CS101 - Unit 2: How to Repeat, Finding All the Links on a Page

Contents

Introduction

In Unit 1, you wrote a program to extract the first link from a web page. The next step towards building your search engine is to extract all of the links from a web page. In order to write a program to extract all of the links, you need to know these two key concepts:

1. **Procedures** - a way to package code so it can be reused with different inputs.
2. **Control** - a way to have the computer execute different instructions depending on the data (instead of just executing instructions one after the other).

Recall this code from then end of Unit 1:

```
page = ...contents of some web page...
start_link = page.find('<a href=')
start_quote = page.find('"', start_link)
end_quote = page.find('"', start_quote + 1)
```

```
url = page[start_quote + 1:end_quote]
print url
```

This finds and prints the first link on the page. To keep going, we could update the value of page to be the characters from the **end_quote**, and repeat the same code again:

```
page = page[end_quote:]
start_link = page.find('<a href=')
start_quote = page.find('"', start_link)
end_quote = page.find('"', start_quote + 1)
url = page[start_quote + 1:end_quote]
print url
```

```
page = page[end_quote:]
start_link = page.find('<a href=')
start_quote = page.find('"', start_link)
end_quote = page.find('"', start_quote + 1)
url = page[start_quote + 1:end_quote]
print url
```

This code will print out the next two links on the web page. Clearly, this is tedious work. The reason for computers is to avoid having to do tedious, mechanical work! In addition to being tedious, repeating the same code over and over again like this will not work well because some pages only have a few links while other pages have more links than the number of repetitions.

In this unit, you will learn three important programming constructs: procedures, if statements, and while loops. Procedures, also known in Python as "functions," enable you to abstract code from its inputs; **if** statements allow you to write code that executes differently depending on the data; and **while** loops provide a convenient way to repeat the same operations many times. You will combine these procedures to solve the problem of finding all of the links on a web page.

Motivating Procedures

Procedural abstraction is a way to write code once that works on any number of different data values. By turning our code into a procedure, we can use that code over and over again with different inputs to get different behaviors.

Introducing Procedures

A procedure takes in inputs, does some processing, and produces outputs.

For example, the + operator is a procedure where the inputs are two numbers and the output is the sum of those two numbers. The + operator looks a little different from the procedures we will define since it is built-in to Python with a special

operator syntax. In this unit you will learn how to write and use your own procedures.

Here is the Python grammar for writing a procedure:

```
def <name>(<parameters>):
    <block>
```

The keyword **def** is short for "define".

<name> is the name of a procedure. Just like the name of a variable, it can be any string that starts with a letter and followed by letters, number and underscores.

<parameters> are the inputs to the procedure. A parameter is a list of zero or more names separated by commas: ***<name>***, ***<name>***,... Remember that when you name your parameters, it is more beneficial to use descriptive names that remind you of what they mean. Procedures can have any number of inputs. If there are no inputs, the parameter list is an empty set of closed parentheses: **( )**.

After the parameter list, there is a : (colon) to end the definition header.

The body of the procedure is a ***<block>***, which is the code that implements the procedure. The block is indented inside the definition. Proper indentation tells the interpreter when it has reached the end of the procedure definition.

Consider how to turn the code for finding the first link into a **get_next_target** procedure that finds the next link target in the page contents. Here is the original code:

```
start_link = page.find('<a href=')
start_quote = page.find('"', start_link)
end_quote = page.find('"', start_quote + 1)
url = page[start_quote + 1:end_quote]
```

Next, to make this a procedure, we need to determine what the inputs and outputs are.

Quiz 1: Procedure Code

What are the inputs for the procedure, **get_next_target**?

1. a number giving position of start of link
2. a number giving position of start of next quote
3. a string giving contents of the rest of the web page
4. a number giving position of start of link and a string giving page contents

Answer

Quiz 2: Output

What should the outputs be for **get_next_target**?

1. a string giving the value of the next target url (url)
2. url, page
3. url, end_quote
4. url, start_link

Answer

Return Statement

To make the **get_next_target** procedure, we first add a procedure header and indent our existing code in a block:

```
def get_next_target(page):
    start_link = page.find('<a href=')
    start_quote = page.find('"', start_link)
    end_quote = page.find('"', start_quote + 1)
    url = page[start_quote + 1:end_quote]
```

You can change the name of the parameter **page** to **s**. This is more descriptive since the procedure can work on any string. After we rename the parameter, we also need to change the name wherever it is used in the block:

```
def get_next_target(s):
    start_link = s.find('<a href=')
    start_quote = s.find('"', start_link)
    end_quote = s.find('"', start_quote + 1)
    url = s[start_quote + 1:end_quote]
```

To finish the procedure, we need to produce the outputs. To do this, introduce a new Python statement called return. The syntax for return is:

- **return <*expression*>, <*expression*>, ...**

A ***return*** statement can have any number of expressions. The values of these expressions are the outputs of the procedure.

A ***return*** statement can also have no expressions at all, which means the procedure produces no output. This may seem silly, but in fact it is quite useful. Often, you want to define procedures for their side-effects, not just for their outputs. Side-effects are visible, such as the printing done by a ***print*** statement, but are not the outputs of the procedure.

Quiz 3: Return Statement

Complete the **get_next_target** procedure by filling in the **return** statement that produces the desired outputs.

```
def get_next_target(s):
    start_link = s.find('<a href=')
    start_quote = s.find('"', start_link)
    end_quote = s.find('"', start_quote + 1)
    url = s[start_quote + 1:end_quote]
    return ________
```

Answer

Using Procedures

In order to use a procedure, you need the name of the procedure, followed by a left parenthesis, a list of the procedure's inputs (sometimes called operands or arguments), closed by right parenthesis:

- ***<procedure>(<input>,<input>, ...)***

For example, consider the **rest_of_string** procedure defined as:

```
def rest_of_string(s):
    return s[1:]
```

To use this procedure we need to pass in one input, corresponding to the parameter **s**:

```
print rest_of_string('audacity')
    udacity
```

You can see what is going on by adding a **print** statement in the procedure body:

```
def rest_of_string(s):
    print 'Here I am in rest_of_string!'
    return s[1:]

    print rest_of_string('audacity')
    Here I am in rest_of_string!
    udacity
```

You can do anything you want with the result of a procedure, for example you can store it in a variable.

```
def rest_of_string(s):
    print 'Here I am in rest_of_string!'
    return s[1:]

s = rest_of_string('audacity')
```

```
print s
Here I am in rest_of_string!
udacity
```

However, see what happens here:

```
t = rest_of_string('audacity')
print s
Here I am in rest_of_string!

Traceback (most recent all last):
File "/code/knowvm/input/test.py", line 7, in <module>
print s
NameError: name 's' is not defined
```

An error is returned because the variable **s** is not defined outside the block of the procedure.

Think of procedures as mapping inputs to outputs. This is similar to a mathematical *function*. Indeed, many people call procedures in Python like the ones we are defining "functions." Refer to them as *procedures* because they are quite different from mathematical functions. The main differences are:

- A mathematical function always produces the same output given the same inputs. This is not necessarily the case for a Python procedure, which can produce different outputs for the same inputs depending on other state (we will see examples in Unit 3).
- A mathematical function is a pure abstraction that has no associated cost. The cost of executing a Python procedure depends on how it is implemented. (We will discuss how computer scientists think about the cost of procedures in Unit 5.)
- A mathematical function only maps inputs to outputs. A Python procedure can also produce side-effects, like printing.

Procedures are a very important concept and the core of programming is breaking problems into procedures, and implementing those procedures.

Quiz 4: Inc Procedure

What does the **inc** procedure defined below do?

```
def inc(n):
    return n + 1
```

1. Nothing.
2. Takes a number as input, and outputs that number plus one
3. Takes a number as input, and outputs the same number
4. Takes two numbers as inputs, and outputs their sum

Answer

Quiz 5: Sum Procedure

What does the sum procedure defined below do?

```
def sum(a, b):
    a = a + b
```

1. Nothing
2. Takes two numbers as its inputs, and outputs their sum
3. Takes two strings as its inputs, and outputs the concatenation of the two strings
4. Takes two numbers as its inputs, and changes the value of the first input to be the sum of the two number

Answer

Quiz 6: Sum Procedure with a Return Statement

What does the **sum** procedure defined below do?

```
def sum(a,b):
    a = a + b
    return a
```

1. Takes two numbers as its inputs, and outputs their sum.
2. Takes two strings as its inputs, and outputs the concatenation of the two strings.
3. Takes two numbers as its inputs, and changes the value of the first input to be the sum of the two number.

Answer

Quiz 7: Square

Define a procedure, **square**, that takes one number as its input, and outputs the square of that number (result of multiplying the number by itself).

For example:

```
print square(5)
25
```

Answer

Quiz 8: Sum of Three

Define a procedure, **sum3**, that takes three inputs, and outputs the sum of the three input numbers.

```
print sum3(2, 2, 3) â†’ 7
```

Answer

Quiz 9: Abbaize

Define a procedure, **abbaize**, that takes two strings as its input, and outputs a string that is the first input followed by two repetitions of the second input, followed by the first input.

```
abbaize('a', 'b') â†’ 'abba'
abbaize('dog', 'cat') â†’ 'dogcatcatdog'
```

Answer

Quiz 10: Find Second

Define a procedure, **find_second**, that takes two strings as its inputs: a search string and a target string. It should output a number located at the second occurrence of the target string within the search string.

Example:

```
danton = "De l'audace, encore de l'audace, toujours de l'audace."
print find_second(danton, 'audace')
25
```

Answer

Equality Comparisons

Everything so far has been limited. So far, your programs are only able to do the same thing on all data and you cannot do anything that depends on what the data is. Now, let's figure out a way to make code behave differently based on decisions. To do so we want to find a way to make comparisons, which will give you a way to test and ultimately allow your program to decide what to do.

Comparison Operators

Python provides several operators for making comparisons:

- < less than
- > greater than
- <= less than or equal to

- >= greater than or equal to
- == equal to
- != not equal to

All of these operators act on numbers, for example:

- ***<number> <operator> <number>***

The output of a comparison is a **Boolean**: *True* or *False*.

Here are some examples:

```
print 2 < 3
True

print 21 < 3
False
```

You can also make comparisons using expressions:

```
print 7 * 3 < 21
False

print 7 * 3 != 21
False
```

Note the equality is done using two equals signs (double ==), not a single =:

```
print 7 * 3 == 21
True
```

Quiz 11: Equality Comparisons

Why is the equality comparison done using == instead of =?

1. Because = means approximately equal.
2. Because not equal uses two characters !=.
3. Because Guido (van Rossum, the creator of Python) really likes = signs.
4. Because = means assignment.
5. It doesn't matter, we can use either == or =.

Answer

If Statements

An **if** statement provides a way to control what code executes based on the result of a test expression. Here is the grammar of the **if** statement:

```
if <TestExpression>:
      <block>
```

In this statement, the code in the <*block*> runs only if the value of the test expression is *True*. Similar to procedures, the end of the **if** statement block is determined by the indentation.

Here is an example where we use an **if** statement to define a procedure that returns the absolute value of its input:

```
def absolute(x):
    if x < 0:
       x = -x
    return x
```

Quiz 12: If Statements

Define a procedure, **bigger**, that takes in two numbers as inputs, and outputs the greater of the two inputs.

- **bigger(2, 7)** â†' 7
- **bigger(3, 2)** â†' 3
- **bigger(3, 3)** â†' 3

Answer

Is Friend: Else Expressions

You can use an **else** clause in addition to an **if** statement to provide alternative code that will execute when the test expression is false. The test expression determines whether to execute the block inside the **if** statement or the block inside the **else** statement:

```
if <''TestExpression''>:
   <''block''>
else:
   <''block''>
```

Using **else**, you can define **bigger** in a more symmetrical way:

```
def bigger(a, b):
    if a > b:
       return a
     else:
        return b
```

Here is another way to write this:

```
def bigger(a, b):
    if a > b:
       r = a
    else:
        r = b
    return r
```

Quiz 13: Is Friend

Define a procedure, **is_friend**, that takes a string as its input, and outputs a Boolean indicating if the input string is the name of a friend. Assume I am friends with everyone whose name starts with **D** and no one else.

```
print is_friend('Diane')    â†’     True
print is_friend('Fred')     â†’     False
```

Answer

Quiz 14: More Friends

Define a procedure, **is_friend**, that takes a string as its input, and outputs a Boolean indicating if the input string is the name of a friend. Assume I am friends with everyone whose name starts with either **D** or **N**, but no one else.

```
print is_friend('Diane')    â†’ True
print is_friend('Ned')      â†’ True
```

Answer

Or Expressions

An **or** expression gives the logical or (disjunction) of two operands. If the first expression evaluates to *True*, the value is *True* and the second expression is not evaluated. If the value of the first expression evaluates to *False* then the value of the **or** is the value of the second expression.

- **<*Expression*> or <*Expression*>**

Here are a few examples:

```
print True or False
True
print False or True
True
print True or True
True
print False or False
False
```

An important difference between an **or** expression and other operators is that an **or** expression does not necessarily evaluate both of its operand expressions. For example:

```
print True or this_is_an_error
True
```

Even though **this_is_an_error** would produce an error because the variable is not defined, the **or** expression does not produce an error! This is because the second operand expression of an **or** is only evaluated if the first expression evaluates to *False*. When the first operand expression evaluates to *True*, the output of the **or** expression must be *True* regardless of the value of the second operand expression. The Python rules of evaluation require that the second operand expression not be evaluated in cases where the value of the first operand is *True*.

Quiz 15: Biggest

Define a procedure, biggest, that takes three numbers as inputs, and outputs the greatest of the three numbers.

```
biggest (6, 2, 3)  â†’   6
biggest (6, 2, 7)  â†’   7
biggest (6, 9, 3)  â†’   9
```

Answer

```
def biggest(a, b, c):
    if a > b:
        if a > c:
            return a
        else:           # c >= a > b
            return c
    else:               # b >= a
        if b > c:
            return b
        else:           # c >= b >= a
            return c
```

We can reduce the number of if statements, and write this in an easier way by taking advantage of the bigger() procedure we defined earlier.

```
def bigger(a, b):
    if a > b:
        return a
    else:
        return b

def biggest(a, b, c):
    return bigger(bigger(a, b), c)
```

In fact, there is a built-in procedure to find the greatest number. That procedure is max()

```
max(a, b, c)
```

With a very few simple operations, you can simulate any other operation you want.

You've seen:

```
Arithmetic
Comparisons
Procedures (how to define and call them)
if
```

You know enough (in theory) to write every possible computer program!

This remarkable claim was proven by Alan Turing in the 1930s.

Alan Turing

This biographical sketch is adapted from David Evans, Introduction to Computing: Explorations in Language, Logic, and Machines, available free from http://www.computingbook.org.

Alan Turing was born in London in 1912, and developed his computing model while at Cambridge in the 1930s. He developed the model to solve a famous problem posed by David Hilbert in 1928.

The problem, known as the *Entscheidungsproblem* (German for "decision problem") asked for an algorithm that could determine the truth or falsehood of a mathematical statement. To solve the problem, Turing first needed a formal model of an algorithm. For this, he invented the machine model that is now known as a Turing machine and defined an algorithm as any Turing Machine that is guaranteed to eventually halt on any input.

With the model, Turing was able to show that any machine could be simulated by a universal machine with a few very simple operations. The operations he used were slightly different from what we have covered, but can be viewed as equivalent to being able to define and use procedures, being able to make decisions using **if**, and being able to do simple arithmetic and comparisons. Turing also proved that there are some problems that cannot be solved by any algorithm. We will not prove that in this course, but will soon see the most famous example: it is impossible for a program to determine in general if another program will run forever or eventually finish executing.

After publishing his solution to the *Entscheidungsproblem* in 1936, Turing went to Princeton and studied with Alonzo Church (inventor of the Lambda calculus, the basis of the programming language LISP, which is a major influence on Python).

With the start of World War II, Turing joined the highly secret British effort to break Nazi codes at Bletchley Park. Turing was instrumental in breaking the Enigma code which was used by the Nazis to communicate with field units and submarines. Turing designed an electro-mechanical machine known as a *bombe* for efficiently searching possible keys to decrypt Enigma-encrypted messages.

The machines used logical operations to search the possible rotor settings on the Enigma to find the settings that were most likely to have generated an intercepted encrypted message. Bletchley Park was able to break thousands of Enigma messages during the war. The Allies used the knowledge gained from them to avoid Nazi submarines and gain a tremendous tactical advantage.

After the war, Turing continued to make both practical and theoretical contributions to computer science. Among other things, he worked on designing general-purpose computing machines and published a paper speculating on the ability of computers to exhibit intelligence. Turing introduced a test for machine intelligence (now known as the *Turing Test* and the inspiration behind the annoying "CAPTCHA" images that challenge you to prove you are a human before submitting a web form) based on a machines ability to impersonate a human and speculated that machines would be able to pass the test within 50 years (that is, by the year 2000). Turing also studied morphogenesis (how biological systems grow) including why Fibonacci numbers (to come in Unit 6) appear so often in plants.

In 1952, Turing's house was broken into, and Turing reported the crime to the police. The investigation revealed that Turing was a homosexual, which at the time was considered a crime in Britain. Turing did not attempt to hide his homosexuality. He was convicted and given a choice between serving time in prison and taking hormone treatments. He accepted the treatments, and his security clearance was revoked. In 1954, at the age of 41, Turing was found dead in an apparent suicide, with a cyanide-laced partially-eaten apple next to him.

The code-breaking effort at Bletchley Park was kept secret for many years after the war (Turing's report on Enigma was not declassified until 1996), so Turing never received public recognition for his contributions to the war effort. In September 2009, instigated by an on-line petition, British Prime Minister Gordon Brown issued an apology for how the British government treated Alan Turing.

While Loops

Loops are a way of executing something over and over.

The syntax for the **while** loop is very similar to the **if** statement:

```
while <TestExpression>:
      <Block>
```

In contrast to an **if** statement where the block executes either 0 or 1 times

depending on whether the test expression is **True** or **False**, a **while** loop executes any number of times, continuing as long as the test expression is **True**.

In a **if** statement if the test expression is *True*, the block executes once and then continues to the following statement. If the test expression is *False*, execution jumps over the block and continues with the following statement.

If the test expression for a **while** loop is *True*, the block is executed. Then, execution continues by going back to the test expression again. If it still evaluates to *True*, the block is executed again, and execution continues by going back to the test expression one more time. This continues as long as the test expression evaluates to *True*. and again as many times as you need. Once the test expression is *False*, execution jumps to the next statement. There is no guarantee that the test expression eventually becomes *False*, however. The loop could keep running forever, which is known as an **infinite loop**.

Here is an example **while** loop:

```
i = 0
while i < 10:
      print i
      i = i + 1
```

This code will execute the inner block as long as the test expression, **while i < 10**, is *True*. Inside the block, it prints **i** and then adds 1 to **i**. In this example, the while loop repeats 10 times, printing the numbers from 0 to 9. At the end of the loop, the value of **i** is 10.

Quiz 16: While Loops

What does this program do?

```
i = 0
while i != 10:
      i = i + 1
      print i
```

1. Produce an error
2. Print out the numbers from 0 to 9.
3. Print out the numbers from 1 to 9.
4. Print out the numbers from 1 to 10.
5. Run

Answer

Quiz 17: While Loops-2

What does the following code do?

```
i = 1
while i != 10:
      i = i + 2
      print i
```

1. Produce an error.
2. Print out 2, 4, 6, 8.
3. Print out 1, 3, 5, 7, 9.
4. Print out 3, 5, 7, 9.
5. Run forever.

Answer

Quiz 18: Print Numbers

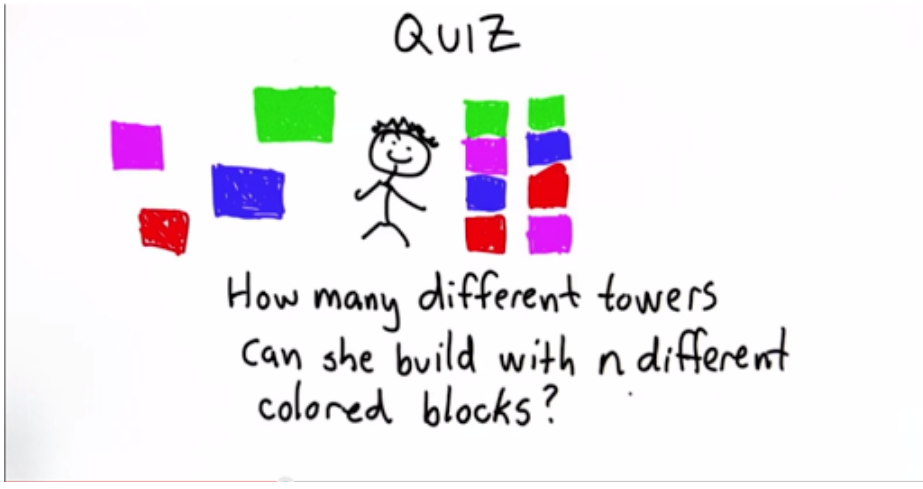
Define a procedure, **print_numbers**, that takes as input a positive whole number, and prints out all the whole numbers from 1 up to and including the input number.

```
print_numbers(3)
1
2
3
```

Answer

Factorial

Suppose you have four blocks and a baby. You want to know how long the baby can play with the blocks without getting bored. The baby wants to try all of the different ways of arranging the blocks by stacking them in a tower.



Think about the baby's choices for the first block, she has four. Say she reaches for

the red block first. When she reaches for the second block she is down to just three choices. If she stacks the green block on top of the red block, she has two choices left, the blue block and the purple block. Next, the baby picks the purple one. Therefore, for her fourth block, the baby only has one choice, the blue one.

To figure out the total number of choices you want to multiply the number of choices for the first block by the number of choices for the second block, by the number of choices for the third block, all the way to the fourth block. The function that you are computing is factorial. For any input n, you compute the factorial, which is the number of ways of arranging *n* items.

- **factorial**(*n*) = *n* * (*n*-1) * (*n*-2) * ... * 2 * 1

Quiz 19: Factorial

Define a procedure, factorial, that takes one number as its input and outputs the factorial of that number.

Answer

Break

Break gives us a way to break out of a loop, even if the test condition is true. The typical structure of the loop with a break looks like this:

```
while <TestExpression>:
      <Code>
      if <BreakTest>:
         break # stop executing the while loop
      <More Code>
<After While>
```

The break statement jumps out of the loop to <*After While*>.

Here is an example showing how we could rewrite **print_numbers** using **break**:

```
def print_numbers(n):
    i = 1
    while True:
          if i > n:
             break
          print i
          i = i + 1
```

This has the same behavior as the previous implementation, except now the test condition for the **while** loop is **True**. This is a bad example: if there is a way to write a loop without using break, it is usually best to avoid it. We will see soon an example where it is more difficult to write the loop without using break, since it is

not clear before executing part of the block if the loop should continue repeating.

Quiz 20: Break

Which of the following are always equivalent to:

```
while <T>:
    <S>
```

1.

```
while <T>:
    if False:
        break
    <S>
```

2.

```
while <T>:
    <S>
    break
```

3.

```
while True:
    if <T>:
        break
    <S>
```

4.

```
while <T>:
    <S>
    if <T>:
        <S>
    else:
        break
```

Answer

Multiple Assignment

So far we have defined a procedure to eliminate writing tedious code:

```
def get_next_target(page):
    start_link = page.find('<a href=')
    start_quote = page.find('"', start_link)
    end_quote = page.find('"', start_quote + 1)
    url = page[start_quote + 1:end_quote]
```

```
    return url, end_quote
```

Although you have not yet used a procedure that returns two things, it is pretty simple to do. You can do this by having two values on the left side of an assignment statement.

Assigning multiple values on the left side of an assignment statement is called **multiple assignment**. To write a multiple assignment you can put any number of names separated by commas, an equal sign and then any number of expressions separated by commas. The number of names and the number of expressions has to match so that the value of the first expression can be assigned to the first name and so on.

- <*name1*>, <*name2*>, ... = <*expression1*>, <*expression2*>, ...

Example:

```
a, b = 1, 2
```

Therefore, in order to get the two values, (url, end_quote) to return from the procedure above, you will need to have the two variables on the left side and the procedure on the right. Here is the syntax to do that:

```
url, endpos = get_next_target(page)
```

Quiz 21: Multiple Assignments

What does this do?

```
s, t = t, s
```

1. Nothing
2. Makes s and t both refer to the original value of t
3. Swaps the values of s and t
4. Error

Answer

No Links

There is one concern with the **get_next_target** procedure that has to be fixed before tackling the problem of outputting all of the links, which is:

What should happen if the input does not have another link?

Test the code in the interpreter using a test link to print **get_next_target**:

```
def get_next_target(page):
```

```
    start_link = page.find('<a href=')
    start_quote = page.find('"', start_link)
    end_quote = page.find('"', start_quote + 1)
    url = page[start_quote + 1:end_quote]
    return url, end_quote

print get_next_target('this is a <a href="http://udacity.com">link!</a>')
    ('http://udacity.com', 37)
```

When you run this code you get both outputs as a tuple, that is, the link followed by the position of the end quote. A tuple is an immutable list, meaning it cannot be modified. In the same way a list is enumerated with brackets, a tuple definition is bounded by parenthesises.

Or you can write this using a double assignment to return just the url:

```
def get_next_target(page):
    start_link = page.find('<a href=')
    start_quote = page.find('"', start_link)
    end_quote = page.find('"', start_quote + 1)
    url = page[start_quote + 1:end_quote]
    return url, end_quote

url, endpos = get_next_target('this is a <a href="http://udacity.com">link!<

print url
http://udacity.com
```

What happens if we pass in a page that doesn't have a link at all?

```
def get_next_target(page):
    start_link = page.find('<a href=')
    start_quote = page.find('"', start_link)
    end_quote = page.find('"', start_quote + 1)
    url = page[start_quote + 1:end_quote]
    return url, end_quote

url, endpos = get_next_target('good')

print url
goo
```

The program returns, "goo" because when the find operation does not find what it is looking for it returns -1. When -1 is used as an index, it eliminates the last character of the string.

NOTE: using -1 for starting position in .find procedure makes the procedure to start searching at the end of string. (-1 = last index- on string)

Compare with a string that includes double quotes:

```
def get_next_target(page):
start_link = page.find('<a href=')
        start_quote = page.find('"', start_link)
        end_quote = page.find('"', start_quote + 1)
        url = page[start_quote + 1:end_quote]
        return url, end_quote

url, endpos = get_next_target('Not "good" at all!')

print url
Not
```

In the end, the code above is not very useful. It is going to be very hard to tell when you get to the last target because maybe Not could be a valid url, but we don't know that.

Quiz 22: No Links

Think about making **get_next_target** more useful in the case where the input does not contain any link. This is something you can do! Here is a hint:

```
def get_next_target(page):
    start_link = page.find('<a href=') # HINT: put something into the code h

    start_quote = page.find('"', start_link)
    end_quote = page.find('"', start_quote + 1)
    url = page[start_quote + 1:end_quote]
    return url, end_quote

url, endpos = get_next_target('Not "good" at all!')
print url
```

Modify the **get_next_target** procedure so that if there is a link it behaves as before, but if there is no link tag in the input string, it outputs *None, 0*.

Answer

Print All Links

At this point you have a piece of code, **get_next_target**, to replace a formerly tedious program. Here is where you are so far, with a few modifications:

```
page = contents of some web page as a string
url, end pos = get_next_target(page)
print url
page = page[endpos:] #replaced end_quote with endpos
```

```
url, endpos = get_next_target(page)
...
```

This code will have to repeat and keep going until the url that's returned is None.<

So far, you have seen a way to keep going, which is a while loop, you have seen a way to test the url, and now you have everything you need to print all the links on the page!

Quiz 23: Print All Links

In the following code, fill in the test condition for the while loop and the rest of the else statement:

```
def print_all_links(page):
    while ________: # what goes as the test condition for the while?
    url, endpos = get_next_target(page)
         if url: # test whether the url you got back is None
         print url
         page = page[endpos:] # advance page to next position
         else: # if there was no valid url, then '''get_next_target''' did r
```

Answer

Go back to the xkcd web page we looked at earlier and try something a little more interesting.

Go to the xkcd.com home page, click view source to find the first link on the page and notice how many links there are in the source code â€" quite a few.

Using your **print_all_links** code, print get_page and try passing in the page url, 'http://xkcd.com/353'. Your program should return the page's source code when you run it.

```
def print_all_links(page):
    while True:
          url, endpos = get_next_target(page)
          if url:
                print url
                page = page[endpos:]
          else:
                break

    print get_page('http://xkcd.com/353')
```

Since we do not need the entire source code for what we are looking for, try your `print_all_links` procedure to print all of the links on the xkcd.com page to print the links on the page.

- def print_all_links(page):
    - while True:
        - url, endpos = get_next_target(page) if url:
            - print url
                - page = page[endpos:]

            else:

                - break

print_all_links(get_page('http://xkcd.com/353'))

}}} There are a few links that are not returned, but you will learn about those situations in the coming units.

Congratulations! You just learned how to print all the links on a web page, every possible computer program, and are in good shape to continue building your web browser! In the next unit you will learn how to collect the links from a web page and do something with them.

Answer Key

Quiz 1: Answer

- c. a string giving contents of the rest of the web page.

One way to see this is to look at the code you are trying to replace, and identify the values that must be known before running the code. In this case, the value of page must be known before this code is executed since it is used on the right side of an assignment statement before it is defined.

Quiz 2: Answer

To determine the outputs, we need to think about what is needed after the procedure. Anything computed by the procedure that we want to use after the procedure finishes must be an output.

To answer this question, look at the code after the procedure.

```
print url
page = page[end_quote:]
```

Since we already know the value of **page**, as indicated by the fact that it is known before the procedure was called, then the best answer is c. The reason we want

end_quote as an output is because knowing where the end of the quote is allows us to advance the page so that the next time we look for a link target, we won't find the same one. Instead, we assign a new value to **page** that is made up of the subsequent characters in the current value of **page** starting from the **end_quote** to skip over the link we just found.

Quiz 3: Answer

We are looking to return two things, the value of **url** and the value of **end_quote**. Do this by just returning those two values:

```
    return url, end_quote
```

In this example, the input to the procedure is a single string, and its outputs are a string (**url**) and a number (**end_quote**). The inputs and outputs of procedures can be anything you want, and nearly all the work in computing is done by passing inputs to procedures, and then using their outputs as the inputs to other procedures.

For example, procedures in a self-driving cars use data sensed by laser range finders, cameras, and pressure sensors as inputs, and produce outputs that control the steering and brakes on the car.

Quiz 4: Answer

- b. there is one input with an output of the input value plus one

Quiz 5: Answer

Nothing!

Let's look at this in the Python interpreter:

```
def sum(a,b):
    a = a + b

print sum(1, 1)
None
```

The reason the result is **None** is because the **sum** procedure does not return anything. The value **None** is a special value that means that a variable has no value.

To produce a result, we need to add a **return** statement to the **sum** procedure:

```
def sum(a,b):
    a = a + b
```

```
    return a
```

Now, when we use sum it returns the **sum** of its two input numbers:

```
print sum(2, 123)
125
```

Note that even if you pass in variables as the inputs, the values those variables refer to do not change. For example:

```
def sum(a,b):
    a = a + b

    a = 2
    b = 123
      sum(a, b)

print a
2
```

Even though the value of the parameter **a** is changed inside the body of the procedure, the name **a** inside the procedure is different from the name **a** outside the procedure.

Quiz 6: Answer

- b. & c. because the plus operator works on both strings and numbers

Quiz 7: Answer

```
def square(n):
    return (n*n)
```

Here are some examples using **square**: #!highlight python x = 37 print square(x) 1369 y = square(x) print square (y) 1874161 } The last example is the same as:

```
x = 37
print square(square(x))
1874161
```

This is an example of **procedure composition**. We compose procedures by using the outputs of one procedure as the inputs of the next procedure. In this case, we use the output of **square(x)** as the next input to **square**.

Connecting procedures using composition is a very powerful idea. Most of the work of programs is done by composing procedures.

Quiz 8: Answer

```
def sum3 (a, b, c):
    return a + b + c
```

Quiz 9: Answer

```
def abbaize(a, b)
    return a + b + b + a

print abbaize('dog', 'cat')
dogcatcatdog
```

Quiz 10: Answer

```
def find_second(search, target):
      first = search.find(target)
      second = search.find(target, first + 1)
      return second
```

You could eliminate the variable **second**:

```
def find_second(search, target):
      first = search.find(target)
      return search.find(target, first + 1)
```

You could even reduce this to one line by eliminating the variable **first**:

```
def find_second(search, target):
    return search.find(target, search.find(target) + 1)
```

Quiz 11: Answer

The correct answer is d. The meaning of = (assignment) and == (equality comparison) are very different:

- **i = 21** assigns 21 to **i**
- **i == 21** is a comparison that will output **True** or **False**

Quiz 12: Answer

```
def bigger(a, b):
    if a > b:
       return a
    return b
```

Quiz 13: Answer

```
def is_friend(name):
```

```
    if name [0] == 'D'
          return True
    else:
          return False
```

There is no real need to use if statement here, since we can just return the result of the comparison directly:

```
def is_friend(name):
    return name [0] == 'D'
```

Quiz 14: Answer

```
def is_friend(name):
     if name[0] == 'D':
          return True
      if name [0] == 'N':
          return True
      return False
```

Another way to define this would be to use an **else** clause:

```
def is_friend(name):
      if name[0] == 'D':
          return True
      else:
          if name [0] == 'N':
              return True
          else:
              return False
```

Note how the inner if statement is intended inside the else clause.

A third way of writing this would be to use an or expression, which we will describe next:

```
def is_friend(name):
      if name[0] == 'D' or name[0] == 'N'
```

Quiz 15: Answer

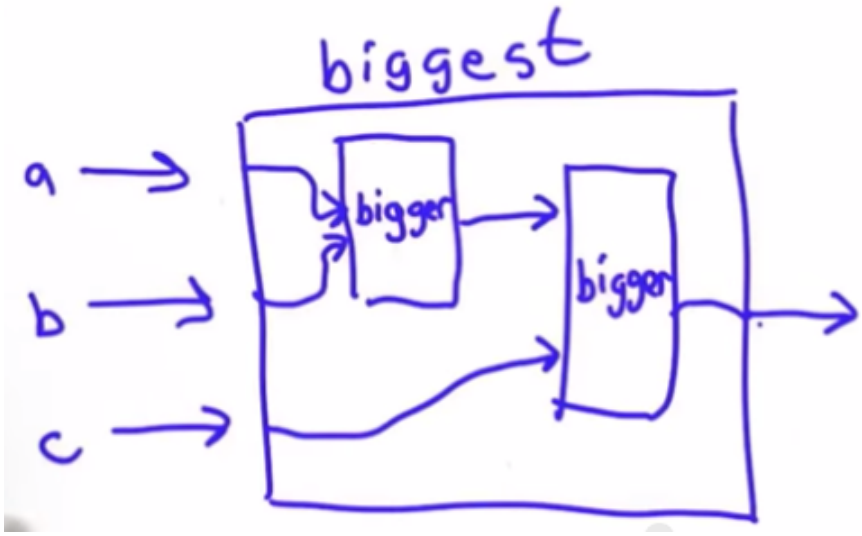
```
def biggest (a, b, c):
      if a > b:
          if a > c:
              return a
          else:
              return c
      else: # b >= a
```

```
        if b > c:
            return b
        else: # c >=b >= a
            return c
```

Another way to answer this that would be shorter and simpler is to use the **bigger** procedure we defined earlier:

```
def bigger (a, b):
    if a > b:
            return a
    else:
            return b
```

We can define **biggest** by composing two calls to **bigger**.



The code below is a much shorter way of defining biggest, taking advantage of the earlier definition of **bigger**:

```
biggest(a, b, c):
          return bigger(bigger(a, b), c)
```

An even simpler way to define **bigger** would be to use Python's built-in **max** operator, which works on any number of inputs and outputs the maximum value. We could then define:

```
def biggest(a, b, c):
      return max(a, b, c)
```

Of course, if you already knew about the built-in **max** operator, there would be no need to define **biggest** at all. The point is you know enough now to define it yourself!

It's even better than that: you actually know enough already to write every possible computer program!

Quiz 16: Answer

- d. Print out the numbers from 1 to 10.

Quiz 17: Answer

- e. This will run forever because the test condition of the loop is always **True**.

Quiz 18: Answer

```
def print_numbers(n):
      i = 1
      while i <= n:
          print i
          i = i + 1

print_numbers(3)
1
2
3
```

Another approach:

```
def print_numbers(n):
i = 0
while i < n:
      i = i + 1
      print i

print_numbers(3)
1
2
3
```

Quiz 19: Answer

Here is one solution:

```
def factorial (n):
      result = 1
      while n >=1 :
          result = result * n
          n = n -1
      return result
```

```
print factorial(4)
24
```

This result states that there are 24 different ways for the baby to arrange the blocks. You could also use this program to figure out how many different ways to arrange a deck of 52 playing cards, which is a really big number. Give it a try in your interpreter.

Quiz 20: Answer

- The answers are a and d.

Quiz 21: Answer

The answer is c, the multiple assignment expression swaps the value of *s* and *t*. Think about how this is different from writing two assignment statements. In the multiple assignment expression, both of the values on the right side get evaluated before they get their assignments.

Quiz 22: Answer

Write an **if** statement that will return *None* for the **url** when no hyperlink is found in the page.

```
def get_next_target(page):
      start_link = page.find('<a href=')
      if start_link == -1:
          return None, 0
      start_quote = page.find('"', start_link)
      end_quote = page.find('"', start_quote + 1)
      url = page[start_quote + 1:end_quote]
      return url, end_quote

url, endpos = get_next_target('Not "good" at all!>link!</a>')
if url:
      print "Here!"
else:
      print "Not here!"
print url

Not here!
```

When a string, such as **url**, is used as the condition in the **if** statement, the condition is evaluated as **True** for any nonempty string and **False** whenever the string is empty or the condition evaluates to *None*.

Quiz 23: Answer

```
def print_all_links(page):
      while True:
          url, endpos = get_next_target(page)
          if url:
              print url
              page = page[endpos:]
          else:
              break

print_all_links('this <a href="test1">link 1</a> is <a href="test2">link 2</
  test1
  test2
  test3
```

Thank's to mongnr for his help !