(), randint(1,6) etc.`

Quick Recap: Starting Execution Point

- The program starts at the first executable instruction that is not indented.
- In the case of your programs thus far all statement have been un-indented (save loops/branches) so it's just the first statement that is the starting execution point.

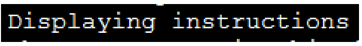
```
HUMAN_CAT_AGE_RATIO = 7
age = input("What is your age in years: ")
catAge = age * HUMAN_CAT_AGE_RATIO
…
```

- But note that the body of functions MUST be indented in Python.

Functions: An Example That Puts Together All The Parts Of The Easiest Case

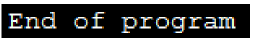
- **Name of the example program**: `1firstExampleFunction.py`
  - Learning objective:

```
def displayInstructions():
    print("Displaying instructions")
```



```
Displaying instructions
```

```
# Main body of code (starting execution point, not indented)
displayInstructions()
print("End of program")
```



```
End of program
```

Functions: An Example That Puts Together All The Parts Of The Easiest Case

- **Name of the example program**: `1firstExampleFunction.py`

```
def displayInstructions():
    print("Displaying instructions")
```

(Something new in this section): Function definition

```
# Main body of code (starting execution point)
displayInstructions()
print("End of program")
```

(You've done this before): Function call

Defining The Main Body Of Code As A Function

- Good style: unless it's mandatory, all instructions must be inside a function.
- Rather than defining instructions outside of a function the main starting execution point can also be defined explicitly as a function.
- (The previous program rewritten to include an explicit start function)
  Example program: `2firstExampleFunctionV2.py`
  - Learning objective: enclosing the start of the program inside a function

```
def displayInstructions():
    print ("Displaying instructions")

def start():
    displayInstructions()
    print("End of program")
```

- **Important:** If you explicitly define the starting function then do not forgot to explicitly call it!

```
start ()
```

Don't forget to start your program!
Program starts at the first executable un-indented instruction

Stylistic Note

- By convention the starting function is frequently named '`main()`' or in my case '`start()`'.

```
def main():
```

- OR

```
def start():
```

- Another convention is to define and call this function at the very end of your program.
- Both of these things are is done so the reader can quickly find the beginning execution point.

Creating Your Variables: Inside Functions

- Before this section of notes: all statements (including the creation of a variables) occur outside of a function

```
HUMAN_CAT_AGE_RATIO = 7
age = input("What is your age in years: ")
catAge = age * HUMAN_CAT_AGE_RATIO
…
```

- Now that you have learned how to define functions, **ALL your variables must be created with the body of a function**.
- Constants can still be created outside of a function (more on this later).

'Outside': OK for constants only

```
HUMAN_CAT_AGE_RATIO = 7

def getInformation():
    age = input("What is your age in years: ")
    catAge = age * HUMAN_CAT_AGE_RATIO
```

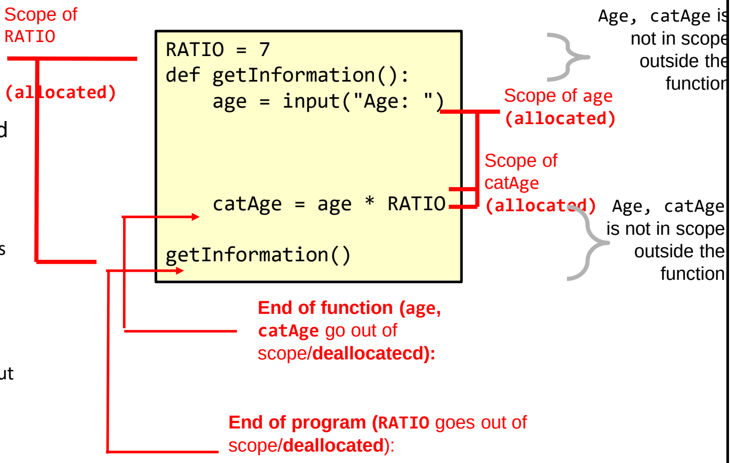
Inside function body: all variables (e.g. 'age', 'catAge') must be here

Local Variables

- Characteristics
  - Locals only get allocated (created in memory) when the function is called.
  - Locals get de-allocated (unavailable in memory) when the function ends.
- Benefits (why create them this way)
  - 1st: more efficient use of memory
  - 2nd: minimize the occurrence of side effects of global variables
    - This is the main reason why it's regarded as bad style in actual practice.
    - But details are more complex so the explanation will come later.
  - 3rd: pedagogical (creating variables locally forces you to apply important programming concepts such as parameter passing, function return values and scope).

Scope: Visually Showing When Memory Locations Can Be Accessed

- The scope of an identifier (variable, constant) is where it may be accessed and used.
- In Python[1]:
  - An identifier comes into scope (becomes visible to the program and can be used) after it has been declared.
  - An identifier goes out of scope (no longer visible so it can no longer be used) at the end of the indented block where the identifier has been declared.



```
RATIO = 7
def getInformation():
    age = input("Age: ")


    catAge = age * RATIO

getInformation()
```

Scope of RATIO (allocated)

Scope of age (allocated)

Scope of catAge (allocated)

Age, catAge is not in scope outside the function

Age, catAge is not in scope outside the function

End of function (age, catAge go out of scope/**deallocatecd):**

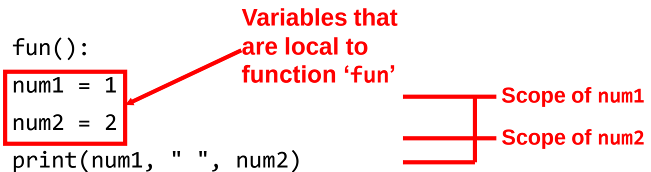
End of program (RATIO goes out of scope/**deallocated**):

[1] The concept of scoping (limited visibility) applies to all programming languages. The rules for determining when identifiers come into and go out of scope will vary with a particular language.

James Tam

Working With Local Variables: Putting It All Together

- **Name of the example program**: 3secondExampleFunction.py
  - Learning objective: creating/defining variables that only exist while a function runs (local to that function).

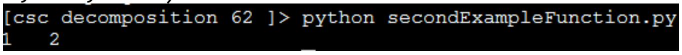


```
def fun():
    num1 = 1
    num2 = 2
    print(num1, " ", num2)
```

Variables that are local to function 'fun'

Scope of num1

Scope of num2



```
[csc decomposition 62 ]> python secondExampleFunction.py
1    2
```

```
# start function
fun()
```

James Tam

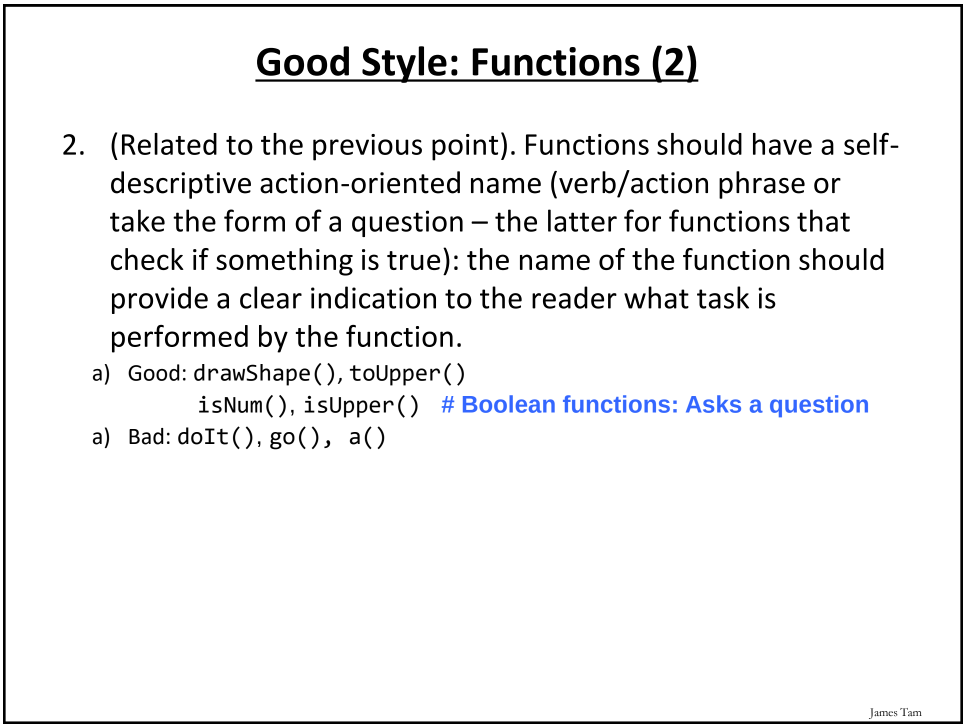
Variables Vs. Named Constants

- As you have already been taught:
  - Variables can change as the programs run while named constants don’t change after they’ve been set to the initial value.
  - To visually distinguish the two variables use lower case while constants are capitalized.
- Your program should consistently distinguish the two!
  - The following is only a ‘constant’ in name only and is treated like a variable.

```
PI = 3.14
radius = 10
area = PI * (radius ** 2)

PI = 3.1 #Do not change the value in a constant!
```

Good Style: Functions

1. Each function should have one well defined task. If it doesn’t then this may be a sign that the function should be decomposed into multiple sub-functions.
   a) Clear function: A function that squares a number.
   b) Ambiguous function: A function that calculates the square and the cube of a number.
      - Writing a function that is too specific makes it less useful (in this case what if we wanted to perform one operation but not the other).
   - Also functions that perform multiple tasks can be harder to test.



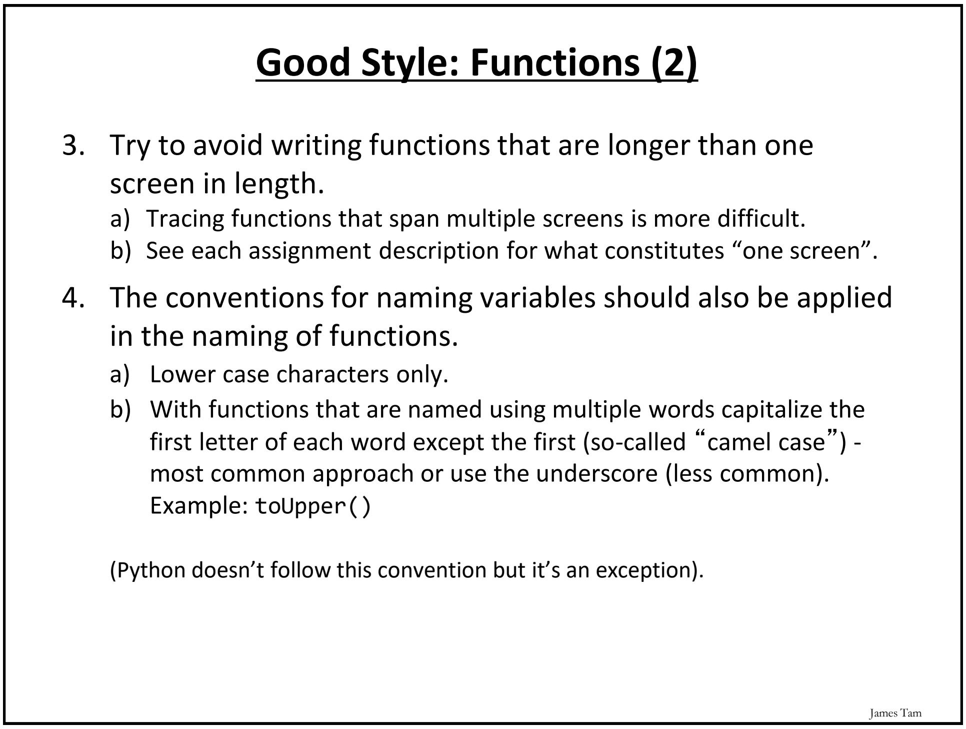
Good Style: Functions (2)

2. (Related to the previous point). Functions should have a self-descriptive action-oriented name (verb/action phrase or take the form of a question – the latter for functions that check if something is true): the name of the function should provide a clear indication to the reader what task is performed by the function.

   a) Good: `drawShape()`, `toUpper()`
      `isNum()`, `isUpper()`   # Boolean functions: Asks a question

   a) Bad: `doIt()`, `go()`, `a()`

James Tam



Good Style: Functions (2)

3. Try to avoid writing functions that are longer than one screen in length.
   a) Tracing functions that span multiple screens is more difficult.
   b) See each assignment description for what constitutes “one screen”.

4. The conventions for naming variables should also be applied in the naming of functions.
   a) Lower case characters only.
   b) With functions that are named using multiple words capitalize the first letter of each word except the first (so-called “camel case”) - most common approach or use the underscore (less common). Example: `toUpper()`

(Python doesn’t follow this convention but it’s an exception).

James Tam

After This Section You Should Now Know

- How and why the top down approach can be used to decompose problems
  - What is procedural programming
- How to write the definition for a function
- How to write a function call
- How and why to declare variables locally
- Function specific style requirements

Copyright Notification

- Unless otherwise indicated, all images in this presentation were provided courtesy of James Tam.