# PROCEDURES; ITERATION

LECTURE 03-1

JIM FIX, REED COLLEGE CSCI 121

## SUMMARY OF USER-DEFINED FUNCTIONS

- A function’s code consists of an indented ***body*** of statements.
    - These statements are ones like the top-level ones used in scripts.
- The function's lines of code compute using the ***parameter*** variables.
- The last line executed is a `return` statement.
    - It computes a value that gets "handed" back or *returned*.
- A function can be ***called*** several times within a program’s code.
    - With each call, different values are passed to the function.

# SUMMARY

- A function’s code consists of an indented ***body*** of statements.
  - These statements are ones like the top-level ones used in scripts.
- The function's lines of code compute using the ***parameter*** variables.
- The last line executed is a `return` statement.
  - It computes a value that gets "handed" back or *returned*.
- A function can be ***called*** several times within a program’s code.
  - With each call, different values are passed to the function.
- "Procedures" are like functions, also defined using `def`.
  - They perform some work but don’t return a value.

# SUMMARY

- A function's code consists of an indented ***body*** of statements.
    - ➡ These statements are ones like the top-level ones used in scripts.
- The function's lines of code compute using the ***parameter*** variables.
- The last line executed is a `return` statement.
    - ➡It computes a value that gets "handed" back or *returned*.
- A function can be ***called*** several times within a program's code.
    - ➡With each call, different values are passed to the function.
- "Procedures" are like functions, also defined using `def`.
    - ➡They perform some work but don't return a value.
    - ➡Sometimes they do work ***and*** return a value.
    - ^^^ ***This is just my terminology, not Python's.***

# MISSING CASES?

- What happens if you (accidentally) forget a case?

```
def example(value):
    if value > 0:
        return "positive"
    elif value < 0:
        return "negative"
```

- What happens in the missing case?

```
>>> example(3)
'positive'
>>> example(-4)
'negative'
>>> example(0)
????
```

# MISSING CASES

- What happens if you (accidentally) forget a case:

```
def example(value):
    if value > 0:
        return "positive"
    elif value < 0:
        return "negative"
```

- What happens in the missing case?

```
>>> print(example(3))
positive
>>> print(example(4))
negative
>>> print(example(0))
None
```

# MISSING CASES

- What happens if you (accidentally) forget a case:

```
def example(value):
    if value > 0:
        return "positive"
    elif value < 0:
        return "negative"
```

- What happens in the missing case?

```
>>> print(repr(example(3)))
'positive'
>>> print(repr(example(4)))
'negative'
>>> print(repr(example(0)))
'None'
```

# MISSING CASES

- What happens if you (accidentally) forget a case:

```
def example(value):
    if value > 0:
        return "positive"
    elif value < 0:
        return "negative"
```

- What happens in the missing case?

```
>>> print(example(3))
positive
>>> print(example(4))
negative
>>> print(example(0))
None
```

- There is a special Python value `None` that is implicitly returned.
- Confusingly, the interpreter does not display the `None` value.

# MISSING CASES

- What happens if you (accidentally) forget a case:

```
def example(value):
    if value > 0:
        return "positive"
    elif value < 0:
        return "negative"
```

- What happens in the missing case?

```
>>> print(example(3))
positive
>>> print(example(4))
negative
>>> print(example(0))
None
```

- There is a special Python value `None` that is implicitly returned.
- *Make sure in your functions you've an explicit* `return` *for every case!*

# PROGRAMMER-DEFINED PROCEDURES

- Python has the same `def` syntax for defining ***procedures***
  - This is my term for a "function that does not return a value."
  - This is also my term for a "function that does some side work like `input` or `print`."
  - They do some stuff, perform some actions.
- For example

```
def printBoxTop(size):
    dashes = “-“ * size
    print(“+” + dashes + “+”)

def printBox(width):
    printBoxTop(width)
    print(“|” + (“ “*width) + “|”)
    printBoxTop(width)
```

- Below is its use. It's as if we've invented a `printBox` statement.

```
>>> printBox(4)
+----+
|    |
+----+
>>>
```

# EXAMPLE SCRIPT WITH PROCEDURES

```
def printBoxTop(size):
    dashes = “-“ * size
    print(“+” + dashes + “+”)

def greetTheUser(name):
    print(“Hi, “ + name + ”. Nice to meet ya!”)

def printBox(w):
    printBoxTop(w)
    print(“|” + (“ “ * w) + “|”)
    printBoxTop(w)

user = input(“What’s your name? “)
greetTheUser(user)
print(“I’d like to make you a box.”)
width = int(input(“How wide of a box would you like? “))
printBox(width)
print(“Here is one that is twice as wide:”)
printBox(width * 2)
```

# PROCEDURES RETURN THE NONE VALUE

- All three of these procedures do the exact same thing:

```
def greetThenReturn_version1(name):
    print("Hi, " + name + ".")

def greetThenReturn_version2(name):
    print("Hi, " + name + ".")
    return

def greetThenReturn_version3(name):
    print("Hi, " + name + ".")
    return None
```

- The first implicitly returns **`None`**. The first explictly returns but implictly returns **`None`**. The third explicitly returns the **`None`** value.

# NONE IS WEIRDLY HANDLED BY THE PYTHON INTERPRETER

- Here is some fun with `None`, and with procedures (that return `None`):

```
>>> print("hello")
hello
>>> print(None)
None
>>> "hello"
'hello'
>>> None
>>> 3+4
7
>>> print(print("hello"))
hello
None
>>> greetThenReturnNone("Jim")
Hello, Jim.
>>> print(greetThenReturnNone("Jim"))
Hello, Jim.
None
```

# FUNCTIONS VS. PROCEDURES

- In Python, procedures are really just functions.
  - Python doesn't distinguish procedures from functions.
  - This is just my personal dichotomy, from older languages (Pascal, C).
- "**Function**":
  - A function gets passed some parameters, executes, and then returns a result.
  - A function is ***used within an expression***.
- "**Procedure**":
  - A procedure is something that (typically) performs some action/work, and typically does not return a value.
  - A procedure that returns no value is ***used as a statement***.
  - When a procedure's work is done, Python continues executing after the line where it was called. (Control "jumps" then returns.)

# ITERATION WITH LOOPS

- We look at code that uses ***iteration*** or ***loop*** statements.
  - In Python, these are the **`while`** and **`for`** statements.
  - These statements allow us to repeat actions several times.
    - *Definite* loops: perform an action several times.
    - *Indefinite* loops: perform an action until a condition is met.
- **Reading** for this week's material:
    - TP Ch 5
    - CP Ch 1.5

# AN INFINITE LOOP

- Python lets you execute the same statement repeatedly with a `while` loop statement. For example:

```
print(“This line runs once, first.”)
while True:
    print(“This line keeps getting run.”)
print(“This line never runs.”)
```

- Output of the script above:

```
This line runs once, first.
This line keeps getting run.
This line keeps getting run.
This line keeps getting run.
This line keeps getting run.
…
```

- **NOTE:** hit [CTRL-c] to terminate the Python script's execution.

# MORE LOOPING FOREVER

- The prior example loops forever. And so does this one:

```
print(“This line runs once, first.”)
while True:
    print(“This line keeps getting run.”)
    print(“And so does this one.”)
print(“This line never runs.”)
```

- Output of the script above:

```
This line runs once, first.
This line keeps getting run.
And so does this one.
This line keeps getting run.
And so does this one.
This line keeps getting run.
And so does this one.
…
```

# LOOPING, UNROLLED

- The behavior of that script is like this infinite script:

```
print(“This line runs once, first.”)
print(“This line keeps getting run.”)
print(“And so does this one.”)
print(“This line keeps getting run.”)
print(“And so does this one.”)
print(“This line keeps getting run.”)
print(“And so does this one.”)
print(“This line keeps getting run.”)
print(“And so does this one.”)
...
```

# LOOPING, UNROLLED

- Well, technically, it's more like this infinite script:

```
print(“This line runs once, first.”)
if True:
    print(“This line keeps getting run.”)
    print(“And so does this one.”)
    if True:
        print(“This line keeps getting run.”)
        print(“And so does this one.”)
        if True:
            print(“This line keeps getting run.”)
            print(“And so does this one.”)
            if True:
                ...

print(“This line never runs.”)
```

# COUNTING FOREVER

- The prior example loops forever. And so does this one:

```
hellos_said = 0
while True:
    print(“Hello!!!”)
    hellos_said = hellos_said + 1
    print(“That was 'hello' #” + str(hellos_said) + ".")
print(“This line never runs.”)
```

- Output of the script above:

```
Hello!!!
That was 'hello' #1.
Hello!!!
That was 'hello' #2.
Hello!!!
That was 'hello' #3.
Hello!!!
That was 'hello' #4.
…
```

# COUNTING FOREVER, UNROLLED

- Well, technically, it's more like this infinite script:

```
hellos_said = 0                                   # sets to 0
if True:
    print(“Hello!!!”)
    hellos_said = hellos_said + 1                 # sets to 1
    print(“That was 'hello' #”+str(hellos_said) + ".")
    if True:
        print(“Hello!!!”)
        hellos_said = hellos_said + 1             # sets to 2
        print(“That was 'hello' #”+str(hellos_said) + ".")
        if True:
            print(“Hello!!!”)
            hellos_said = hellos_said + 1         # sets to 3
            print(“That was 'hello' #”+str(hellos_said) + ".")
            if True:
                ...
```

# COUNTING ONLY SO FAR

- This outputs a count from 0 up to 5:

```
print(“I'm going to count for you.”)
count = 0
while count < 6:
    print(count)
    count = count + 1
print(“I'm done counting now.”)
```

- Output of the script above:

```
I'm going to count for you.
0
1
2
3
4
5
I'm done counting now.
```

# COUNTING ONLY SO FAR

- This outputs a count from 0 up to ***2***:

```
print(“I'm going to count for you.”)
count = 0
while count < 3:
    print(count)
    count = count + 1
print(“I'm done counting now.”)
```

- Output of the script above:

```
I'm going to count for you.
0
1
2
I'm done counting now.
```

# UNROLLED

- Here is an unrolling of that loop's code:

```
print("I'm going to count for you.")
count = 0                             # sets to 0
if count < 3:
    print(count)
    count = count + 1                 # sets to 1
    if count < 3:
        print(count)
        count = count + 1             # sets to 2
        if count < 3:
            print(count)
            count = count + 1         # sets to 3
            if count < 3:
                print(count)          # never happens
                count = count + 1     # sets to 4; never happens
                if count < 3:
                    ...
print("I'm done counting now.")
```

# COUNTING ACCORDING TO AN INPUT

- This outputs a count from 0 up to some input value:

```
print(“I'm going to count for you.”)
max = int(input("Enter how far you'd like me to count: "))
count = 0
while count <= max:
    print(count)
    count = count + 1
print(“I'm done counting now.”)
```

- Output of the script above:

```
I'm going to count for you.
Enter how far you'd like me to count: 4
0
1
2
3
4
I'm done counting now.
```

# ANATOMY OF A WHILE LOOP

- The template below gives the syntax of a while loop statement:

```
lines of statements to execute first
while condition-expression:
    lines of statements to execute if the condition holds
    …
lines of statements to executed when the condition no longer holds
```

# EXECUTION OF A WHILE LOOP

- The template below gives the syntax of a while loop statement:

```
lines of "set up" statements to execute first
while condition-expression:
    lines of "loop body" statements to execute if the condition holds
    ...
lines of "follow up" to execute when the condition no longer holds
```

- Here is how Python executes this code:

1. Executes the **set up** code.
2. It evaluates the **condition**. If `False` it *skips* to **Step 5**.
3. Otherwise, if `True`, it evaluates the **loop body**'s code.
4. It goes back to **Step 2**.
5. It executes the **follow up**, and subsequent, code.

# ANATOMY OF A COUNTING LOOP

- Here is the standard structure of a "counting loop":

```
initialize the count to the start-value
while count < one-too-far:
    actions to perform with that particular count value
    increment the count by 1
at this point can now use the fact that count == one-too-far
```

- This is an extremely common coding pattern…
  - ➡ **PLEASE TAKE THIS TEMPLATE TO HEART!!!!**

# DEFINITE VS. INDEFINITE LOOPS

- Some terminology:
  - "*Count up to 6.*" and "*Count up to the input value.*" are examples of *definite* loops.
  - "*Get an input until they've entered something valid.*" is an example of an *indefinite* loop. The number of repetitions isn't known.

- An example of the second kind of coding:

```
def get_float(prompt):
    return float(input(prompt))

def get_area():
    a = get_float(“Circle area? “)
    while a < 0.0:
        a = get_float(“Not an area. Try again:”)
    return a
```

# DEFINITE VS. INDEFINITE LOOPS

- Some terminology:
  - "*Count up to 6.*" and "*Count up to the input value.*" are examples of *definite* loops.
  - "*Get an input until they've entered something valid.*" is an example of an *indefinite* loop. The number of repetitions isn't known.
- An example of the second kind of coding:

```
def get_float(prompt):
    return float(input(prompt))

def get_area():
    a = get_float(“Circle area? “)
    while a < 0.0:
        a = get_float(“Not an area. Try again:”)
    return a
```

*Note that the loop body might not run at all!*

# NESTING CONTROL STATEMENTS WITHIN A LOOP

- Of course you can put a conditional statement within a loop's body.

```
count = 0
while count < 6:
    if count % 2 == 0:
        print(str(count) + ” is even.”)
    else:
        print(str(count) + ” is odd.”)
    count = count + 1
print(“Done.”)
```

- Output of the script above:

```
0 is even.
1 is odd.
2 is even.
3 is odd.
4 is even.
5 is odd.
Done.
```

# NESTING A LOOP WITHIN A LOOP

- Nested loops are a common programming pattern:

```
a = 0
while a < 6:
    b = 0
    while b < 8:
        print(str(a)+str(b),end=“ ”)
        b = b + 1
    print()
    a = a + 1
print(“Done.”)
```

- ***What does this do???***

```
0 is even.
1 is odd.
2 is even.
3 is odd.
4 is even.
5 is odd.
Done.
```

# NESTING A LOOP WITHIN A LOOP

- Nested loops are a common programming pattern:

```
a = 0
while a < 6:
    b = 0
    while b < 8:
        print(str(a)+str(b),end=“ ”)
        b = b + 1
    print()
    a = a + 1
print(“Done.”)
```

- ***It outputs a sequence of digit pairs, separated by spaces:***

```
00 01 02 03 04 05 06 07
10 11 12 13 14 15 16 17
20 21 22 23 24 25 26 27
30 31 32 33 34 35 36 37
40 41 42 43 44 45 46 47
50 51 52 53 54 55 56 57
Done.
```

# NESTING A LOOP WITHIN A LOOP

- Nested loops are a common programming pattern:

```
a = 0
while a < 6:
    b = 0
    while b < 8:
        print(str(a)+str(b),end=“ ”)
        b = b + 1
    print()
    a = a + 1
print(“Done.”)
```

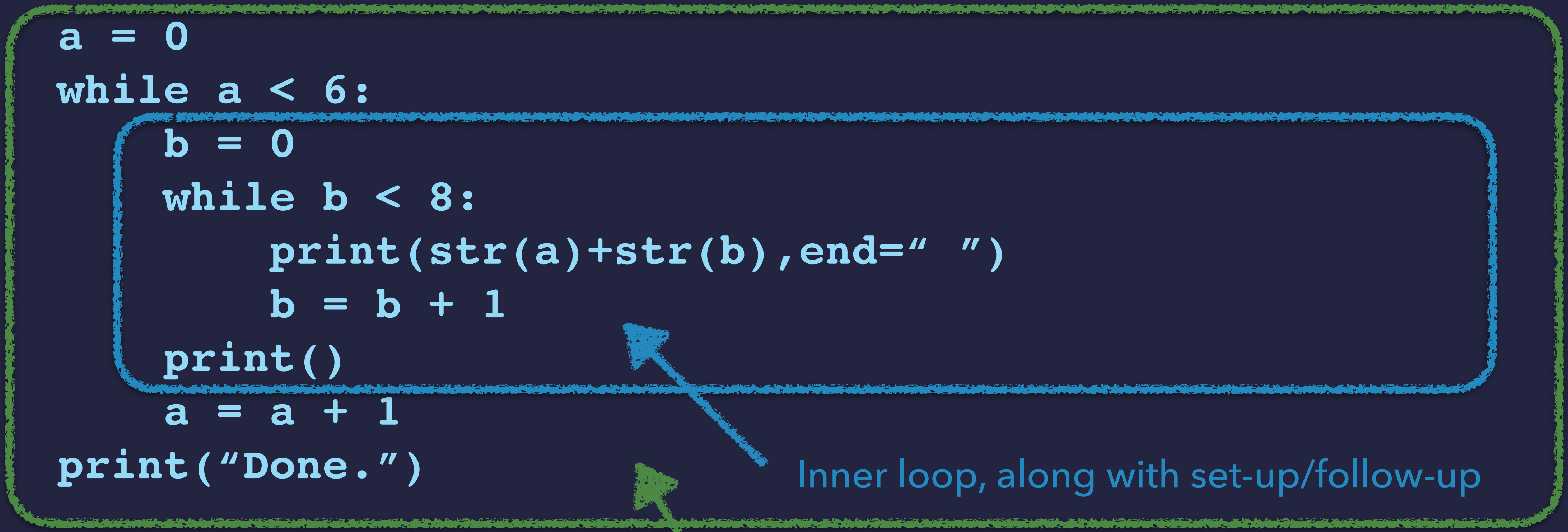
Inner loop, along with set-up/follow-up

- ***It outputs a sequence of digit pairs, separated by spaces:***

```
00 01 02 03 04 05 06 07
10 11 12 13 14 15 16 17
20 21 22 23 24 25 26 27
30 31 32 33 34 35 36 37
40 41 42 43 44 45 46 47
50 51 52 53 54 55 56 57
Done.
```

# NESTING A LOOP WITHIN A LOOP

- Nested loops are a common programming pattern:



```
a = 0
while a < 6:
    b = 0
    while b < 8:
        print(str(a)+str(b),end=“ ”)
        b = b + 1
    print()
    a = a + 1
print(“Done.”)
```

Inner loop, along with set-up/follow-up

- ***It outputs a sequence of digit pairs, separated by spaces:***

```
00 01 02 03 04 05 06 07
10 11 12 13 14 15 16 17
20 21 22 23 24 25 26 27
30 31 32 33 34 35 36 37
40 41 42 43 44 45 46 47
50 51 52 53 54 55 56 57
Done.
```

Outer loop, along with set-up/follow-up

# NESTING A LOOP WITHIN A LOOP

- Nested loops are a common programming pattern:

```
a = 0
while a < 6:
    b = 0
    while b < 8:
        print(str(a)+str(b),end=‘ ’)
        b = b + 1
    print()
    a = a + 1
print(“Done.”)
```

Executed once for each value of a.

Inner loop, along with set-up/follow-up

- *It outputs a sequence of digit pairs, separated by spaces:*

```
00 01 02 03 04 05 06 07
10 11 12 13 14 15 16 17
20 21 22 23 24 25 26 27
30 31 32 33 34 35 36 37
40 41 42 43 44 45 46 47
50 51 52 53 54 55 56 57
Done.
```

Outer loop, along with set-up/follow-up

# BREAKING OUT OF A LOOP

- Here is another way of writing the counting loop.

```
print(“Counting from 0 to 5:”)
count = 0
while True:
    if count >= 6:
        break
    print(count)
    count = count + 1
print(“Done.”)
```

- The code uses a **`break`** statement to jump down to the follow-up code.
- If within several loops, it jumps to just after the innermost one.
- This is an artificial example
- Using **`break`** statements can sometimes make code more readable than code that expresses all the "break out" or stopping conditions.

# USING CONDITION VARIABLES TO GOVERN LOOPING

- Using **`break`** to express other break-out conditions:

```
while count < 6:
    if somethingElseMakesMeStop(...)
        break
    ...
    count = count + 1
print(“Done.”)
```

- I worry that **`break`** can sometimes be missed by other coders.
- I usually prefer using explicit break-out conditions instead, like so:

```
done = False
while !done and count < 6:
    if somethingElseMakesMeStop(...)
        done = True
    if not done:
        ...
        count = count + 1
print(“Done.”)
```

# USING CONDITION VARIABLES TO GOVERN LOOPING

- Using `break` to express other break-out conditions:

```
while count < 6:
    if somethingElseMakesMeStop(...)
        break
    ...
    count = count + 1
print(“Done.”)
```

***PLEASE use `break` sparingly, and with taste.***

- I worry that `break` can sometimes be missed by other coders.
- I usually prefer using explicit break-out conditions instead, like so:

```
done = False
while !done and count < 6:
    if somethingElseMakesMeStop(...)
        done = True
    if not done:
        ...
        count = count + 1
print(“Done.”)
```

# USING RETURN WITHIN A LOOP

- This procedure uses **`return`** to exit its loop and the procedure:

```
def countUpTo(n)
    count = 1
    while True:
        if count > n:
            return
        print(count)
        count = count + 1
```

- The **`return`** statement breaks out of the loop and returns back to the place where **`countUpTo`** was called.

# SUMMARY

- The while loop statement expresses **iterative** code.
  - Allows you to perform a series of actions *until* a condition holds.
  - The negation of this *terminating condition* is the loop's condition.
- It's possible for the code to loop forever. This is an ***infinite*** loop.
- Counting loops are common examples of ***definite*** loops.
- Loops that iterate an undetermined number of times are ***indefinite***.

# SUMMARY (CONT'D)

- Loop bodies can contain other control statements:
  - For example, you can have `if` statements or other `while` statements.
  - If another loop statement is inside, then it is a ***nested loop***.
  - If a `break` statement, we can jump out of the loop mid-body.
  - If a `return` statement, we exit the loop *and* the function/procedure.