STRINGS, BRANCHING, ITERATION

name = value

VARIABLES (REVISITED)

- **name**
  - descriptive
  - meaningful
  - helps you re-read code
  - cannot be keywords

- **value**
  - information stored
  - can be updated by reassigning it using another assignment

VARIABLE BINDING WITH =

- compute the **right hand side → VALUE**
- store it (aka bind it) in the **left hand side → VARIABLE**
- left hand side will be replaced with new value
- = is called assignment

```
x = 2
x = x*x
y = x+1
```

Compute value first, then bind it to variable name; this will overwrite value of x

y = 5

BINDING EXAMPLE

- swap variables
  - is this ok?

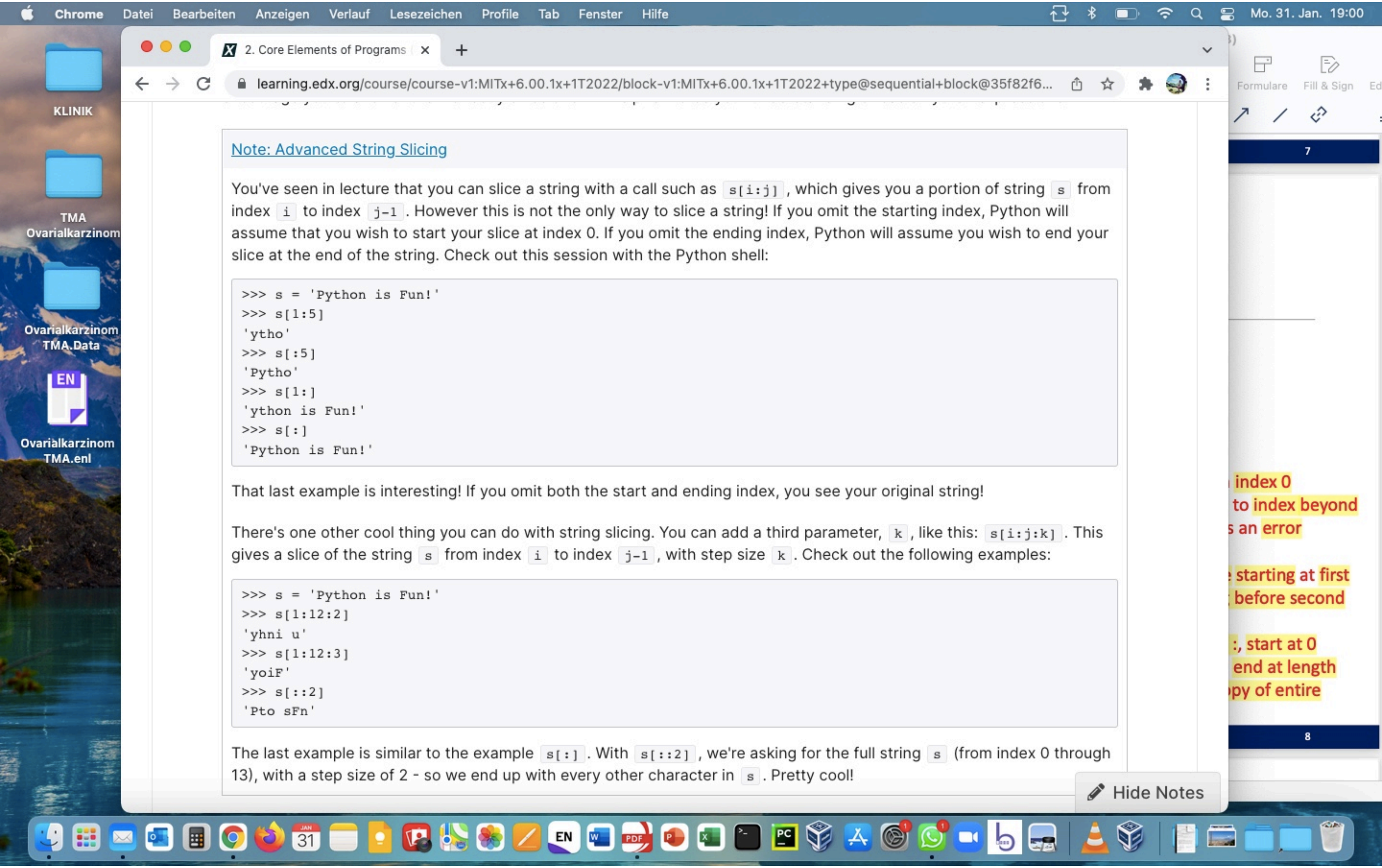
didnt swap them because there's
a sequence to this operation

```
x = 1
y = 2
y = x
x = y
```

This does NOT
do what you
think it does!

- swap variables
  - this is ok!

```
x = 1
y = 2
temp = y
y = x
x = temp
```



Note: Advanced String Slicing

You've seen in lecture that you can slice a string with a call such as `s[i:j]`, which gives you a portion of string `s` from index `i` to index `j-1`. However this is not the only way to slice a string! If you omit the starting index, Python will assume that you wish to start your slice at index 0. If you omit the ending index, Python will assume you wish to end your slice at the end of the string. Check out this session with the Python shell:

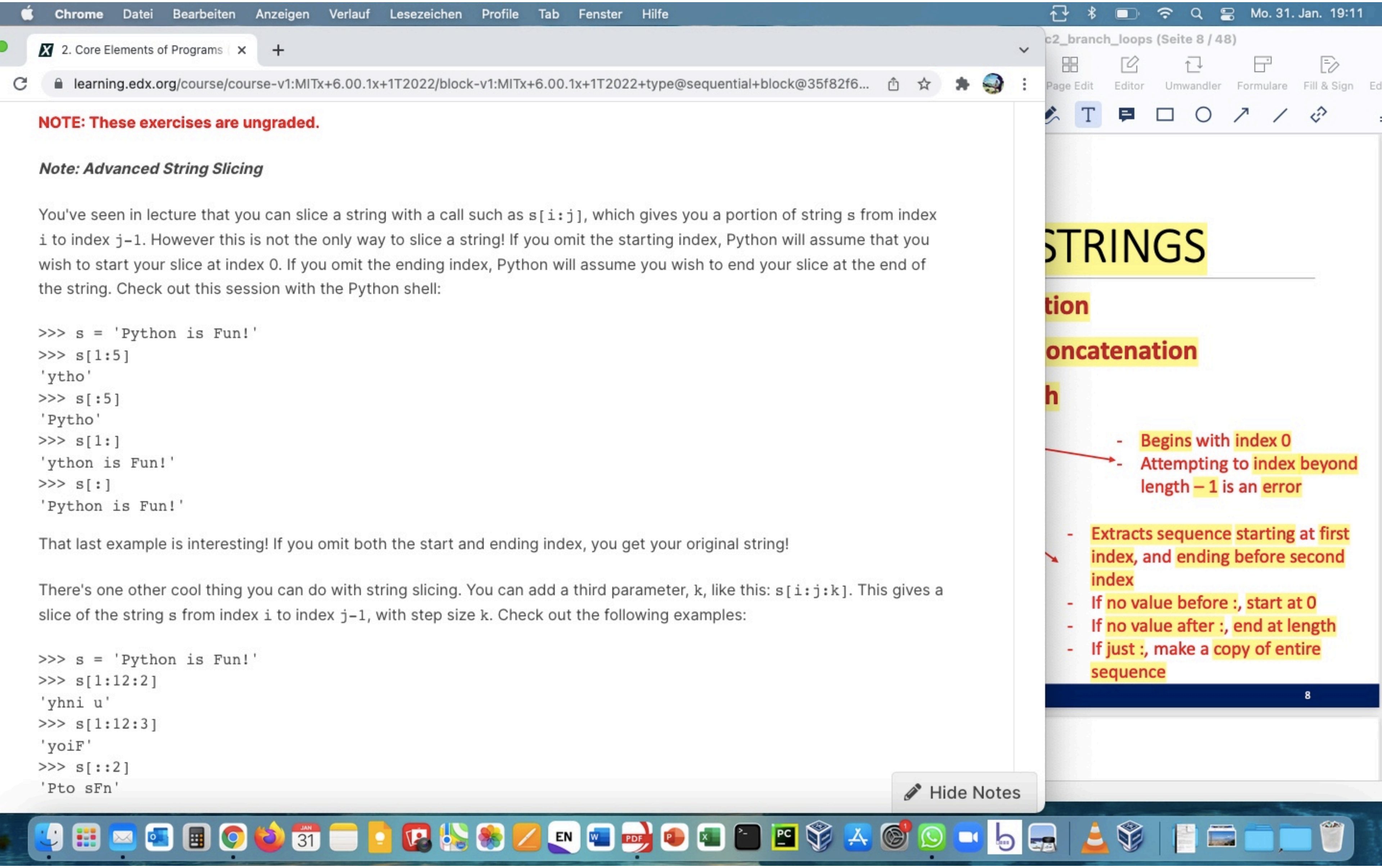
```
>>> s = 'Python is Fun!'
>>> s[1:5]
'ytho'
>>> s[:5]
'Pytho'
>>> s[1:]
'ython is Fun!'
>>> s[:]
'Python is Fun!'
```

That last example is interesting! If you omit both the start and ending index, you see your original string!

There's one other cool thing you can do with string slicing. You can add a third parameter, `k`, like this: `s[i:j:k]`. This gives a slice of the string `s` from index `i` to index `j-1`, with step size `k`. Check out the following examples:

```
>>> s = 'Python is Fun!'
>>> s[1:12:2]
'yhni u'
>>> s[1:12:3]
'yoiF'
>>> s[::2]
'Pto sFn'
```

The last example is similar to the example `s[:]`. With `s[::2]`, we're asking for the full string `s` (from index 0 through 13), with a step size of 2 - so we end up with every other character in `s`. Pretty cool!


Chrome
Datei
Bearbeiten
Anzeigen
Verlauf
Lesezeichen
Profile
Tab
Fenster
Hilfe
Mo. 31. Jan. 19:11
2. Core Elements of Programs
learning.edx.org/course/course-v1:MITx+6.00.1x+1T2022/block-v1:MITx+6.00.1x+1T2022+type@sequential+block@35f82f6...
c2_branch_loops (Seite 8 / 48)
Page Edit
Editor
Umwandler
Formulare
Fill & Sign
Ed
Hide Notes

NOTE: These exercises are ungraded.

Note: Advanced String Slicing

You've seen in lecture that you can slice a string with a call such as `s[i:j]`, which gives you a portion of string `s` from index `i` to index `j-1`. However this is not the only way to slice a string! If you omit the starting index, Python will assume that you wish to start your slice at index 0. If you omit the ending index, Python will assume you wish to end your slice at the end of the string. Check out this session with the Python shell:

```
>>> s = 'Python is Fun!'
>>> s[1:5]
'ytho'
>>> s[:5]
'Pytho'
>>> s[1:]
'ython is Fun!'
>>> s[:]
'Python is Fun!'
```

That last example is interesting! If you omit both the start and ending index, you get your original string!

There's one other cool thing you can do with string slicing. You can add a third parameter, `k`, like this: `s[i:j:k]`. This gives a slice of the string `s` from index `i` to index `j-1`, with step size `k`. Check out the following examples:

```
>>> s = 'Python is Fun!'
>>> s[1:12:2]
'yhni u'
>>> s[1:12:3]
'yoiF'
>>> s[::2]
'Pto sFn'
```

TRINGS

tion

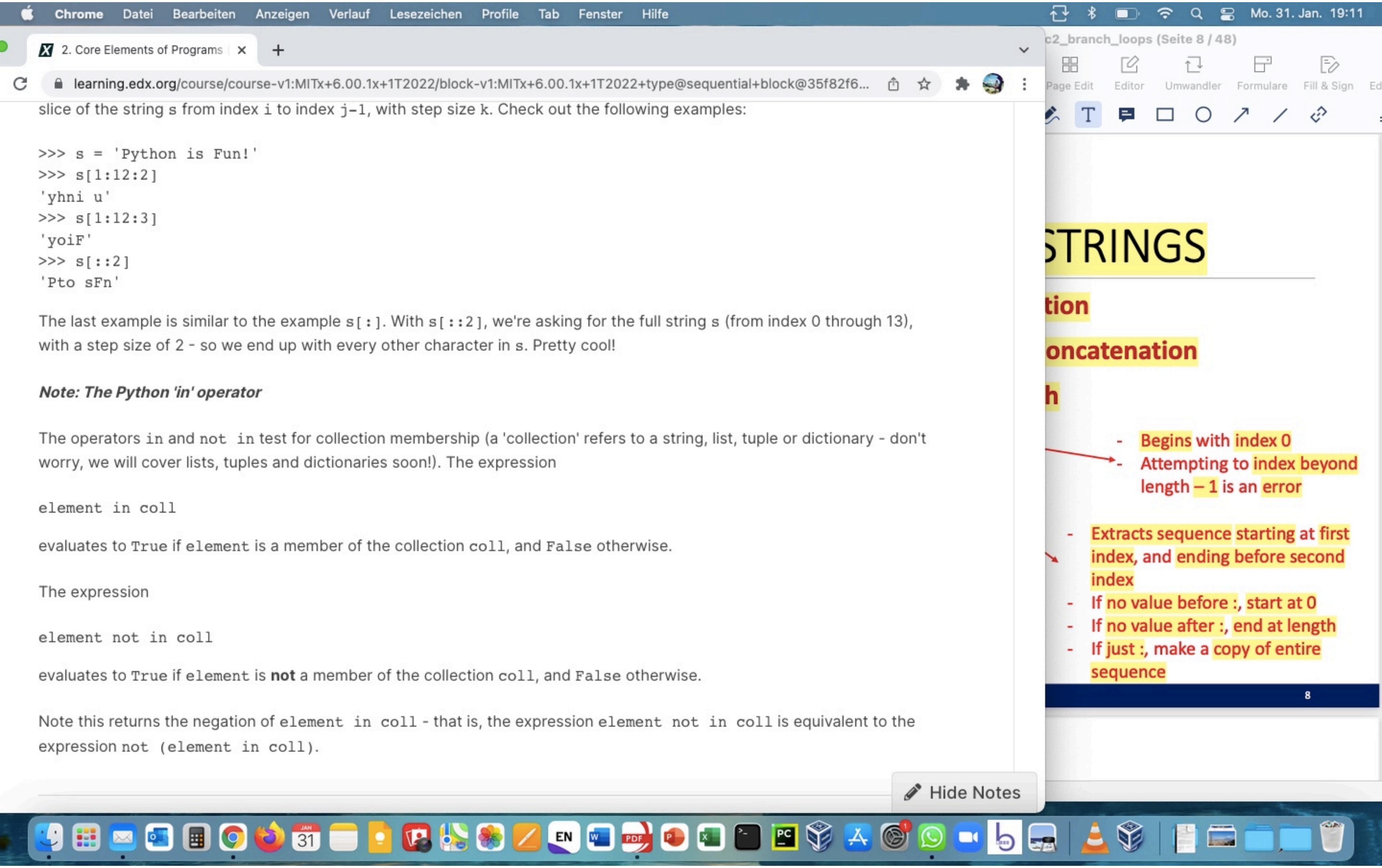
oncatenation

h

- Begins with index 0
- Attempting to index beyond length – 1 is an error

- Extracts sequence starting at first index, and ending before second index
- If no value before :, start at 0
- If no value after :, end at length
- If just :, make a copy of entire sequence

8


Chrome
Datei
Bearbeiten
Anzeigen
Verlauf
Lesezeichen
Profile
Tab
Fenster
Hilfe
Mo. 31. Jan. 19:11
2. Core Elements of Programs
learning.edx.org/course/course-v1:MITx+6.00.1x+1T2022/block-v1:MITx+6.00.1x+1T2022+type@sequential+block@35f82f6...
c2_branch_loops (Seite 8 / 48)
Page Edit
Editor
Umwandler
Formulare
Fill & Sign
Ed
Hide Notes

slice of the string s from index i to index j-1, with step size k. Check out the following examples:

```
>>> s = 'Python is Fun!'
>>> s[1:12:2]
'yhni u'
>>> s[1:12:3]
'yoiF'
>>> s[::2]
'Pto sFn'
```

The last example is similar to the example `s[:]`. With `s[::2]`, we're asking for the full string `s` (from index 0 through 13), with a step size of 2 - so we end up with every other character in `s`. Pretty cool!

Note: The Python 'in' operator

The operators `in` and `not in` test for collection membership (a 'collection' refers to a string, list, tuple or dictionary - don't worry, we will cover lists, tuples and dictionaries soon!). The expression

```
element in coll
```

evaluates to `True` if `element` is a member of the collection `coll`, and `False` otherwise.

The expression

```
element not in coll
```

evaluates to `True` if `element` is **not** a member of the collection `coll`, and `False` otherwise.

Note this returns the negation of `element in coll` - that is, the expression `element not in coll` is equivalent to the expression `not (element in coll)`.

STRINGS

tion

oncatenation

h

- Begins with index 0
- Attempting to index beyond length – 1 is an error

- Extracts sequence starting at first index, and ending before second index
- If no value before :, start at 0
- If no value after :, end at length
- If just :, make a copy of entire sequence

8

TYPES

- variables and expressions
  - `int`
  - `float`
  - `bool`
  - `string` -- NEW
  - … and others we will see later

STRINGS

a sequence of characters

- letters, special characters, spaces, digits
- enclose in **quotation marks** or **single quotes**

```
hi = "hello there”
greetings = ‘hello’
```

quotation marks if apostrofs are needed inside our string: "isn't"

- **concatenate** strings

```
name = "eric"
greet = hi + name
greeting = hi + " " + name
```

space

we have overloaded it:
addition can be applied to different data types:
if the two things are strings, concatenate them.
if I give you two numbers, just add them together using straightforward arithmetic.

OPERATIONS ON STRINGS

- `‘ab’+ ‘cd’` → **concatenation**
- `3* ‘eric’` → **successive concatenation**
- `len(‘eric’)` → the **length**
- `‘eric’[1]` → **indexing**
    - Begins with index 0
    - Attempting to index beyond length – 1 is an error
- `‘eric’[1:3]` → **slicing**
    - Extracts sequence starting at first index, and ending before second index
    - If no value before :, start at 0
    - If no value after :, end at length
    - If just :, make a copy of entire sequence

INPUT/OUTPUT: print

- used to **output** stuff to console
- keyword is `print`

```
x = 1
print(x)
x_str = str(x)
print("my fav num is", x, ".", "x =", x)
print("my fav num is " + x_str + ". " + "x = " + x_str)
```

INPUT/OUTPUT: `input("")`

- prints whatever is within the quotes
- user types in something and hits enter
- returns entered sequence
- can bind that value to a variable so can reference

```
text = input("Type anything... ")
print(5*text)
```

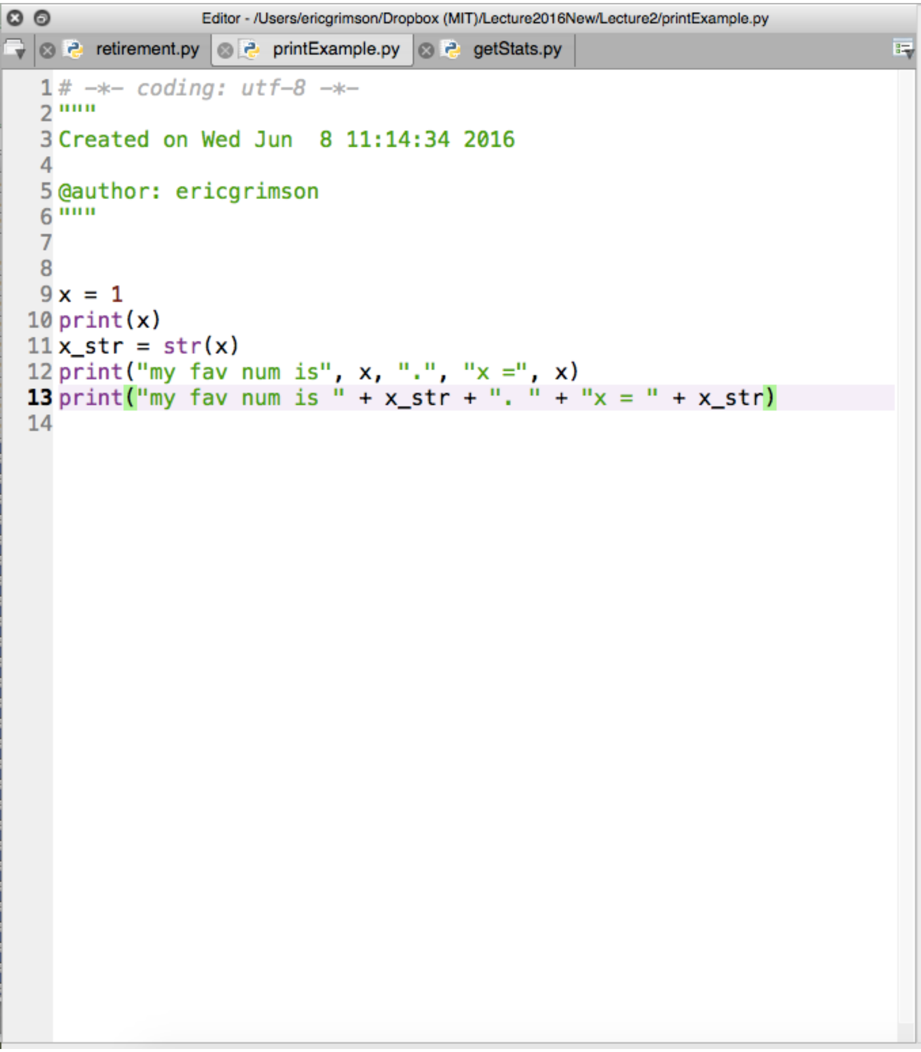
- `input` **returns a string** so must cast if working with numbers

```
num = int(input("Type a number... "))
print(5*num)
```

IDE’s

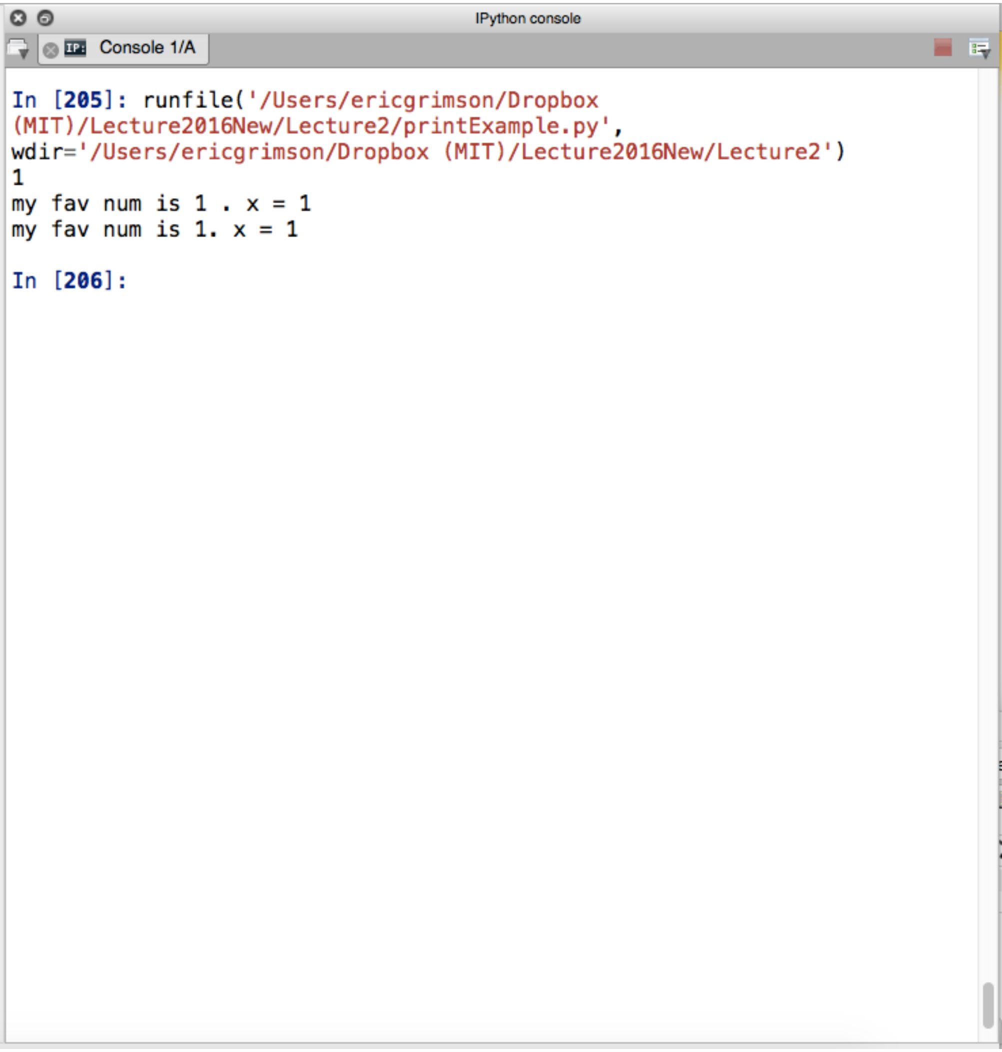
- painful to just type things into a shell
- better to have a text editor – integrated development environment (IDE)
  - IDLE or Anaconda are examples
- comes with
  - Text editor – use to enter, edit and save your programs
  - Shell – place in which to interact with and run your programs; standard methods to evaluate your programs from the editor or from stored files
  - Integrated debugger (we’ll use later)

text file


Editor - /Users/ericgrimson/Dropbox (MIT)/Lecture2016New/Lecture2/printExample.py
retirement.py
printExample.py
getStats.py

```python
# -*- coding: utf-8 -*-
"""
Created on Wed Jun  8 11:14:34 2016

@author: ericgrimson
"""


x = 1
print(x)
x_str = str(x)
print("my fav num is", x, ".", "x =", x)
print("my fav num is " + x_str + ". " + "x = " + x_str)

```

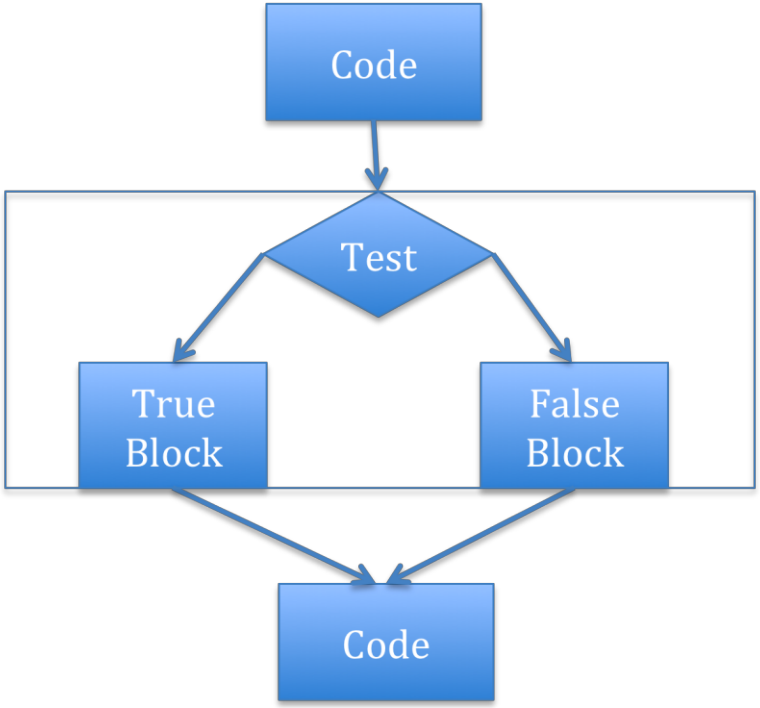
shell



```
In [205]: runfile('/Users/ericgrimson/Dropbox
(MIT)/Lecture2016New/Lecture2/printExample.py',
wdir='/Users/ericgrimson/Dropbox (MIT)/Lecture2016New/Lecture2')
1
my fav num is 1 . x = 1
my fav num is 1. x = 1

In [206]:
```

BRANCHING PROGRAMS (REVISITED)

- The simplest branching statement is a **conditional**
  - A test (expression that evaluates to `True` or `False`)
  - A block of code to execute if the test is `True`
  - An optional block of code to execute if the test is `False`



COMPARISON OPERATORS ON `int` and `float`

- `i` and `j` are any variable names

`i>j`

`i>=j`

`i<j`

`i<=j`

`i==j` → **equality** test, `True` if `i` equals `j`

`i!=j` → **inequality** test, `True` if `i` not equal to `j`

LOGIC OPERATORS ON bools

- `a` and `b` are any variable names

`not a` → `True` if `a` is `False`
`False` if `a` is `True`

`a and b` → `True` if both are `True`

`a or b` → `True` if either or both are `True`

CONTROL FLOW - BRANCHING

```
if <condition>:
    <expression>
    <expression>
    ...
```

```
if <condition>:
    <expression>
    <expression>
    ...
else:
    <expression>
    <expression>
    ...
```

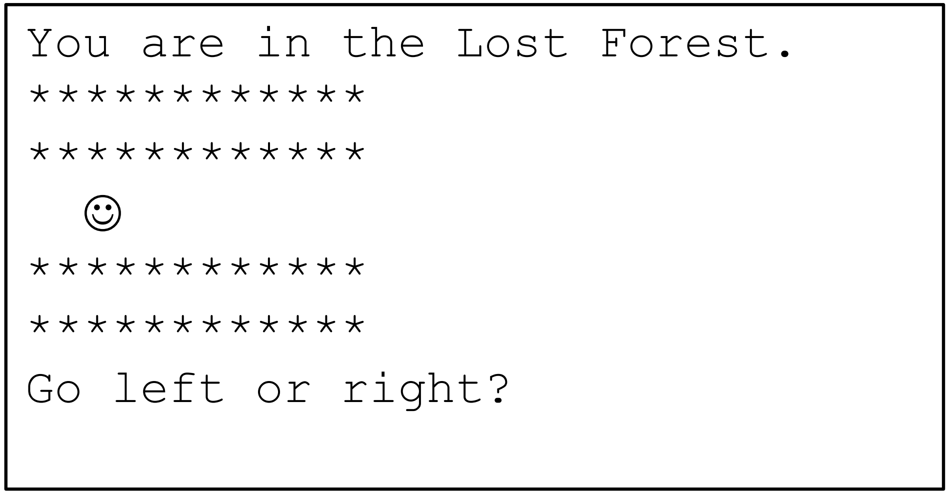
```
if <condition>:
    <expression>
    <expression>
    ...
elif <condition>:
    <expression>
    <expression>
    ...
else:
    <expression>
    <expression>
    ...
```

only flow through code once

- `<condition>` has a value `True` or `False`
- evaluate expressions in that block if `<condition>` is `True`

USING CONTROL IN LOOPS

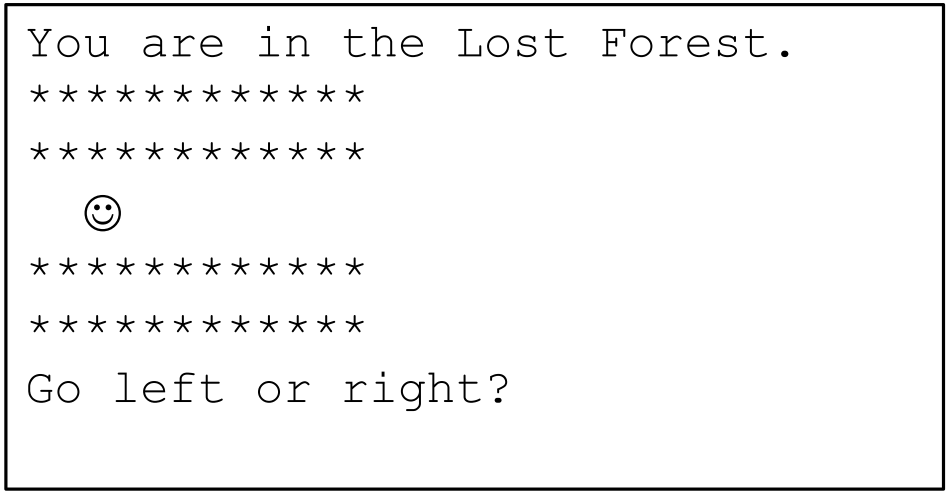
- simple branching programs just make choices, but path through code is still linear
- sometimes want to reuse parts of the code indeterminate number of times



```
You are in the Lost Forest.
************
************
  ☺
************
************
Go left or right?
```

- You are playing a video game, and are lost in some woods
- If you keep going right, takes you back to this same screen, stuck in a loop

```
if <exit right>:
    <set background to woods_background>
    if <exit right>:
        <set background to woods_background>
        if <exit right>:
            <set background to woods_background>
            and so on and so on...
        else:
            <set background to exit_background>
    else:
        <set background to exit_background>
else:
    <set background to exit_background>
```



```
You are in the Lost Forest.
************
************
  ☺
************
************
Go left or right?
```

- You are playing a video game, and are lost in some woods
- If you keep going right, takes you back to this same screen, stuck in a loop

True

```
while <exit_right>:
    <set background to woods_background>
<set background to exit_background>
```

I'm going to do this,
and I'm going to go back around and do it again.
And I'll keep looping around and around on this while
until this condition is false, in which case I'll jump out and do the next kind of thing

CONTROL FLOW: `while` LOOPS

```
while <condition>:
    <expression>
    <expression>
    ...
```

- `<condition>` evaluates to a Boolean
- if `<condition>` is `True`, do all the steps inside the while code block
- check `<condition>` again
- repeat until `<condition>` is `False`

while LOOP EXAMPLE

```
You are in the Lost Forest.
************
************
  ☺
************
************
Go left or right?
```

As long as n is equal to right, it will keep prompting me, asking for an input, until I finally type in left, at which case, this will be false, and I'll jump down and pick up the print statement

```
n = input("You are in the Lost Forest. Go left or right? ")
while n == "right":
    n = input("You are in the Lost Forest. Go left or right? ")
print("You got out of the Lost Forest!”)
```

CONTROL FLOW: while and for LOOPS

```
# more complicated with while loop
n = 0
while n < 5:
    print(n)
    n = n+1
```

I need to set up a variable outside so that I can test it

And I need inside to have something that actually changes that variable, otherwise I'm never going to get out of the loop.

-> you can write a for loop using a while loop

```
# shortcut with for loop
for n in range(5):
    print(n)
```

range(5) gives us the integers 0, 1, 2, 3, 4 in turn

for is going to work through all of the values returned by that expression range (5) one at a time, executing the body of the code

CONTROL FLOW: `for` LOOPS

```
for <variable> in range(<some_num>):
    <expression>
    <expression>
    ...
```

- each time through the loop, `<variable>` takes a value
- first time, `<variable>` starts at the smallest value
- next time, `<variable>` gets the prev value + 1
- etc.

range(start,stop,step)

- default values are `start = 0` and `step = 1` and is optional
- loop until value is `stop - 1`

```
mysum = 0
for i in range(7, 10):
    mysum += i
print(mysum)
```

give me the range of numbers from 7 up to but not including 10

+= add to my sum the value of i

```
mysum = 0
for i in range(5, 11, 2):
    mysum += i
print(mysum)
```

Start at 5, end when I get up to 11, but do it by 2

break STATEMENT

- immediately exits whatever loop it is in
- skips remaining expressions in code block
- exits only innermost loop

```
while <condition_1>:
    while <condition_2>:
        <expression_a>
        break
        <expression_b>
    <expression_c>
```

when I hit break, it will never execute that expression.
It will pop out of all of this and pick up at that point

break STATEMENT

```
mysum = 0
for i in range(5, 11, 2):
    mysum += i
    if mysum == 5:
        break
print(mysum)
```

- what happens in this program?

first time around i is going to have the value 5.
I'm going to increment my sum by 1, but then this test is true,
and this break pulls me out of that entire loop and stops the computation.

`for` VS `while` LOOPS

`for` loops

- **know** number of iterations

Because I've defined the range of things over which I'm going to do the work

- can **end early** via `break`
- uses a **counter**

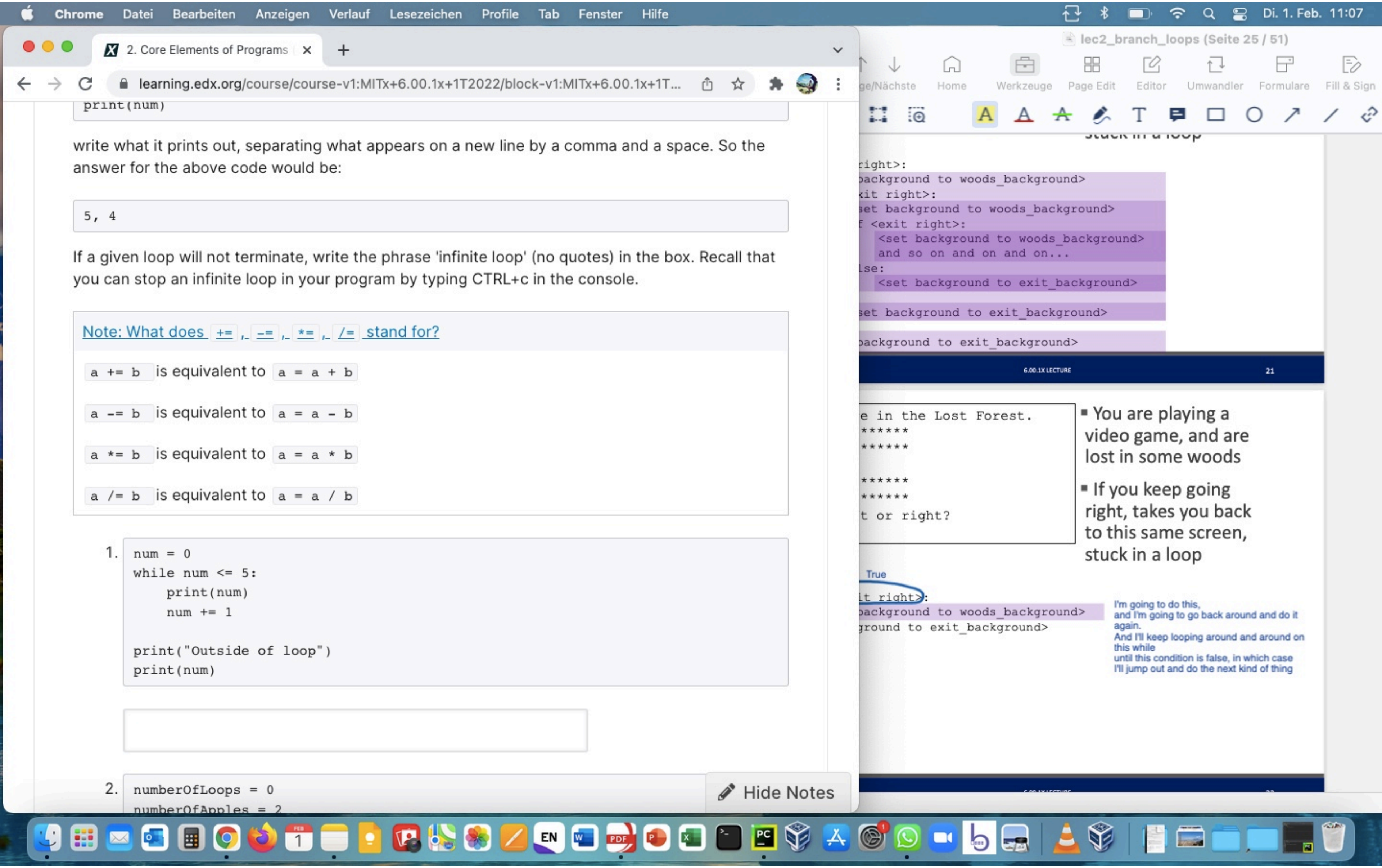
captured inside the for loop itself

- **can rewrite** a `for` loop using a `while` loop

by taking that variable that I'm using, that counter I'm using, pulling it outside, initializing it, and explicitly doing the increment to the counter inside of the loop

`while` loops

- **unbounded** number of iterations
- can **end early** via `break`
- can use a **counter but must initialize** before loop and increment it inside loop
- **may not be able to rewrite** a `while` loop using a `for` loop



```
print(num)
```

write what it prints out, separating what appears on a new line by a comma and a space. So the answer for the above code would be:

```
5, 4
```

If a given loop will not terminate, write the phrase 'infinite loop' (no quotes) in the box. Recall that you can stop an infinite loop in your program by typing CTRL+c in the console.

Note: What does `+=`, `-=`, `*=`, `/=` stand for?

`a += b` is equivalent to `a = a + b`

`a -= b` is equivalent to `a = a - b`

`a *= b` is equivalent to `a = a * b`

`a /= b` is equivalent to `a = a / b`

1.
```
num = 0
while num <= 5:
    print(num)
    num += 1

print("Outside of loop")
print(num)
```

2.
```
numberOfLoops = 0
numberOfApples = 2
```

stuck in a loop

```
right>:
background to woods_background>
xit right>:
set background to woods_background>
f <exit right>:
  <set background to woods_background>
  and so on and on and on...
lse:
  <set background to exit_background>

set background to exit_background>

background to exit_background>
```

6.00.1X LECTURE 21

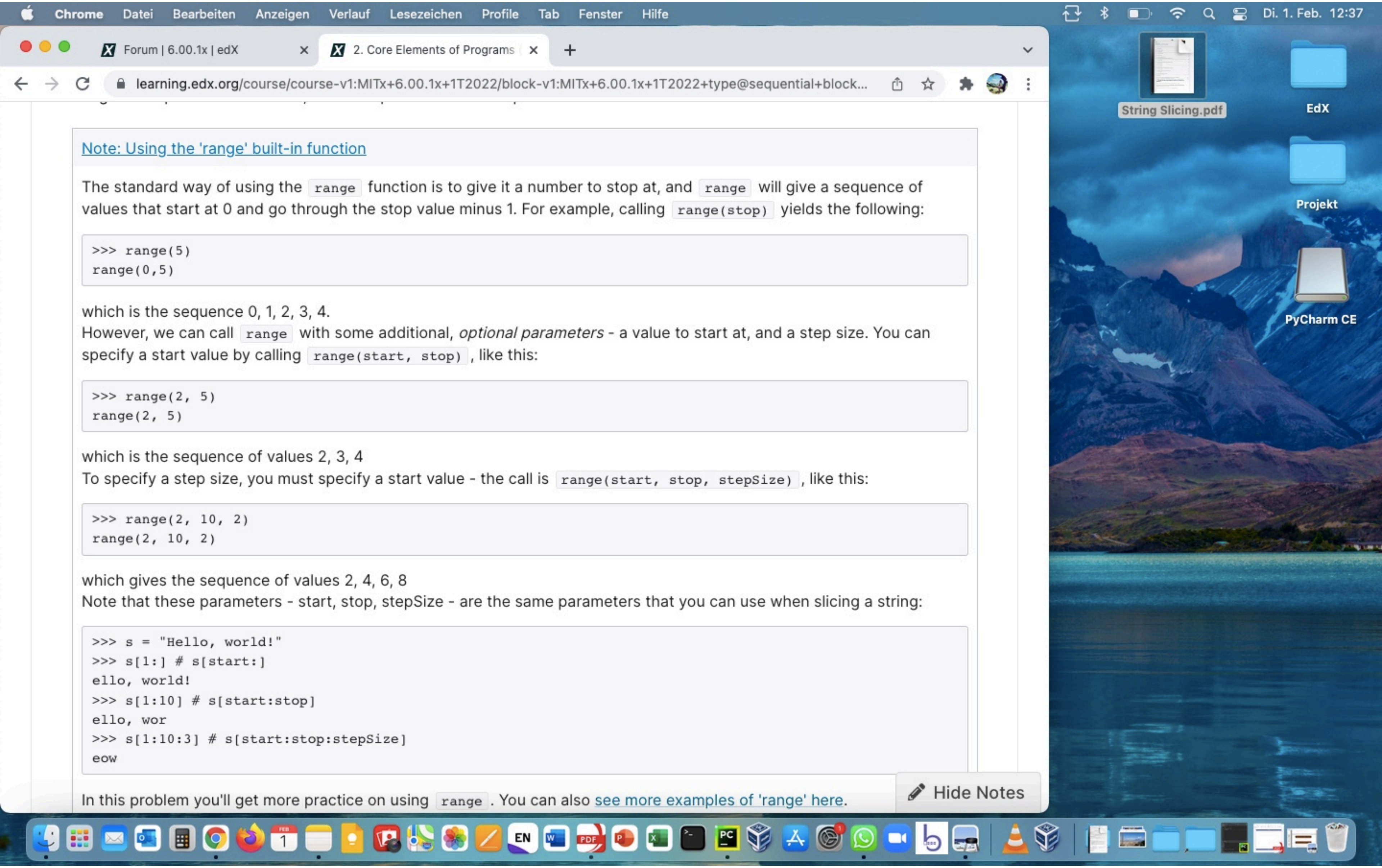
```
e in the Lost Forest.
******
******

******
******
t or right?
```

- You are playing a video game, and are lost in some woods
- If you keep going right, takes you back to this same screen, stuck in a loop

True

```
it right>:
background to woods_background>
ground to exit_background>
```

I'm going to do this,
and I'm going to go back around and do it again.
And I'll keep looping around and around on this while
until this condition is false, in which case I'll jump out and do the next kind of thing



Note: Using the 'range' built-in function

The standard way of using the `range` function is to give it a number to stop at, and `range` will give a sequence of values that start at 0 and go through the stop value minus 1. For example, calling `range(stop)` yields the following:

```
>>> range(5)
range(0,5)
```

which is the sequence 0, 1, 2, 3, 4.
However, we can call `range` with some additional, *optional parameters* - a value to start at, and a step size. You can specify a start value by calling `range(start, stop)`, like this:

```
>>> range(2, 5)
range(2, 5)
```

which is the sequence of values 2, 3, 4
To specify a step size, you must specify a start value - the call is `range(start, stop, stepSize)`, like this:

```
>>> range(2, 10, 2)
range(2, 10, 2)
```

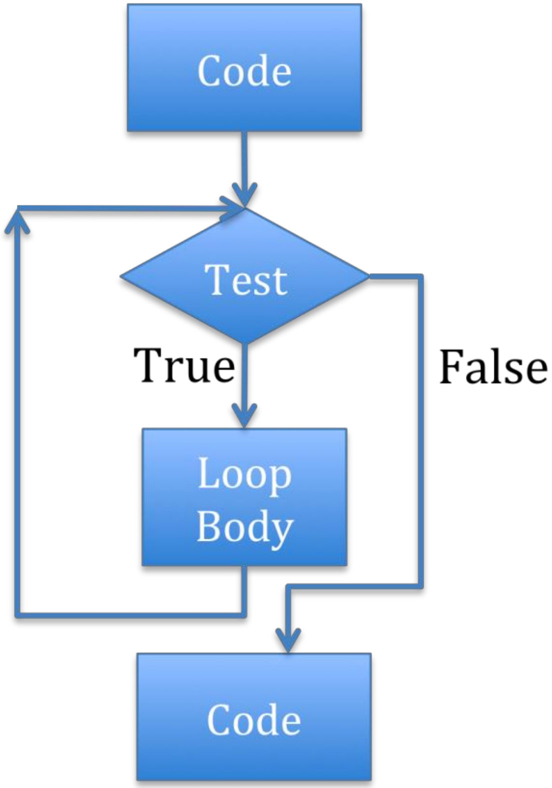
which gives the sequence of values 2, 4, 6, 8
Note that these parameters - start, stop, stepSize - are the same parameters that you can use when slicing a string:

```
>>> s = "Hello, world!"
>>> s[1:] # s[start:]
ello, world!
>>> s[1:10] # s[start:stop]
ello, wor
>>> s[1:10:3] # s[start:stop:stepSize]
eow
```

In this problem you'll get more practice on using `range`. You can also see more examples of 'range' here.

ITERATION

- Concept of iteration let's us extend simple branching algorithms to be able to write programs of arbitrary complexity
  - Start with a test
  - If evaluates to `True`, then execute loop body once, and go back to reevaluate the test
  - Repeat until test evaluates to `False`, after which code following iteration statement is executed



AN EXAMPLE

```
x = 3
ans = 0
itersLeft = x
while (itersLeft != 0):
    ans = ans + x
    itersLeft = itersLeft - 1
print(str(x) + '*' + str(x) + ' = ' + str(ans))
```

This code squares the value of x by repetitive addition.

(of x)

When x was equal to 3, I ran through this portion of the code three times. 3*3=9
When x was equal to 5, I went through it five times. 5*5=25

STEPPING THROUGH CODE

```
x = 3
ans = 0
itersLeft = x
while (itersLeft != 0):
    ans = ans + x
    itersLeft = itersLeft - 1
print(str(x) + '*' + str(x) + ' = ' + str(ans))
```

x	ans	itersLeft
3	0	3
	3	2
	6	1
	9	0

+ x
+ x
+ x

-1
-1
-1

while accumulating an answer in ans

variable to determine numbers of iterations is reduced by every repetition

Some properties of iteration loops:

- need to set an iteration variable outside the loop
- need to test variable to determine when done
- need to change variable within the loop, in addition to other work

ITERATIVE CODE

- Branching structures (conditionals) let us jump to different pieces of code based on a test
  - Programs are **constant time**
- Looping structures (e.g., while) let us repeat pieces of code until a condition is satisfied
  - Programs now take time that depends on values of variables, as well as length of program

CLASSES OF ALGORITHMS

- Iterative algorithms allow us to do more complex things than simple arithmetic
- We can repeat a sequence of steps multiple times based on some decision; leads to new classes of algorithms
- One useful example are “guess and check” methods

GUESS AND CHECK

- Remember our “declarative” definition of square root of `x`
- If we could guess possible values for square root (call it `g`), then can use definition to check if `g*g = x`
- We just need a good way to generate guesses

FINDING CUBE ROOT OF INTEGER

3√x=k
k**3=x

- One way to use this idea of generating guesses in order to find a cube root of `x` is to first try `0**3`, then `1**3`, then `2**3`, and so on
- Can stop when reach `k` such that `k**3 > x`
- Only a finite number of cases to try

3√x=ans
ans**3=x

SOME CODE

```
x = int(input('Enter an integer: '))
ans = 0
while ans**3 < x:
    ans = ans + 1
if ans**3 != x:
    print(str(x) + ' is not a perfect cube')
else:
    print('Cube root of ' + str(x) + ' is ' + str(ans))
```

input returns something as a string,
so I'm going to convert it into an integer.
It's going to assume I typed in an integer.

It's simply using a
loop, right here, to
generate guesses

and then as long as I have something that's
less than the thing I'm trying to find the cube of,
I'm just going to increment add 1 to it.

And I'm going to keep doing that until I get
something that

something where the cube is either
equal to x or greater than x.
And once I get there, I'll simply check to see,
did I actually get the cube, by doing a test.

And it's going to keep doing that until it gets
either to something that is the right thing,
or has gone too far, in which case,
I'm simply going to do a check to see which case I'm in.

EXTENDING SCOPE

- Only works for positive integers
- Easy to fix by keeping track of sign, looking for solution to positive case

easily extend my code to build new versions of things
to handle cases that I didn't think about when I wrote
the first version of the code

SOME CODE

```
x = int(input('Enter an integer: '))
ans = 0
while ans**3 < abs(x):
    ans = ans + 1
if ans**3 != abs(x):
    print(str(x) + ' is not a perfect cube')
else:
    if x < 0:
        ans = - ans
    print('Cube root of ' + str(x) + ' is ' + str(ans))
```

"abs," which is a built in function,
to take the absolute value of x

decide down here
whether in fact I want the negative or positive version

LOOP CHARACTERISTICS

- Need a loop variable
  - Initialized outside loop
  - Changes within loop
  - Test for termination depends on variable
- Useful to think about a **decrementing function**
  - Maps set of program variables into an integer
  - When loop is entered, value is non-negative
  - When value is <= 0, loop terminates, and
  - Value is decreased every time through loop
- Here we can use `abs(x) - ans**3`

WHAT IF MISS A CONDITION?

- Suppose we don't initialize the variable?
  - Likely get a NameError; or worse use an ~~expected~~ unexpected value to initiate the computation
- Suppose we don't change the variable inside the loop?
  - Will end up in an infinite loop, never reaching the terminating condition

GUESS-AND-CHECK

- you are able to **guess a value** for solution
- you are able to **check if the solution is correct**

  you change the conditions inside the loop
- keep guessing until find solution or guessed all values
- the process is **exhaustive enumeration**

  One, you're going to exhaust all possible options to use.
  And two is it's going to take a while to run, so you get tired waiting for it to finish

CLEANER GUESS-AND-CHECK – cube root

```
cube = 8
```

use "range" to generate all possible things I want to use as a guess:
[0, 1, 2, 3, 4, 5, 6, 7, 8]

```
for guess in range(cube+1):
    if guess**3 == cube:
        print("Cube root of ", cube, ” is ", guess)
```

That code is going to run through all possible options for guess, but it's only going to print something out if in fact I find something that is the cube root-- if there is a perfect cube

CLEANER GUESS-AND-CHECK – cube root

```
cube = 8
for guess in range(abs(cube)+1):
    if guess**3 >= abs(cube):
        break
if guess**3 != abs(cube):
    print(cube, ’is not a perfect cube’)
else:
    if cube < 0:
        guess = -guess
    print('Cube root of ' + str(cube) + ' is ' + str(guess))
```

EXHAUSTIVE ENUMERATION

- Guess and check methods can work on problems with a finite number of possibilities (exhaustively testing all of those)
- Exhaustive enumeration is a good way to generate guesses in an organized manner