Reminder: What we did in Lecture 2 (Highlights)

- Boolean constants: True and False.
- Python’s Boolean operators: or, and, not
- Conditional Statements
- Nested conditional statements
- range
- Iteration
- Lists
- Iterating over ranges, lists, and strings
Lecture 3: Planned Topics

- Additional operations on lists and strings, e.g. *slicing*.
- Functions
- Equality and Identity
- Mutable vs. immutable classes
- Effects of Assignments
- Python’s Memory Model
- Un-assignment: Deletion
We introduced Boolean variables, but did not yet examine their type.
Type of Boolean Variables

We introduced Boolean variables, but did not yet examine their type.

```python
>>> type(True)
<class 'bool'>
>>> type(False)
<class 'bool'>
>>> x=True
>>> type(x)
<class 'bool'>
```

So `bool` is added to our arsenal of types.
More List Comprehension

List comprehension is a powerful tool, allowing the succinct, powerful creation of new lists from old ones.

```python
>>> l=['benny', 'daniel', 'ilan', 'adam', 'amir']

>>> [str.title(st) for st in l if len(st)>4]
['Benny', 'Daniel']

>>> [str.upper(st) for st in l if str.startswith(st,'a')]
['ADAM', 'AMIR']
```
Additional operations on lists: Slicing

```python
>>> list_a = [1,2,3,4,5,6,7,8,9,10]
>>> len(list_a)
10
>>> list_a[1:5]     # slicing
[2,3,4,5]
>>> list_a[0:10:2]  # slicing an arithmetic progression
[1,3,5,7,9]
>>> list_a[::2]     # shorthand for previous slicing
[1,3,5,7,9]
>>> list_a[::-1]    # reversing the list
[10,9,8,7,6,5,4,3,2,1]
>>> list_a[10::-1]  # same as before
[10,9,8,7,6,5,4,3,2,1]
>>> list_a[-1::-1]  # index -1 refers to last element
[10,9,8,7,6,5,4,3,2,1]
>>> list_a[-1:-11:-1] # and -11 here is one before first
[10,9,8,7,6,5,4,3,2,1]
>>> list_a[8:3:-2]  # arithmetic progression with $\delta = -2$
[9,7,5]
>>> list_a[3:8:-2]
[ ]            # outcome is an empty list. This is NOT an error
>>> list_a
[1,2,3,4,5,6,7,8,9,10]     # slicing did NOT change original list
```
Slicing Can Be Tricky: Help on Slicing

```python
>>> help(slice)
Help on class slice in module builtins:
class slice(object)
    | slice([start[, stop[, step]])
    | Create a slice object. Used for extended slicing (e.g. a[0:10:2]).
    | Methods defined here:
    # truncated additional info
indices(...)
    | S.indices(len) -> (start, stop, stride)
    |
    | Assuming a sequence of length len, calculate the start and stop
    | indices, and the stride length of the extended slice described by
    | S. Out of bounds indices are clipped in a manner consistent with the
    | handling of normal slices.
    |
    | -----------------------------------------------
    | Data descriptors defined here:
    |
    | start
    |
    | step
    |
    | stop
    |
    # truncated additional info
```
They Slice **Strings** Too, Don’t They?

```python
>>> len("Rye Bread")
9
>>> "Rye Bread"[0:9]  # everything
'Rye Bread'
>>> "Rye Bread"[:9]  # shorthand for previous
'Rye Bread'
```
They Slice Strings Too, Don’t They?

```python
>>> len("Rye Bread")
9
>>> "Rye Bread"[0:9] # everything
'Rye Bread'
>>> "Rye Bread"[:] # shorthand for previous
'Rye Bread'
>>> "Rye Bread"[:4] # first 4 characters
'Rye '
>>> "Rye Bread"[4:] # everything but first 4 characters
'Bread'
>>> "Rye Bread"[:4]+"Rye Bread"[4:] # concatenate prefix and suffix
'Rye Bread'
```
More Sliced Strings

>>> "Rye Bread"[-1]    # counting from right end: last character
'd'
>>> "Rye Bread"[-2]    # counting from right end: 2nd from last character
'a'
>>> "Rye Bread"[-10]   # pushing our luck here
Traceback (most recent call last):
  File "<pyshell#16>", line 1, in <module>
    "Rye Bread"[-10]
IndexError: string index out of range
More Sliced Strings

```python
>>> "Rye Bread"[-1]  # counting from right end: last character
'd'
>>> "Rye Bread"[-2]  # counting from right end: 2nd from last character
'a'
>>> "Rye Bread"[-10]  # pushing our luck here
Traceback (most recent call last):
  File "<pyshell#16>", line 1, in <module>
    "Rye Bread"[-10]
IndexError: string index out of range

>>> "Rye Bread"[-4:]  # last 4 characters (suffix)
'read'
>>> "Rye Bread"[:4]  # everything except last 4 characters (prefix)
'Rye B'

>>> "Rye Bread"[:4] + "Rye Bread"[-4:]  # concatenate prefix and suffix
'Rye Bread'
```
Last Sliced Strings

```python
>>> "Rye Bread"[0:9:2]  # every second character, starting from first
  'ReBed'
>>> "Rye Bread"[0::2]  # shorthand for previous
  'ReBed'
>>> "Rye Bread"[:9:2]  # shorthand for previous previous
  'ReBed'
>>> "Rye Bread"[:2]  # shorthand for previous previous previous
  'ReBed'
```
Last Sliced Strings

```python
>>> "Rye Bread"[0:9:2]  # every second character, starting from first
'ReBed'
>>> "Rye Bread"[0::2]  # shorthand for previous
'ReBed'
>>> "Rye Bread"[:9:2]  # shorthand for previous previous
'ReBed'
>>> "Rye Bread"[::2]  # shorthand for previous previous previous
'ReBed'

>>> "Rye Bread"[9::1]  # everything, backwards
'daerB eyR'
>>> "Rye Bread"[::-1]  # shorthand for previous
'daerB eyR'
```

Any iterable class (essentially different kinds of ordered sequences) in Python can also be indexed and sliced.
Functions
Functions

(Figures taken from Wikipedia and (the colorful one at the top right) from www.mathworks.com site.)
Let us start with excerpts from Wikipedia definition of the term function:

A function, in a mathematical sense, expresses the idea that one quantity (the argument of the function, also known as the input) completely determines another quantity (the value, or the output). A function assigns exactly one value to each input of a specified type. A specific input in a function is called an argument of the function. For each argument value $x$, the corresponding unique $y$ in the codomain is called the function value at $x$, output of $f$ for an argument $x$, or the image of $x$ under $f$. The image of $x$ may be written as $f(x)$ or as $y$. 
Functions in Programming Languages

We saw before a variant of the following piece of code

```python
>>> (True and (not True)) or ((not True) and True)
False
>>> (False and (not False)) or ((not False) and False)
False
>>> (True and (not False)) or ((not True) and False)
True
```

And remarked that it is annoying and time consuming to write and rewrite the same expression, only with different values.
Functions in Programming Languages

We saw before a variant of the following piece of code

```python
>>> (True and (not True)) or ((not True) and True)
False
>>> (False and (not False)) or ((not False) and False)
False
>>> (True and (not False)) or ((not True) and False)
True
```

And remarked that it is annoying and time consuming to write and rewrite the same expression, only with different values.

A function will come in handy:

```python
def xor(x,y):
    ''' computes xor of two Boolean variables, x and y '''
    return (x and (not y)) or ((not x) and y)
```

```python
>>> xor(True,True)
False
>>> xor(True,False)
True
>>> xor(False,False)
False
>>> xor(False,True)
True
```
Another Example: Functions in Programming Languages

We saw a variant of the following piece of code. We now achieve the same functionality but doing it the right way, using a function.

```python
def test4(n):
    """ tests divisibility by 4 and 8 ""

    if n % 4 != 0:
        print(n,"is not divisible by 4")
    else:
        if n % 8 == 0: # nested "if"
            print(n,"is divisible by 8")
        else: # nested "else"
            print(n,"is divisible by 4 but not by 8")
```

We place this code in a separate file, `divisibility_by_4.py`. The suffix `.py` identifies it as a Python program. To run it, hit the F5 button on your keyboard.
Another Example: Functions in Programming Languages

We saw a variant of the following piece of code. We now achieve the same functionality but doing it the right way, using a function.

```python
def test4(n):
    """ tests divisibility by 4 and 8"""

    if n % 4 != 0:
        print(n,"is not divisible by 4")
    else:
        if n % 8 == 0: # nested "if"
            print(n,"is divisible by 8")
        else: # nested "else"
            print(n,"is divisible by 4 but not by 8")
```

We place this code in a separate file, `divisibility_by_4.py`. The suffix `.py` identifies it as a Python program. To run it, hit the F5 button on your keyboard.

```bash
>>> test4(17)
17 is not divisible by 4
>>> test4(32)
32 is divisible by 8
>>> test4(12)
12 is divisible by 4 but not by 8
```
Documenting Python’s Functions

Python provides a mechanism to document functions. This is text (possibly multi line, unlike a comment) between triple quotes, called a docstring. (Three single quotes or three double double quotes.)

```python
def xor(x, y):
    '''computes xor of two Boolean variables, x and y'''
    return (x and (not y)) or ((not x) and y)
```

Everything between the start and end of the triple quotes is part of a single string, including carriage returns and other quote characters. You can use triple quotes anywhere, but they are most often used when defining docstrings.
Documenting Python’s Functions

Python provides a mechanism to document functions. This is text (possibly multi line, unlike a comment) between triple quotes, called a docstring. (Three single quotes or three double double quotes.)

```python
def xor(x, y):
    ''' computes xor of two Boolean variables, x and y '''
    return (x and (not y)) or ((not x) and y)
```

Everything between the start and end of the triple quotes is part of a single string, including carriage returns and other quote characters. You can use triple quotes anywhere, but they are most often used when defining docstrings.

When typing `help` with the function name (in parenthesis), the function’s docstring is printed.

```python
>>> help(xor)
Help on function xor in module __main__:

xor(x, y)
    computes xor of two Boolean variables, x and y
```

(Some of these explanations are taken from *dive into Python, section 2.3.*)
def xor(x, y):
    return (x and (not y)) or ((not x) and y)

The keyword `def` indicates the beginning of a function definition. The function name, in our case `xor`, follows. Then, in parenthesis, are the formal parameters `(x, y` in our case). There could be zero or more formal parameters. This ends in a colon, indicating the beginning of the function body. Following the colon, the body is, as usual, indented by a tab. The value following the `return` keyword is the returned value, of the function.
A function is called by specifying its name, followed by actual parameters, or arguments.

The number of actual parameters is identical to the number of formal parameters (there are some exceptions to this rule, which will be pointed out at a later stage).

The actual parameters are evaluated and passed to the body of the function to be executed.
The test4 Function: No Returned Value

def test4(n):
    """ tests divisibility by 4 and 8""

    if n % 4 != 0:
        print(n, "is not divisible by 4")
    else:
        if n % 8 == 0: # nested "if"
            print(n, "is divisible by 8")
        else: # nested "else"
            print(n, "is divisible by 4 but not by 8")

Important remark: This function contains no return statement.
Consequently, it does not return any value. See what happens when assigning
a=test4(21), then asking the interpreter for the value of a.

>>> a=test4(15)
15 is not divisible by 4
>>> a
>>>

Yet test4 does deliver information to the user, not the caller, via the
print command.
The test4 Function: No Returned Value

def test4(n):
    """ tests divisibility by 4 and 8""
    if n % 4 != 0:
        print(n,"is not divisible by 4")
    else:
        if n % 8 == 0: # nested "if"
            print(n,"is divisible by 8")
        else: # nested "else"
            print(n,"is divisible by 4 but not by 8")

Important remark: This function contains no return statement. Consequently, it does not return any value. See what happens when assigning a=test4(21), then asking the interpreter for the value of a.

>>> a=test4(15)
15 is not divisible by 4
>>> a

Yet test4 does deliver information to the user, not the caller, via the print command.
Palindromes

A palindrome is a string \( w \) satisfying \( w = w^R \) (the string and its reverse are the same).

- “Madam I’m Adam”
- “Dennis and Edna sinned”
- “Red rum, sir, is murder”
- “Able was I ere I saw Elba”
- “In girum imus nocte et consumimur igni” (Latin: “we go into the circle by night, we are consumed by fire”).
- “νιψον ανοµηµατα µη µοναν οψιν” (Nipson anomemata me monan opsin. Greek: “Wash [the] sins not only [the] face”. Inscribed at the baptismal font in the basilica of St. Sophia, Constantinople).

- And yes, we have cheated a bit by ignoring spaces as well as lower/upper case...
- Palindromes appear in nature. For example as DNA restriction sites – short genomic strings over \( \{A, C, T, G\} \), being cut by (naturally occurring) restriction enzymes.
Palindromes in Hebrew

פלינדרומים או מילה מתהפכת

ויקיפדיה:

•لد כתוב בתוך דלי
•נתנו תואר לאברום, Maher באלרואות נטן
•כלול לא בשיאו רק רמאי שבא למלוך
•דעו מאברכים כי לא בוש אבוש, שאב אשוש אלאיכם כי בא מעדו

כתבה ב"הארץ" מ 28/2/2013 על הפינדרומים האורkeh ביוותר
בעבריתشرفוסיםאנזורונהעלידי,نظמןعقوبات

http://www.haaretz.co.il/news/education/1.1940711
You will see palindromes again in the Computational Models course, where you will learn that finite automata cannot recognize them, while push down automata can.

However, we are not there yet, and all these automata will remain unknown to us for a while. What we want now is to write down a program that on input word, a string, checks if word is a palindrome. If it is, the program should return True. If not, it should return False.

How should we go about writing such program? Suppose word is of length $\ell$. The natural thing to do is to go over every index, $i$, in the interval 0 to $\ell - 1$, and check if $\text{word}[i] == \text{word}[\ell - 1 - i]$. 
How should we go about writing such program? Suppose \texttt{word} is of length $\ell$. The natural thing to do is to go over every index, $i$, in the range $0$ to $\ell - 1$, and check if $\texttt{word}[i] \neq \texttt{word}[\ell-(i+1)]$ (why +1?). If no mismatch is found, \texttt{word} is indeed a palindrome.

```python
def is_palindrome(word):
    ''' checks if word is a palindrome '''
    l=len(word)
    for i in range(len(word)):
        if word[i]!=word[-(i+1)]: # mismatch, no palindrome
            return False
    return True # matches all the way, a palindrome
```
def is_palindrome(word):
    ''' checks if word is a palindrome '''
    l = len(word)
    for i in range(len(word)):
        if word[i] != word[-(i+1)]:  # mismatch, no palindrome
            return False
    return True  # matches all the way, a palindrome

Let us run the palindrome checking function on several inputs:

>>> is_palindrome("998")
False
>>> is_palindrome("99899")
True
>>> is_palindrome(99899)
Traceback (most recent call last):
  File "<pyshell#55>" , line 1 , in <module>
      is_palindrome(99899)
  File "/Users/admin/Documents/InttroCS2011/Code/Intro3/palindrome.py", line 12, in is_palindrome
      l = len(word)
TypeError: object of type 'int' has no len()
def is_palindrome(word):
    ''' checks if word is a palindrome '''
    l=len(word)
    for i in range(1, len(word)+1):
        if word[i-1]!=word[-i]: # mismatch, no palindrome
            return False
        return (True) # matches all the way, a palindrome

As planned, we iterate over every index, \( i \), checking if \( \text{word}[i] \neq \text{word}[l-1-i] \). As soon as an inequality is found, we conclude this is not a palindrome and return False (which also aborts the function’s remaining execution). If all checks found equality, we conclude this is a palindrome.
Run Time Analysis for Identifying Palindromes

Suppose our input, \texttt{word}, is a string of length \( n \). This means it is composed of \( n \) characters.

How many operations will the function \texttt{is_palindrome} take to execute on \texttt{word}?

The answer to this depends to a large extent on \texttt{word} itself.
Run Time Analysis for Identifying Palindromes

Suppose our input, \texttt{word}, is a string of length \( n \). This means it is composed of \( n \) characters.

How many operations will the function \texttt{is_palindrome} take to execute on \texttt{word}?

The answer to this depends to a large extent on \texttt{word} itself. In the best case (from runtime point of view), \texttt{word[0]} and \texttt{word[-1]} are not equal. In this case, the execution will terminate after one comparison.

In the worst case (again, from runtime point of view), \texttt{word} is a palindrome. In this case, the execution will terminate after \( n \) comparisons.

You may wonder what will the average case looks like. To answer it, we should know something about the distribution of inputs. We will not tackle this question right now.
def is_palindrome_print(word):
    ''' checks if word is a palindrome; print diagnostic values '''
    l=len(word)
    for i in range(len(word)):
        print(i,word[i],word[-(i+1)])
        if word[i]!=word[-(i+1)]: # mismatch, no palindrome
            return False
    return True # matches all the way, a palindrome

If you have difficulties following what is going on inside the loop, it may be helpful to print intermediate results. Here, in every iteration we print its "index", i, and the two relevant string characters.

>>> is_palindrome_print("001200")
0 0 0
1 0 0
2 1 2
False
>>> is_palindrome_print("0110")
0 0 0
1 1 1
2 1 1
3 0 0
True
Improved Code for Identifying Palindromes

```python
def is_palindrome2(word):
    ''' checks if word is a palindrome '''
    return (word == word[::-1]) # does word equal its reverse
```

This code is definitely slimmer and more elegant than the previous one.

Is it also better performance wise?

We’ll leave this for you to ponder. Think both of the best case and worst case.
This Slide was Intentionally Left Blank
Identity, equality, or lack thereof, are of course central issues in society and politics. Struggle over them has shaped and defined nations and societies, as we witness again recently in neighboring countries, here in Israel, and even next to Wall Street.

“The struggle for identity and equality fought by everyday people”, pic taken from www.graydogsbooks.com

These larger issues are, however, out of the scope of our course. We will deal with them only in the context of Python objects.
Equality and Identity in Python

As we already saw, Python can check equality of an integer and a float.

```python
>>> 1==1.0
True
```

Python’s interpreter *coerces* the integer (1) to a float, and then checks equality of the two values. In this case, the two values are indeed equal, so it returns `True`.

But are these two objects (numbers) *identical*? Let us ask Python first:
Equality and Identity in Python

As we already saw, Python can check equality of an integer and a float.

```python
>>> 1==1.0
True
```

Python’s interpreter coerces the integer (1) to a float, and then checks equality of the two values. In this case, the two values are indeed equal, so it returns True.

But are these two objects (numbers) identical? Let us ask Python first:

```python
>>> 1 is 1.0
False
>>> 1 is not 1.0
True
```

These identity operators is and is not examine if the two objects refer to are the same object in memory. As we saw above, identity is a stricter relation than equality (identity implies equality, but not vice versa).
We can assign values to variables, then check for identity. The identity operator does not care about the variable’s name. It does check identity of the objects referred to by the variables’ names.

```python
>>> x=2
>>> y=2.0
>>> z=4-2
>>> t=x

>>> x is y
False
>>> x is z
True
>>> x is t
True
```
More Equality and Identity in Python

```python
>>> x=2
>>> y=2.0
>>> z=4-2
>>> t=x

>>> x is y
False
>>> x is z
True
>>> x is t
True
```

Consider the assignment to `x` and `z` above. After evaluation of the expressions on the right hand sides of the assignments, both objects have the value 2 (an integer).

`x is z` returned True, so can conclude that the two variables refer to the same object in memory. This is very different from other programming languages, where the variable acts as a “slot in memory”, and different identifiers refer to different memory locations.

We will get back to Python’s memory model in a few sides.
Lists are Mutable

In lecture 2, we saw that Python’s list is an ordered sequence of elements. Furthermore, list elements have indices, enabling direct (aka “random”) access. We now ask if lists elements can not just be individually accessed, but also be individually assigned?

```python
>>> list3 = [1,2,3]
>>> list3[2]
3
>>> list3[2] = "Agama stellio"
>>> list3
[1,2,"Agama stellio"]
```
The assignment `list3[2] = "Agama stellio"`, has mutated (changed) the list. How many of you are surprised?
Lists are Mutable

In lecture 2, we saw that Python’s list is an ordered sequence of elements. Furthermore, list elements have indices, enabling direct (aka “random”) access.

We now ask if lists elements can not just be individually accessed, but also be individually assigned?

```python
>>> list3 = [1,2,3]
>>> list3[2]
3
>>> list3[2] = "Agama stellio"
>>> list3
[1,2,‘Agama stellio’]

The assignment `list3[2] = "Agama stellio"`, has mutated (changed) the list.

How many of you are surprised?
Who is this ”Agama stellio”, and what was its contribution to Israeli literary heritage (list two cases or more)?
A Small Diversion

Who is this "Agama stellio", and what was its contribution to Israeli literary heritage (list two cases or more)?
Strings are not Mutable

Like lists, strings are also indexed (the official term is “subscriptable”): Individual characters can be directly accessed, using their index. Consider our favorite string, for example:

```python
>>> species = "Agama stellio"
>>> species[0]
'A'
>>> species[1]
'g'
>>> species[5]
'

However, unlike lists, we cannot mutate strings. Trying to do so will result in an error.
```
Strings are **not** Mutable

Like lists, strings are also indexed (the official term is “subscriptable”): Individual characters can be directly accessed, using their index. Consider our favorite string, for example:

```plaintext
>>> species = "Agama stellio"
>>> species[0]
'A'
>>> species[1]
'g'
>>> species[5]
' '
```

However, unlike lists, we **cannot** mutate strings. Trying to do so will result in an error.

```plaintext
>>> species[2]="t"
Traceback (most recent call last):
  File "<pyshell#4>" , line 1 , in <module>
    species[2]="t"
TypeError: 'str' object does not support item assignment
```
Numbers are Surely not Mutable

Unlike lists or strings, numbers cannot be indexed (namely they are not “subscriptable”): We cannot directly access their bits, nor can we modify them.

Consequently, numbers are not mutable.
Assignments and Reassignments to Integer Variables

We start with a few assignments to integer variables, of `<class 'int'>`:

```python
>>> n=10
>>> m=n
>>> n=11
>>> n
11
>>> m
10
```

So far, no surprises (I hope).

Essentially the same behavior will occur with numbers of type `float`. 
Assignments to String Variables

And now, a few assignments to string variables, `<class 'str'>:

```python
>>> word1="PhD TBF"
>>> word2=word1
>>> word1="Gee Whiz"
>>> word2
'PhD TBF'
>>> word1
'Gee Whiz'
```

No surprises here either.
Assignments to List Variables

And a few assignments to list variables, `<class 'list'>`:

```python
>>> list1=[1,2,3]
>>> list2=list1
>>> list1=[6,7,8,9]
>>> list2
[1,2,3]
>>> list1
[6,7,8,9]
```

Still, no surprises (you may start wondering if this discussion is leading anywhere).
Assignments to List Variables, take 2

A few assignments to components of list variables, `<class 'list'>`:

```python
>>> list1=[1,2,3]
>>> list2=list1
>>> list1[0]=97  # mutating list1
>>> list1
[97,2,3]  # this is to be expected
>>> list2  # this is to be expected
[97,2,3]  # list2 also mutated!!!
```

What the %$⊗#$ is happening here?
Assignments to List Variables, take 2

A few assignments to components of list variables, `<class 'list'>`:

```python
>>> list1=[1,2,3]
>>> list2=list1
>>> list1[0]=97  # mutating list1
>>> list1
[97,2,3]  # this is to be expected
>>> list2
[97,2,3]  # list2 also mutated!!!
```

What the %$⨂#$ is happening here?

The assignment `list1=[1,2,3]` creates a list object, `[1,2,3]`, and a reference from the variable name, `list1`, to this object. The assignment `list2=list1` does not create a new object. It just creates a new variable name, `list2`, which now refers to the same object.

When we mutate, `list1[0]=97`, we do not change these references. Thus, displaying `list2` produces `[97,2,3]`. 
Python’s id Function

Python’s interpreter has a built in function, id, which returns the “identity” of an object. This is an integer which is guaranteed to be unique and constant for this object during its lifetime. Using a different terminology, id is the address of the object in memory.

Clarification: id(object1) == id(object1) if and only if object1 is object2.

Warning: For optimization reasons, two objects with non-overlapping lifetimes may have the same id value. Furthermore, in two different executions, the same object may be assigned different id. And obviously this is platform dependent.

When using id, we recommend using the hexadecimal (base 16) representation of the outcome. This obviously is equivalent, yet often more transparent than the decimal representation.
Python’s Memory Model and the \texttt{id} Function

Variables’ names (identifiers) in Python correspond to an object in memory. As a result of a new assignment to the same variable identifier, a new object (in a different location in memory) is referred to by the same identifier. The \texttt{id} allows us to “probe” memory locations directly, and even compare them over time (something that cannot be done with the \texttt{is} operator).

```python
>>> x=1
>>> id(x)
1494016
>>> hex(id(x))
'0x16cc00'
>>> x=2
>>> hex(id(x))  # new object, new memory location
'0x16cc10'  # exactly 16 away from previous
>>> x="no worries, mate"
>>> hex(id(x))
'0x15b0aa0'  # far away from previous
```
Reassigning Immutable Objects

```python
>>> x=1
>>> id(x)
1494016
>>> hex(id(x))
'0x16cc00'
>>> x=2
>>> hex(id(x))  # new object, new memory location
'0x16cc10'  # exactly 16 away from previous
>>> x="no worries, mate"
>>> hex(id(x))
'0x15b0aa0'  # far away from previous
>>> x=1
>>> hex(id(x))
'0x16cc00'  # back to original memory location
>>> y=1
>>> hex(id(y))
'0x16cc00'  # same as last location of x
```

From the last two examples we conclude that the address of “small” immutable objects, like small integers, is determined by their value, and seems independent of execution history.
For “large” immutable objects, e.g. large numbers, the (address of the) object is typically not uniquely determined by the value. For example, with $2^{100} + 1$

```python
>>> x = 2**200 + 1
>>> y = 2**200 + 1
>>> x == y
True
>>> x is y
False
# we will now probe the exact addresses
>>> hex(id(x))
'0x170d098'
>>> hex(id(y))
'0x170d048'
```
Reassigning Immutable Objects, cont. cont.

In fact, “large” immutable objects, e.g. large numbers, need not be all that large to behave in this unexpected manner.

>>> 257 is 257
True
>>> x=257
>>> y=257
>>> x is y
False
>>> hex(id(257))
'0x15c32d0'
>>> hex(id(257))
'0x15c3630'

# two distinct incarnations of 257
In fact, “large” immutable objects, e.g. large numbers, need not be all that large to behave in this unexpected manner.

```python
>>> 257  is  257
True
>>> x=257
>>> y=257
>>> x  is  y
False
>>> hex(id(257))
'0x15c32d0'
>>> hex(id(257))
'0x15c3630'
```

# two distinct incarnations of 257

Confused? You should be!

You may still retain some faith in human kind in general, and in Guido van Rossum in particular, by knowing that positive integers up to 256, a few negative integers, and single characters, do have a single, preassigned location in memory.
For mutable objects, we saw that some “components” of the object can subsequently be changed. This does not change the memory location of the object. For example, mutating a list.

```python
>>> list1=[1,2,3]
>>> hex(id(list1))
'0x290deb8'
>>> list1[0]=97 # mutating list1. current memory location NOT changed
>>> list1
[97,2,3] # mutated indeed
>>> hex(id(list1))
'0x290deb8' # object memory location UNCHANGED
```
For mutable objects, we saw that some “components” of the object can subsequently be changed. This does not change the memory location of the object. For example, mutating a list.

```python
>>> list1=[1,2,3]
>>> hex(id(list1))
'0x290deb8'
>>> list1[0]=97  # mutating list1. current memory location NOT changed
>>> list1
[97,2,3]  # mutated indeed
>>> hex(id(list1))
'0x290deb8'  # object memory location UNCHANGED
```

# and now, let’s just repeat the first assignment
```python
>>> list1=[1,2,3]
>>> hex(id(list1))
'0x290d968'  # NEW object, new memory location
```

For mutable objects, like lists, a new assignment to the same identifier with identical value creates a new object with a new address.
First, let us examine lists with identical values yet different addresses.

```python
>>> list1=[1,2,3]
>>> hex(id(list1))
'0x15e9b48'
>>> list2=list1
>>> hex(id(list2))
'0x15e9b48'  # same same
>>> list3=[1,2,3]
>>> hex(id(list3))
'0x15e9cb0'  # but different
```
One More Look at Mutable Object

First, let us examine lists with identical values yet different addresses.

```python
>>> list1=[1,2,3]
>>> hex(id(list1))
'0x15e9b48'
>>> list2=list1
>>> hex(id(list2))
'0x15e9b48'  # same same
>>> list3=[1,2,3]
>>> hex(id(list3))
'0x15e9cb0'  # but different
```

Now let us see what happens with the components of these lists.

```python
>>> list1[0] is list3[0]
True
>>> hex(id(list1[0]))
'0x16cc00'  # looks familiar?
>>> hex(id(list3[0]))
'0x16cc00'  # same as previous
```

What graphic images of these lists in memory follow?
A Graphical View: The Balloons Model
Deleting an Object

So far, we saw that an assignment adds a variable name and associates an object with it.

It is also possible to delete a variable. After deletion, the variable no longer exists, and referring to it in an expression yields an error.

```python
>>> x=10
>>> x
10
>>> del x
>>> x
raceback (most recent call last):
    File "<pyshell#125>", line 1, in <module>
    x
NameError: name 'x' is not defined

>>> s = 200
>>> t = s
>>> del s # s is gone
>>> t # t is still alive and kicking
200
```