State and Side Effects

Rob Hackman

Fall 2025

University of Alberta

Table of Contents

Considering identifiers

We have not spent much time investigating how Python keeps track of all the identifiers in our program, and what values they store.

We will now begin to consider this, and devise our own system for modelling the behaviour of Python's identifiers.

Scope and identifier access

Consider the following Python definitions:

```
def foo(x):
  fooVar = x*3
  return bar(x*15)
```

```
def bar(z):
  result = z*x + fooVar
  return result
```

We import these functions and evaluate the following expression:

```
foo(3)
```

The above expression causes an error shown below:

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 3, in foo
  File "<stdin>", line 2, in bar
NameError: name 'x' is not defined
```

Scope and identifier access

Consider the following Python definitions:

```
def foo(x):
  fooVar = x*3
  return bar(x*15)
```

```
def bar(z):
  result = z*x + fooVar
  return result
```

Scope and identifier access

Consider the following Python definitions:

```
def foo(x):
  fooVar = x*3
  return bar(x*15)
```

```
def bar(z):
  result = z*x + fooVar
  return result
```

The problem is that each function definition creates its own *scope*. Identifiers that are *defined* within a scope are said to be *local* to that scope. This is why multiple functions could use the same name for identifiers with no conflict since they were the same identifier but defined within different scopes.

Identifiers must be defined before their value is read. Identifiers defined within a scope are accessible only within that scope *or* scopes that are nested within that scope.

Scope and identifier access

Consider the following Python definitions:

```
def foo(x):
  fooVar = x*3
  return bar(x*15)
```

```
def bar(z):
  result = z*x + fooVar
  return result
```

So when are identifiers defined within a scope? In Python identifier definition happens **implicitly** in one of two cases:

- When a function is defined with parameters, the parameter names are identifiers defined within the scope of that function
- The first time an identifier is assigned to in a scope the identifier is defined within that scope

Some languages require explicit definition of identifiers to avoid confusion.

Understanding scope

We said on the previous slides that "Identifiers defined within a scope are accessible only within that scope *or* scopes that are nested within that scope". So how can a scope be nested within another scope?

Understanding scope

We said on the previous slides that "Identifiers defined within a scope are accessible only within that scope *or* scopes that are nested within that scope". So how can a scope be nested within another scope?

Since each function definition creates its own scope we have already seen many instances where scopes are defined within scopes. Whenever a `lambda` function is written within the body for some function F, or whenever a nested function is defined within the body of some function F, a scope is being created within the scope of function F.

Nested functions and scoping

Consider if our definition for `bar` and `foo` was instead rewritten so that `bar` was an inner function within `foo`:

```
def foo(x):
  def bar(z):
    result = z*x + fooVar
    return result
  fooVar = x*3
  return bar(x*15)
```

Now when we evaluate the expression `foo(3)` we don't get an error and simply get the result `144`. So in this case the function `bar` *was* able to access the parameter of `foo` `x`, and also the variable `fooVar` that was defined within the scope of `foo`.

Identifiers outside of function definitions

So far we have only ever defined identifiers within the scope of a function.

Identifiers outside of function definitions

So far we have only ever defined identifiers within the scope of a function.

Except that's not strictly true! We know that function names are just identifiers, and since we know functions are first-class values we know that function names are just identifiers that are defined to be a value that is a function as opposed to something like an integer.

Identifiers outside of function definitions

So far we have only ever defined identifiers within the scope of a function.

Except that's not strictly true! We know that function names are just identifiers, and since we know functions are first-class values we know that function names are just identifiers that are defined to be a value that is a function as opposed to something like an integer.

So what scope do the identifiers we define as non-nested functions exist in?

Where are non-nested functions defined?

If we consider what we've been told about scopes and the behaviour we already experienced we can derive a truth that must exist. We've seen that functions can call other functions, for example recall `ascendList`:

```
def append(elem, l):
  if isEmpty(l):
    return cons(elem, empty())
  ror = append(elem, rest(l))
  return cons(first(l), ror)

def ascendList(n):
  if n == 0:
    return cons(0, empty())
  ror = ascendList(n-1)
  return append(n, ror)
```

Where are non-nested functions defined?

If we consider what we've been told about scopes and the behaviour we already experienced we can derive a truth that must exist. We've seen that functions can call other functions, for example recall `ascendList`:

```
def append(elem, l):
  if isEmpty(l):
    return cons(elem, empty())
  ror = append(elem, rest(l))
  return cons(first(l), ror)

def ascendList(n):
  if n == 0:
    return cons(0, empty())
  ror = ascendList(n-1)
  return append(n, ror)
```

- `ascendList` could refer to the identifer append even though append was not defined within the scope of `ascendList`

Where are non-nested functions defined?

If we consider what we've been told about scopes and the behaviour we already experienced we can derive a truth that must exist. We've seen that functions can call other functions, for example recall `ascendList`:

```
def append(elem, l):
  if isEmpty(l):
    return cons(elem, empty())
  ror = append(elem, rest(l))
  return cons(first(l), ror)

def ascendList(n):
  if n == 0:
    return cons(0, empty())
  ror = ascendList(n-1)
  return append(n, ror)
```

- `ascendList` could refer to the identifer append even though append was not defined within the scope of `ascendList`
- Given what we've been told the implication is that append *must* be defined within the scope that the `ascendLists` scope is nested within.

Where are non-nested functions defined?

If we consider what we've been told about scopes and the behaviour we already experienced we can derive a truth that must exist. We've seen that functions can call other functions, for example recall `ascendList`:

```
def append(elem, l):
  if isEmpty(l):
    return cons(elem, empty())
  ror = append(elem, rest(l))
  return cons(first(l), ror)

def ascendList(n):
  if n == 0:
    return cons(0, empty())
  ror = ascendList(n-1)
  return append(n, ror)
```

So then the question is: in which scope are append and `ascendList` defined?

The global scope

In every Python program there is a scope called the *global* scope. The global scope is the scope in which every identifier that is defined not within a function body is defined — this includes the definitions of non-nested functions, since function definitions are defining an identifier (the function name).

Since the global scope contains all definitions that are not defined within a function body the implication is that *all* scopes other than the global scope are nested within the global scope. Per our rule that means that if something is defined in the global scope then it is accessible within any scope.

Global constants

Consider the following Python program:

```
def circleArea(r):
  pi = 3.14
  return pi*r**2

def circleDiameter(r):
  pi = 3.14
  return pi*2*r
```

Here both functions `circleArea` and `circleDiameter` need to use our approximation of π.

Global constants

Consider the following Python program:

```
def circleArea(r):
  pi = 3.14
  return pi*r**2

def circleDiameter(r):
  pi = 3.14
  return pi*2*r
```

Here both functions `circleArea` and `circleDiameter` need to use our approximation of π.

What if later on when writing our program we realize we need more precision on our calculations, and thus needed to use a closer approximation of π?

Global constants

Consider the following Python program:

```
def circleArea(r):
  pi = 3.14
  return pi*r**2

def circleDiameter(r):
  pi = 3.14
  return pi*2*r
```

As it stands currently, we would have to go through our entire program and find every place we used our approximation of `pi` And update it, hoping we don't miss any.

Shown here are only two places, but there could be *many* places in our program we used this approximation of π which possible two problems.

Global constants

Consider the following Python program:

```
def circleArea(r):
  pi = 3.14159
  return pi*r**2

def circleDiameter(r):
  pi = 3.1459
  return pi*2*r
```

- First problem — we are lazy. We don't want to have to search through our whole program updating our approximation of pi everywhere it is used.

Global constants

Consider the following Python program:

```
def circleArea(r):
  pi = 3.14159
  return pi*r**2

def circleDiameter(r):
  pi = 3.1459
  return pi*2*r
```

- First problem — we are lazy. We don't want to have to search through our whole program updating our approximation of pi everywhere it is used.
- Second problem — it is error-prone! The more places we must update our approximation the more chances that we either miss one or make a mistake when updating one. Look above... there is a mistake!

Global constants

Consider the following Python program:

```
def circleArea(r):
  pi = 3.14159
  return pi*r**2

def circleDiameter(r):
  pi = 3.1459
  return pi*2*r
```

What then is a better solution?

Global constants

Consider the following Python program:

```
def circleArea(r):
  pi = 3.14159
  return pi*r**2

def circleDiameter(r):
  pi = 3.1459
  return pi*2*r
```

What then is a better solution?

Define our approximation of π only *once* and have everywhere in our code use that one definition.

Global constants

Consider the following Python program:

```
pi = 3.14159
def circleArea(r):
  return pi*r**2

def circleDiameter(r):
  return pi*2*r
```

If we define the identifier `pi` in the global scope then every function can access that same definition, and we now only have one place where the definition must be updated if we want to change it.

Such an identifier is called a *global constant*. Global, because it resides in the global scope, and constant because we never intend to change its value during the execution of our program.

Global constants

While anything can be defined within the global scope, it is best practice to only use the global scope for values you intend to be constant and never change.

Global *variables*, that is global identifiers where we intend to change the value of the variable over the execution of our program, are considered by many to be bad practice as they make it harder to reason about a program and what it does!

If a function refers to the value of a global variable during its execution then that function is no longer said to be a *pure* function. So far all the functions we've written together have been pure functions, when functions become impure it becomes much harder to speak about what a program does and understand a program.

A note on global variables

As mentioned the use of global identifiers that vary in value is considered a bad practice as it makes programs more complicated and hard to understand.

As such, in this course global variables are disallowed, and if any global data is used it *must* be a constant.

Pure functions

What is a *pure* function? A pure function is very simply a function that meets two conditions:

- The function always produces the same result whenever called on the same set of arguments. i.e. for some function f, some collection of arguments A and some value v if $f(A)- > v$ then *everytime* f is called on A it produces v.
- The function has no *side effects*. We have not learned how to incur any side effects, more on this later.

Functions as we understand them mathematically are all pure functions. A mathematician may argue it would be better nomenclature to use the word functions *only* for pure functions, and to call impure functions something else altogether!

Global variables and function impurity

Allowing global identifiers to vary in value over the run of our program is one way in which we may define impure functions, which become harder to reason about. Consider the following Python program:

```
base = 10
def digitCount(n):
  if n < base:
    return 1
  return 1 + digitCount(n//base)

count1 = digitCount(5321) # 4
base = 8
count2 = digitCount(5321) # 5
```

Global variables and function impurity

```python
base = 10
def digitCount(n):
  if n < base:
    return 1
  return 1 + digitCount(n//base)

count1 = digitCount(5321) # 4
base = 8
count2 = digitCount(5321) # 5
```

The first time `digitCount` was called on the argument 5321 it produced the answer four. However, the next time `digitCount` was called on 5321 it produced a different result — five! So `digitCount` is not a pure function since it does not always produce the same result when called on the same argument.

Global variables and function impurity

```
base = 10
def digitCount(n):
  if n < base:
    return 1
  return 1 + digitCount(n//base)

count1 = digitCount(5321) # 4
base = 8
count2 = digitCount(5321) # 5
```

The impurity of `digitCount` comes from the fact that its execution relies on the *state* of the global variable `base` which may change.

Global variables and function impurity

```
base = 10
def digitCount(n):
  if n < base:
    return 1
  return 1 + digitCount(n//base)

count1 = digitCount(5321) # 4
base = 8
count2 = digitCount(5321) # 5
```

The reason programs with impure functions become harder to reason about is because now whenever seeing a function call on a particular value it is not enough to know what the function does to determine what the result will be, you must also know the answer to the question "what is the state of my program at the particular point in time that I am calling that function".

A note on global identifiers

With our current knowledge global identifiers can only become global variables if we assign them more than once in the global scope. Assigning to an identifier that shares the same name as a globally defined identifier will *not* modify that global identifier. Consider:

```
someNum = 0
def update(n):
  previous = someNum
  someNum = previous + n
  return someNum
update(5) -> 5
update(5) -> 5
```

A note on global identifiers

```
someNum = 0
def update(n):
  previous = someNum
  someNum = previous + n
  return someNum
update(5) -> 5
update(5) -> 5
```

Why does the second call to `update` not return ten? We might think the following happened:

A note on global identifiers

```python
someNum = 0
def update(n):
  previous = someNum
  someNum = previous + n
  return someNum
update(5) -> 5
update(5) -> 5
```

Why does the second call to `update` not return ten? We might think the following happened:

1. First call to `update(5)` sets `previous` to `someNum`, which is zero

A note on global identifiers

```
someNum = 0
def update(n):
  previous = someNum
  someNum = previous + n
  return someNum
update(5) -> 5
update(5) -> 5
```

Why does the second call to `update` not return ten? We might think the following happened:

1. First call to `update(5)` sets `previous` to `someNum`, which is zero
2. It then sets `someNum` equal to `previous` + `n` which is $0 + 5$ so `someNum` becomes five, and it returns `someNum` which is five.

A note on global identifiers

```
someNum = 0
def update(n):
  previous = someNum
  someNum = previous + n
  return someNum
update(5) -> 5
update(5) -> 5
```

Why does the second call to `update` not return ten? We might think the following happened:

1. First call to `update(5)` sets `previous` to `someNum`, which is zero
2. It then sets `someNum` equal to `previous` + `n` which is $0 + 5$ so `someNum` becomes five, and it returns `someNum` which is five.
3. The second call to `update(5)` sets `previous` to `someNum`, which is five

A note on global identifiers

```
someNum = 0
def update(n):
  previous = someNum
  someNum = previous + n
  return someNum
update(5) -> 5
update(5) -> 5
```

Why does the second call to `update` not return ten? We might think the following happened:

1. First call to `update(5)` sets `previous` to `someNum`, which is zero
2. It then sets `someNum` equal to `previous` + `n` which is $0 + 5$ so `someNum` becomes five, and it returns `someNum` which is five.
3. The second call to `update(5)` sets `previous` to `someNum`, which is five
4. It then sets `someNum` equal to `previous` + `n` which is $5 + 5$ so `someNum` becomes ten, and it returns `someNum` which is ten.

A note on global identifiers

```
someNum = 0
def update(n):
  previous = someNum
  someNum = previous + n
  return someNum
update(5) -> 5
update(5) -> 5
```

The previous series of events is incorrect though, because it relies on an incorrect assumption. The assumption specifically in these two steps:

2. It then sets `someNum` equal to `previous` + `n` which is $0 + 5$ so
3. The second call to `update(5)` sets `previous` to `someNum`, which is five `someNum` becomes five, and it returns `someNum` which is five.

It is the assumption that the global identifier `someNum` was updated to 5 in the first function call, so that when we read it in the second its value is 5 that is incorrect.

A note on global identifiers

```
someNum = 0
def update(n):
  previous = someNum
  someNum = previous + n
  return someNum
update(5) -> 5
update(5) -> 5
```

Recall what we learned about identifier definitions — specifically identifiers are implicitly defined in a scope when one of the two things occurs:

- When a function is defined with parameters, the parameter names are identifiers defined within the scope of that function
- **The first time an identifier is assigned to in a scope the identifier is defined within that scope**

A note on global identifiers

```
1  someNum = 0
2  def update(n):
3    previous = someNum
4    someNum = previous + n
5    return someNum
6  update(5) -> 5
7  update(5) -> 5
```

This means that on line 4 when our function executes `someNum = previous + n` it is *not* updating the global identifier `someNum` but rather defining a *new* identifier named `someNum` within the scope of `update`.

So the real order of events is the following.

A note on global identifiers

```
someNum = 0
def update(n):
  previous = someNum
  someNum = previous + n
  return someNum
update(5) -> 5
update(5) -> 5
```

Whenever `update` is called the following occurs:

A note on global identifiers

```
1  someNum = 0
2  def update(n):
3    previous = someNum
4    someNum = previous + n
5    return someNum
6  update(5) -> 5
7  update(5) -> 5
```

Whenever `update` is called the following occurs:

- Line 3 defines a local variable `previous` by reading the value of the global identifier `someNum`, which is zero.

A note on global identifiers

```
1  someNum = 0
2  def update(n):
3    previous = someNum
4    someNum = previous + n
5    return someNum
6  update(5) -> 5
7  update(5) -> 5
```

Whenever `update` is called the following occurs:

- Line 3 defines a local variable `previous` by reading the value of the global identifier `someNum`, which is zero.
- Line 4 defines a local variable `someNum` by setting it equal to the value of `previous` and `n`. Assuming the global `someNum` is a constant that will always set the local `someNum` to $0 + n$.

A note on global identifiers

```
1 someNum = 0
2 def update(n):
3   previous = someNum
4   someNum = previous + n
5   return someNum
6 update(5) -> 5
7 update(5) -> 5
```

Whenever `update` is called the following occurs:

- Line 3 defines a local variable `previous` by reading the value of the global identifier `someNum`, which is zero.
- Line 4 defines a local variable `someNum` by setting it equal to the value of `previous` and `n`. Assuming the global `someNum` is a constant that will always set the local `someNum` to $0 + n$.
- Line 5 returns the value of `someNum`, since a locally defined variable named `someNum` is available that is the one used and the value returned is simply n.

Identifier shadowing

We have already noted that scopes can exist within other scopes, and that identifiers defined within a scope are accessible only within the scope they are defined *or* scopes that are contained within that scope.

Now we have seen, however, that one scope nested within another may have the *same* identifier defined — in that case whose version of that identifier is used?

In the case where an identifier is referenced that is defined in multiple scopes, the *closest* scope in which that identifier is accessible is always the one that is used.

Identifier shadowing example

```
id1 = "globalID1"
id2 = "globalID2"
def shadow():
  id1 = "shadowID1"
  def nestOne():
    id2 = "nestID2"
    return id1 + " - " + id2
  def nestTwo():
    return id1 + " - " + id2
  def nestThree(id1):
    return id1 + " - " + id2
  return (nestOne(), nestTwo(), nestThree("nestID1"))
```

The result of the constant function `shadow` is the tuple:

```
  ('shadowID1 - nestID2',
   'shadowID1 - globalID2',
   'nestID1 - globalID2')
```

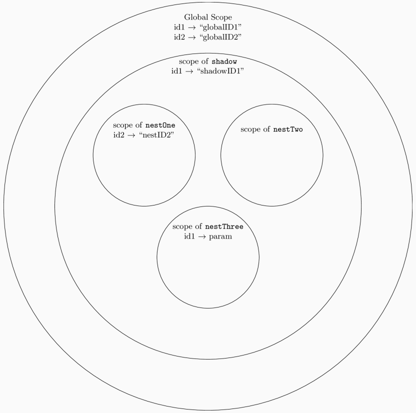
Identifier shadowing example

```
id1 = "globalID1"
id2 = "globalID2"
def shadow():
  id1 = "shadowID1"
  def nestOne():
    id2 = "nestID2"
    return id1 + " - " + id2
  def nestTwo():
    return id1 + " - " + id2
  def nestThree(id1):
    return id1 + " - " + id2
  return (nestOne(), nestTwo(), nestThree("nestID1"))
```

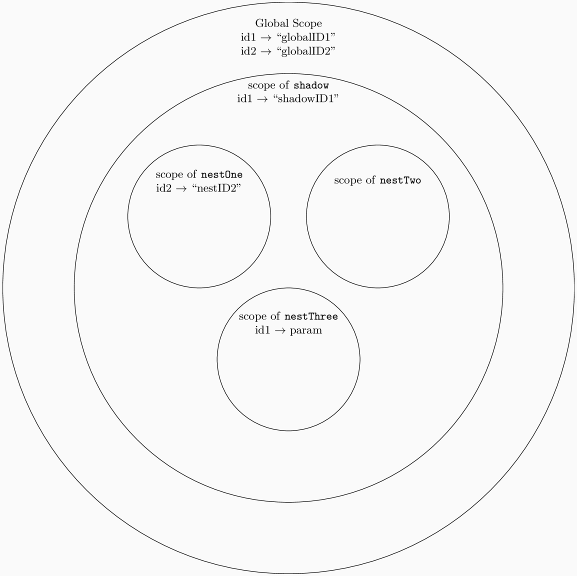
To understand this result we must understand our nested scopes and what is meant when we say the closest scope in which an identifier is accessible is the one that is used.

Envisioning scopes

We can envision each scope as a circle which has identifier definitions with in it, and may have other circles within it that are other scopes. For example the code above may be represented as follows:



Envisioning scopes



Envisioning scopes

Remembering two rules then, we can easily resolve the reason why we got each result for `nestOne()`, `nestTwo()`, and `nestThree("nestID1")`. The rules are as follows:

Envisioning scopes

Remembering two rules then, we can easily resolve the reason why we got each result for `nestOne()`, `nestTwo()`, and `nestThree("nestID1")`. The rules are as follows:

- Identifiers defined within a scope are accessible only within the scope they are defined in *or* scopes that are contained within that scope.

Envisioning scopes

Remembering two rules then, we can easily resolve the reason why we got each result for `nestOne()`, `nestTwo()`, and `nestThree("nestID1")`. The rules are as follows:

- Identifiers defined within a scope are accessible only within the scope they are defined in *or* scopes that are contained within that scope.
- When an identifier is referenced that is defined in multiple scopes, the *closest* scope in which that identifier is accessible is always the one that is used.

Envisioning scopes

So first let us resolve why `nestOne()` produced
`'shadowID1 - nestID2'`

- `nestOne` returns `id1 + " - " + id2`.
- Since the scope of `nestOne` is defined within the scope of `shadow`, which itself is defined within the global scope the code of `nestOne` can only read identifiers defined within one of those three scopes.
- When resolving `id1` the scope of `nestOne` does not have that identifier defined, and so it must look in its enclosing scopes. The identifier is defined in *both* the global scope and the scope of `shadow` — since `shadow` is the "closer" scope to `nestOne` the value from there is used.
- When resolved `id2` the scope of `nestOne` *does* have its own definition for `id2`, and so that value is used.

Envisioning scopes

Next, why does `nestTwo()` produce `'shadowID1 - globalID2'`

- `nestTwo` returns `id1 + " - " + id2`.
- Again `id1` is not defined in the scope of `nestTwo` so it must look to enclosing scopes. The definition of `id1` available in the scope of `shadow` is the closest definition so it is used.
- In `nestTwo's` scope the identifier of `id2` is not defined. Even though `id2` is defined in the scope of `nestOne` the scope of `nestTwo` *cannot* access that definition, because `nestTwo` is not contained within the scope of `nestOne`.
- As such, the resolution of `id2` in `nestTwo` looks in its enclosing scopes, the closest enclosing scope that has a definition for `id2` is the global scope, and so that value is used.

Envisioning scopes

Next, why does `nestThree("nestID1")` produce `'nestID1 - globalID2'`

- `nestThree` returns `id1 + " - " + id2`.
- This time `id1` *is* defined in the scope of `nestThree`, however it is defined to be the parameter. As such, the value to substitute for it must come from the argument this function was called on, which in this case is `"nestID1"`.
- Again, `id2` is not defined within the scope of `nestThree` so it must look in enclosing scopes. The closest definition in an enclosing scope is the global scopes definition of `id2`, so the value `"globalID2"` is used.

Identifier shadowing

When the same identifier is redefined within a nested scope and demonstrated earlier we refer to it as identifier shadowing.

There are ways in Python to stop assignment from defining an identifier, and rather refer to and reassign a copy from an enclosing scope such as the keywords `global` and `nonlocal` but we will not discuss them nor use them in this course but are now made aware of their existence for future learning after this course.

Table of Contents

Remembering our first Python function

Soon we will continue on learning about impurity in Python and how to model the state of our program when impure functions exist. However, it will be useful in discussing that topic to know about another Python type which is called the `NoneType`.

Recall now the very first function we tried to write in Python by translating a mathematical function

We noted that this function didn't work, in that when we called it we didn't see a value produced — however we also *didn't* get an error message from Python which indicates that the function does actually run. So what then does this function produce?

Remembering our first Python function

Soon we will continue on learning about impurity in Python and how to model the state of our program when impure functions exist. However, it will be useful in discussing that topic to know about another Python type which is called the `NoneType`.

Recall now the very first function we tried to write in Python by translating a mathematical function

$$f(x) \longrightarrow 3x + 2$$

We noted that this function didn't work, in that when we called it we didn't see a value produced — however we also *didn't* get an error message from Python which indicates that the function does actually run. So what then does this function produce?

Remembering our first Python function

Soon we will continue on learning about impurity in Python and how to model the state of our program when impure functions exist. However, it will be useful in discussing that topic to know about another Python type which is called the `NoneType`.

Recall now the very first function we tried to write in Python by translating a mathematical function

$$f(x) \longrightarrow 3x + 2$$

```
def f(x):
  3*x + 2
```

We noted that this function didn't work, in that when we called it we didn't see a value produced — however we also *didn't* get an error message from Python which indicates that the function does actually run. So what then does this function produce?

Functions without return statements

If a Python function does not reach a return statement before the end of its execution the functions must still return a value. If this happens the function will return a special value represented by the keyword `None` which has the type of `NoneType`

Similarly to how we learned that the `bool` type had only two possible values, the `NoneType` has only *one* possible value — `None`

The value `None` is used to represent the complete *absence* of a value. Such a type is useful when needing to indicate that *no* value exists.

So, our very first function `f` returns `None`.

Use case of `None`

While it may seem silly to have a value that very specifically represents the absence of a value it is actually extremely useful. Consider now our `findPath` function that we wrote to operate on a tree.

In the function `findPath` we returned the empty `LList` both when no path was possible and when the item we were looking for was stored at the root note, since then there were no steps to from that node to reach the value we were looking for.

This caused a problem when we tried to use our recursive result, we didn't know if the result was empty because our child node contained the value we were looking for or the result was empty because no path was found!

Revisiting `findPath`

We chose to have `findPath` return an empty `LList` when no path was found because we had no better option, but the path with no direction is still a path!

A better alternative is to have `findPath` return the empty `LList` only when the path to the value we're looking for is empty (i.e. the value we're looking for *is* the current node) and instead when *no* path exists instead return `None` to represent the *absence* of a path!

Since `findPath` now returns a different type of value if a path exists or not we must update its function specification.

Updating `findPath`

```python
def findPath(bt, val):
  '''
  findPath returns a LList of the directions left or right
           that one must follow from the root of bt to
           reach the value val. If the value does not
           exist in the tree then it returns None
  bt      - A BinaryTree of X
  val     - X
  returns - anyof(List of str, None)
  '''
```

Having `findPath` return different values when the path is not found versus when the path is empty greatly simplifies our code as we no longer have to check *why* our recursive result was empty, the recursive result itself gives us all the information we need.

Updating `findPath`

```
def findPath(bt, val):
  if bt == ():
    return None
  if bt[0] == val:
    return empty()
  leftPath = findPath(bt[1], val)
  if leftPath != None:
    return cons("left", leftPath)
  rightPath = findPath(bt[2], val)
  if rightPath != None:
    return cons("right", rightPath)
  return None
```

What can one do with `None`

Since the value `None` is meant to represent the complete absence of a value you cannot compute many expressions involving it.

What can one do with `None`

Since the value `None` is meant to represent the complete absence of a value you cannot compute many expressions involving it.

In fact, the only thing you can do with `None` is compare it to another value for equality or inequality!

What can one do with `None`

Since the value `None` is meant to represent the complete absence of a value you cannot compute many expressions involving it.

In fact, the only thing you can do with `None` is compare it to another value for equality or inequality!

Since our function `findPath` may not actually be able to find a path the value `None` was very helpful. There are other use cases of `None` than just functions that *cannot* compute a result. `None` is used as the return value for functions that *don't* produce a result — but what use is a function that doesn't produce a result though?

Table of Contents

Implications of impurity

So far in this course we have only considered pure functions, which has made understanding our code very simple.

Implications of impurity

So far in this course we have only considered pure functions, which has made understanding our code very simple.

When we have a pure function f and are confronted with the question “What does $f(x)$ produce?” the only information needed to answer the question is the actual code that comprises f.

Implications of impurity

So far in this course we have only considered pure functions, which has made understanding our code very simple.

When we have a pure function f and are confronted with the question "What does $f(x)$ produce?" the only information needed to answer the question is the actual code that comprises f.

Upon seeing our first impure function `digitCount` defined earlier, we observed the point that in order to answer the question "What does $digitCount(x)''$ produce it was not enough to know the code that comprises *digitCount*, we also need to know the *state* of our program at the particular point in time that `digitCount` is called.

Considering `digitCount`

Consider our impure function `digitCount`

```
def digitCount(n):
  if n < base:
    return 1
  return 1 + digitCount(n//base)
```

What is the result of the expression `digitCount(7912)`?

Considering `digitCount`

```
def digitCount(n):
  if n < base:
    return 1
  return 1 + digitCount(n//base)
```

What is the result of the expression `digitCount(7912)`?

We don't have enough information to answer that question! So, when impurity is introduced understanding our code gets harder to do. What information is needed to answer the question above?

Considering `digitCount`

```
def digitCount(n):
  if n < base:
    return 1
  return 1 + digitCount(n//base)
```

What is the result of the expression `digitCount(7912)`?

In order to answer the question we need to know “what is the definition of `base` at this point in time?”. More generally, in order to evaluate the result of a function we need to be given a pair of two things:

- The code that comprises that function, which we will call ϕ
- The function that maps identifiers to values, i.e. our identifier definitions, which we will call δ

Considering `digitCount`

```
def digitCount(n):
  if n < base:
    return 1
  return 1 + digitCount(n//base)
```

What is the result of the expression `digitCount(7912)`?

So now, consider the following questions:

- What does digitCount(7912) evaluate to when $\delta = \{\mathit{base} \rightarrow 4\}$
- What does digitCount(7912) evaluate to when $\delta = \{\mathit{base} \rightarrow 6\}$

Note how if the definition of δ is not given we cannot answer these questions!

Modelling impurity

As we introduce impurity to our code, we require more and more information to understand and model the behaviour of our code. The question about `digitCount` extends more generally to our entire program — should our program not just be the execution of pure functions then in order to model what a program we currently need the pair of data (ϕ, δ).

Furthermore, our pair (ϕ, δ) can be modified each time a statement of our program executes, since statements of our program may modify δ by creating new definitions, or updating the definition of identifiers.

Understanding state

So, how do we keep track of the state our program? As each line of our program is executed we must understand that it may modify our state. As such, in our mental model we must consider that at the beginning of our program we have our state and δ is empty. Each statement produces a *new* δ to be used by the next statement, which may or may not have been changed.

```
# delta : {}
x = 22 # delta now {x -> 22}
z = 3*x # delta now {z -> 66, x-> 22}
```

A new form of impurity

A function accessing a global variable is not the only form of impurity that occurs commonly in programming.

One of the most common occurrences of impurity in programming comes from *input*.

Ultimately the abstract goal of every program is to take some input data, transform it in some way, and produce some output data. Where can that input come from?

Reading input

Input can come from any number of places, one of the common places it comes from is from text that is typed on a keyboard by a user.

Since input can be typed, at will, by a user it can be thought of as a potentially infinite stream of characters. It is worth noting however that while input *can* come from a user typing on a keyboard it does not *have* to.

We will shortly learn how to access input in Python.

Aside: special characters

We have already learned that the characters of our strings are actually represented by numbers which are mapped to characters by the ASCII encoding.

Consider a program that allows you to write a text document, like whatever program you use when you write your Python programs. When you save your text file all the characters of your file must be saved and in the order you wrote them.

When you write a Python program however you don't just write one long string, you break up the text you write by using the enter key to add a line break and go to the next line. When you open up your file the program that displays your text file to you still has the line breaks in the same places — so somehow the places you hit the enter key are "remembered".

The new line character

The way that text files “remember” where the user hit the enter key is actually that whenever the enter key is used to provide a line feed in text it is actually just another character that you are typing like any other character!

The character that is entered into a text file by hitting the enter key is called the *new line* character, and in the ASCII table its ordinal value is 10.

How can we write a string literal that contains the newline character? Our string literals can’t go over multiple lines!

The escape character

The backslash character \ is often given special meaning in programming languages and used specially. Rather than the character \ representing a backslash it is instead a metacharacter called the *escape character*.

Instead of literally representing a character the escape character \ instead must be followed by another character and its purpose is "to escape the regular meaning of the following character".

In this way the escape character is useful for writing characters that otherwise don't have a representation!

Using the escape character

Using the escape character is simple, in a string literal you may place the escape character before an escapable[1] character, and instead of using that characters regular meaning in your string its escaped meaning is used. For example, the character `n` is an escapable character, it's regular meaning is the fourteenth letter of the Latin alphabet, its escaped meaning is the newline character! Consider:

```
ord("n") -> 110
len("n") -> 1
ord("\n") -> 10
len("\n") -> 1
```

[1]Only certain characters are escapable, we will see some of them.

Understanding the escape character

```
ord("\n") -> 10
len("\n") -> 1
```

It is very important to note that in our string literal when we write `\n` the pair of characters are our way to denote the *single* newline character. As such the length of the string above is one, it is not the string that contains a backslash and the letter n it is the string that contains one character — the newline character.

It is this newline character that is used to represent line breaks in text.

Other escaped character

Some other examples of common escaped characters are:

- `\t` — the tab character
- `\r` — the carriage return character
- `\\` — represent the literal \character
- `\"` — escape the normal meaning of the double quote character (e.g. might indicate the end or start of a string) and mean literally the quote character
- `\'` — same as above but for single quote

Note, in a string each of these represents only *one* character! That is why the escape character is called a metacharacter, it modifies the meaning of other characters but does not actually count as a character in the string.

Example uses of escape characters

```
# Can use to represent a string that contains
# both single and double quotes
# Represents the sequence of characters:
# Rob's friend said "wow!"
s = "Rob's friend said \"wow!\""
# To insert line breaks and tabs, the following
# represents the sequence of characters
'''
Wow
  great!
    fun!
'''
s = "Wow\n\tgreat!\n\t\tfun!"
# To represent backslashes in our text
# Represents the series of characters
# \(^.^)/
s = "\\(^.^)/"
```

The `input` function

Python provides a built-in function `input` which we call with zero[2] arguments.

The function `input` returns a string, but which string? Since we are calling `input` with zero arguments there is only one of two options:

- `input` is a constant function, that always returns the same value
- `input` is an impure function and doesn't always return the same value for the same arguments

[2]It can also be called with one argument, but need more context before we can discuss that.

Using the `input` function

The `input` function is an impure function as it does not always return the same result for the same arguments. However, in order to understand what the input function returns we need to add even *more* context to the current state of our program!

Specifically, the behaviour of the input function is the following

- `input` reads characters from input stream until the first newline character is read from the input stream
- returns the string of all the characters read except without the newline character at the end

Determining the result of `input`

Given what we understand about the `input` function consider the following questions:

- What does `input` evaluate to when $\delta = \{x \rightarrow 4,\ y \rightarrow 7\}$
- What does `input` evaluate to when $\delta = \{t \rightarrow 6\}$

Determining the result of `input`

Given what we understand about the `input` function consider the following questions:

- What does `input` evaluate to when $\delta = \{x \rightarrow 4,\ y \rightarrow 7\}$
- What does `input` evaluate to when $\delta = \{t \rightarrow 6\}$

We can't answer this question with the information given alone! Specifically, we are told the `input` returns the sequence of characters read from input stream up to the first seen newline character.

Determining the result of `input`

Given what we understand about the `input` function consider the following questions:

- What does `input` evaluate to when $\delta = \{x \to 4,\ y \to 7\}$
- What does `input` evaluate to when $\delta = \{t \to 6\}$

We can't answer this question with the information given alone! Specifically, we are told the `input` returns the sequence of characters read from input stream up to the first seen newline character.

As such, with the introduction of input to our programs it is not enough to know ϕ and δ to understand what the program does...

Example function specification for `input`

We now provide based on our understanding so far a function specification for `input`

```
def input():
  '''
  input produces a string which is the sequence of characters
        seen in the input stream up to but not including
        the first newline character

  returns - a str

  Examples:
    input() -> ???
    input () -> ???
  '''
```

Modelling input

With the addition of input we now need even *more* information as to "what is the state of my program at this point in time".

The information we know is what characters are available in the input stream. Instead of our programs state being represented by the pair (ϕ, δ) we will now represent our programs state with the triplet (ϕ, δ, ι).

ϕ and δ retain their previous meanings. We will represent our input, ι, as a string of all the remaining characters that are available to be read from input.

A note on ι

Our definition of ι is a little bit unrealistic, since it presupposes we have the complete list of all characters that will be fed as input to our program.

The actual input a user types will depend on the user and what the program displays.

However, each time the program runs there *is* one single string that represents all the characters that were entered as input. So whatever initial value we choose for ι represents only one potential run of that program — which is the best we can do since our program's behaviour depends on the input!

Using our model of input

Let's now consider our model of input and the python function `input`. Assume that $\iota =$ `"hello\nfriend\nwow"` at the beginning of the following program:

```
s1 = input()
s2 = input()
```

Using our model of input

Let's now consider our model of input and the python function `input`. Assume that $\iota =$ `"hello\nfriend\nwow"` at the beginning of the following program:

```
s1 = input()
s2 = input()
```

At the end of these two statements what is the value stored in `s1`? What is the value stored in `s2`?

Using our model of input

$\iota =$ `"hello\nfriend\nwow"` at the beginning of the following program:

```
s1 = input()
s2 = input()
```

If we assume the definition and function specification given for `input` previously then `input` reads the characters from input up to the first newline character and returns that string.

In that case both $s1$ and $s2$ would be the string `"hello"`. Futhermore, that would mean *every* call to `input` in this execution of the program would result in `"hello"`. This would not be very useful, however. Clearly something is missing in our definition for input.

Updating our model of input

We now have to update our model of input ι with a new rule:

- Whenever a character is read from ι that character is *consumed* (removed from ι).

This means that much as our δ can change throughout the run of our program, so to will ι. Since the state of our program is represented by the triplet (ϕ, δ, ι) changes to one of these are changes to the state our program exists in.

Whenever a function changes the state of our program in an observable way we call that a *side effect* of the function.

Updating our function specification

Now that we live in a world with side effects, we must update our function specification to indicate the side effects of a function. Our new function specification will be of the form:

```
def fnName(...):
  '''
  fnName purpose statement of fnName, as before

  parameter and return types, as before

  Side Effects:
    listing of side effects
    If no side effects can state none

  Examples, as before
  '''
```

Updating input's function specification

Now, we need an updated function specification for input

```
def input():
  '''
  input produces a string which is the sequence of characters
        seen in the input stream up to but not including
        the first newline character

  returns - a str

  Side Effects:
    Removes the characters read in from the
    input stream.

  Example:
    iota is "hello\nfriend\nwow\n"
    input() -> "hello"
    Side effects: iota becomes "friend\nwow\n"
  '''
```

A note on examples

Note that when we write function specifications for functions that have side effects, and for impure functions, we must now include in each example the state that is necessary to understand the result and also include not only the return value of the function but what side effects take place!

A world with impurity and side effects is a more complex world to reason about and understand!

Evaluating `input`

Now we can answer the question as to what happens in the program below if $\iota =$ `"hello\nfriend\nwow\n"` at the beginning of the following program:

```
# iota is "hello\nfriend\nwow\n"
s1 = input() # Returns "hello"
# iota is now "friend\nwow\n"
s2 = input() # Returns "friend"
# iota is now "wow\n"
```

Each call to `input` is dependent on the current state of ι and also modifies the current state of ι!

Using `input`

We now consider writing our own function that reads from input.

We will write the function `simonSays` which takes a single string parameter `uName` which is the users name. The function will then read one string given from the user in the input stream and returns a message specifying what the user said.

simonSays

```
def simonSays(uName):
  '''
  simonSays takes a username parameter and reads a l
            line of input from the user and returns
            a string specifying what the user said

  uName   - str
  returns - str

  Side Effects:
    removes characters from iota up to and including
    the first newline character.

  Examples:
    iota is "oh I don't know\ncheers\n"
    simonSays("Rob") -> "Rob says 'Oh I don't know'"
    Side effects: iota becomes "cheers\n"
  '''
```

simonSays

```
def simonSays(uName):
  uMsg = input()
  return uName + " says '" + uMsg + "'"
```

A note on `input`

An important detail of the function specification for `input` is that it returns a string!

That means even if the user types a series of digits you will receive a string of those digits.

It is important that if you read a message from input that you expect to be a number you must create the number that the string represents, as we have done in the past with the function `strToInt`.

Conversion between types

Python provides built-in functions for converting between types when it is appropriate. These functions are the called with the name of the type you would like to convert to.

However, these functions can only be used when a conversion is possible. Some examples:

```
int("532") -> 532
str(23732) -> "23732"
float("22.3") -> 22.3
int("2f3") -> An error occurs
```

We may now use such functions for ease of conversion between types.

Working with input

Practice Question: write the function `userReplace` that has the following details:

- Has a single `LList` parameter
- Reads two strings from the input stream, interprets the first string as an index *i* to the `LList` and the second string as an integer value *v*
- Your function should return the result of replacing the *ith* item of the given `LList` with the value *v*

Example:

```
# iota is "2\n457\n"
userReplace(LL(4, 12, 55, 3, 2))
  -> LL(4, 12, 457, 3, 2)
# iota becomes ""
```

Talking to the user

We now have a model of our program's state that allows us to account for reading input, potentially from a user.

However, if our input is in fact coming from a user then we may need some way to convey messages to the user about what values we would like for them to give us.

What we need is some way to display messages for the user to read — what we need is *output*.

The `print` function

We now introduce the built-in function `print` which allows for printing a message to the screen.

The function `print` returns None, as it is not meant to calculate and produce a value, its only purpose is to output a message.

This means that we will have a hard time currently writing our function specification and examples for `print`.

Specification for `print`

```
def print(a0, a1, ..., an):
  '''
  print displays a message to the screen the
        message is each of a0, ..., an interpreted
        as strings and with spaces inbetween and
        followed by a newline.

  a0, ..., an - Any
  returns     - None

  Side Effects:
    ???

  Examples:
    print(5, "hello") -> None
    print("wowow") -> None
  '''
```

Specification for `print`

It becomes clear from our function specification that we are missing something. Since `print` always returns `None` clearly it's return value is of no consequence, so our examples are meaningless.

The work that `print` actually performs is to change the state of our program, but which state?

We need to add to our model of our program's state the series of characters we have printed to the screen which is our *output* — we will call this ω.

Modelling output

Now that our program can output characters we have another part of our programs state to represent — ω. Whenever a program first begins ω will be represented by the empty string, since the program has not yet printed anything.

Each time a program prints a message out we will model that in our state by replacing ω with the result of concatenating the printed string to the end of ω. Once characters have been output they can never be unoutput, so ω only grows.

Now, with output being possible, to fully model the state of our program we must represent our state instead with the quartet (ϕ, δ, ι, ω).

Updated print

```
def print(a0, a1, ..., an):
  '''
  print displays a message to the screen the
        message is each of a0, ..., an interpreted
        as strings and with spaces inbetween and
        followed by a newline.

  a0, ..., an - Any
  returns     - None

  Side Effects:
    Displays a string to the screen, modifying omega

  Examples:
    omega is "hmm"
    print(5, "hello") -> None
    omega becomes "hmm5 hello\n"
  '''
```

Reality of output

It is worth noting that ω is our *model* for how we specify exactly what functions that modify the state of our output.

The reality is that when `print` is called that message is displayed to the screen where the user can see it!

This can be a very useful way to ask the user questions, or inform the user of information.

Additional notes about `print`

Our specification for `print` noted that `print` could take *any* number of arguments and would display to the screen the string version of each of those arguments, separated by space characters, and ending in the newline.

That is what the print function does by default, but what if we don't want it to separate our strings with space characters? What if we don't want it to place a newline at the end of our message to print?

`print` special parameters

The `print` function has some special parameters called *named* parameters. Namely, these two parameters are called `sep` and `end`, and they are used to tell `print` what string we would like inserted in-between our arguments and what string we would like placed at the end of all of our arguments respectively.

Because `print` can take any number of arguments to be the data to display to the screen we need a special way to signify which arguments we're providing for parameters `sep` and `end`. This is the nature of named parameters. When calling the function `print` we can provide *keyword arguments* for these parameters.

Keyword arguments

If a function has named parameters then when calling that function the value for the named parameters is provided by specifying in the argument list of your function call expression an argument of the form `name=expr`. For example:

```
# Omega is ""
print("a", "b", "c")
# Omega now "a b c\n"
print("x", "y", "z", sep="-")
# Omega now "a b c\nx y z\n"
print(1, 2, 3, sep="", end="XX")
# Omega now
# "a b c\nx y z\n123XX"
```

Displaying a banner

Practice Question: Write the function `displayBanner` that takes two parameters `msg` and `borderChar` and prints a banner to the screen. The banner is some formatted text, for example if `msg` is `"Hello world"` and `borderChar` is `"-"` then Your function would print:

```
---------------
- Hello world -
---------------
```

You may assume `msg` contains no new lines and `borderChar` is one character long.

Displaying a banner

Some more banner examples:

```
displayBanner("wow", "%") -> None
# Displays:
# %%%%%%%
# % wow %
# %%%%%%%

displayBanner("shucks golly", "^") -> None
# Displays:
# ^^^^^^^^^^^^^^^^
# ^ shucks golly ^
# ^^^^^^^^^^^^^^^^
```

More generalized banner

Practice Question: Now, make a new version of `displayBanner` that works for messages that may contain newlines and borders that are more than one character. For example `displayBanner("Jeez\nthis is a lot!\ncool!", "oOx")` would display

```
oOxoOxoOxoOxoOxoOxoOxo
oOx      wow       oOx
oOx this is a lot! oOx
oOx     cool!      oOx
oOxoOxoOxoOxoOxoOxoOxo
```

Table of Contents

Mutability

So far all the data types we've worked with have been what is called *immutable*.

If data is *immutable* (unchangeable) it means that the data itself cannot be *mutated* (changed), only new values can be produced.

The intuitive way to think about this is that if I have the number two I cannot change the meaning of the number two, it always means two. I can only produce an integer which has a different meaning.

What is mutation?

Mutation specifically is when a piece of data actually *changes* (or mutates) instead of producing a new value. Mutation is yet another way in which the state of our program can be changed.

The nature of mutation is that it can lead to behaviour that is otherwise unintuitive for our programs. However, before we can understand mutation we need some motivating examples.

Assignment is not mutation

Now that we have understanding of scope, we can model in our minds the result of the following Python program:

```
x = 10
def double(x):
  x = x*2

double(x) -> None
x -> 10
double(x) -> None
x -> 10
```

We know that when `double` is assigning `x` to `x*2` it is assigning the locally defined identifier `x` which is its parameter. As such, `double` cannot have any effects outside its own scope — it can only return a value to whoever called it.

Assignment is not mutation

Now that we have understanding of scope, we can model in our minds the result of the following Python program:

```
x = 10
def double(x):
  x = x*2

double(x) -> None
x -> 10
double(x) -> None
x -> 10
```

What if we *wanted* a function to modify data outside its own scope though, how could we achieve that? Currently, we cannot.

The `list` type

In order to understand mutability we must introduce a mutable data type. We now introduce the built-in Python type called `list`.

The `list` data type is very much like the `tuple` data type we have already learned about. In fact, everything we have learned about tuples also applies to lists, so then how do lists and tuples differ?

The `list` type is mutable while the `tuple` data type is immutable.

Mutability — the ability to change

If a data type is mutable then the data itself can be changed, rather than creating a new value. One of the ways this can be performed with lists is that the individual items of a `list` can be assigned to through indexing, which cannot be done with a `tuple`.

```
t = (1, 2, 3)
t[0] = 5 # illegal, causes an error
l = [1, 2, 3] # square brackets used for list literals
l[0] = 5 # l is now [5, 2, 3]
```

Mutability versus reassignment

The differences between mutability and simply creating a new value are subtle at first, but become very apparent as we learn more. At first, it may seem like the two statements below are no different in their effects, but that is very much not true.

```
t = (1, 2, 3)
t = (5,) + t[1:]
l = [1, 2, 3]
l[0] = [5]
```

After these statements have run both my identifiers `t` and `l` refer to sequences of the values five, two, and three. One may be tempted to think that since in this case the outcome is the same that the processes are the same — they very much are not!

Observing mutability

Consider the following set of Python statements

```
t1 = (1, 2, 3)
t2 = t1
t1 = (5,) + t[1:]
l1 = [1, 2, 3]
l2 = l1
l1[0] = [5]
print(t1, t2) # prints (5, 3, 3) (1, 2, 3)
print(l1, l2) # prints [5, 2, 3] [5, 2, 3]
```

`l2` has also been changed to `[5, 2, 3]`... but how?

Modelling mutability

Let's consider our programs state before and after each of these statements. For space we will always just show the next statement and the immediately prior statement that was just executed at a time.

```
# iota : ""
# omega : ""
# delta : {}
t1 = (1, 2, 3)
```

Modelling mutability

Let's consider our programs state before and after each of these statements. For space we will always just show the next statement and the immediately prior statement that was just executed at a time.

```
t1 = (1, 2, 3)
# iota : ""
# omega : ""
# delta : {t1 -> (1, 2, 3)}
t2 = t1
```

Modelling mutability

Let's consider our programs state before and after each of these statements. For space we will always just show the next statement and the immediately prior statement that was just executed at a time.

```
t2 = t1
# iota : ""
# omega : ""
# delta : {t1 -> (1, 2, 3), t2 -> (1, 2, 3)}
t1 = (5,) + t[1:]
```

Modelling mutability

Let's consider our programs state before and after each of these statements. For space we will always just show the next statement and the immediately prior statement that was just executed at a time.

```
t1 = (5,) + t[1:]
# iota : ""
# omega : ""
# delta : {t1 -> (5, 2, 3), t2 -> (1, 2, 3)}
l1 = [1, 2, 3]
```

Modelling mutability

Let's consider our programs state before and after each of these statements. For space we will always just show the next statement and the immediately prior statement that was just executed at a time.

```
l1 = [1, 2, 3]
# iota : ""
# omega : ""
# delta : {t1 -> (5, 2, 3), t2 -> (1, 2, 3),
#          l1 -> [1, 2, 3]}
l2 = l1
```

Modelling mutability

Let's consider our programs state before and after each of these statements. For space we will always just show the next statement and the immediately prior statement that was just executed at a time.

```
l2 = l1
# iota : ""
# omega : ""
# delta : {t1 -> (5, 2, 3), t2 -> (1, 2, 3),
#          l1 -> [1, 2, 3], l2 -> [1, 2, 3]}
l1[0] = [5]
```

Modelling mutability

Let's consider our programs state before and after each of these statements. For space we will always just show the next statement and the immediately prior statement that was just executed at a time.

```
l1[0] = [5]
# iota : ""
# omega : ""
# delta : {t1 -> (5, 2, 3), t2 -> (1, 2, 3),
#          l1 -> [5, 2, 3], l2 -> [1, 2, 3]}
print(t1, t2)
```

Modelling mutability

Let's consider our programs state before and after each of these statements. For space we will always just show the next statement and the immediately prior statement that was just executed at a time.

```
print(t1, t2)
# iota : ""
# omega : "(5, 2, 3) (1, 2, 3)\n"
# delta : {t1 -> (5, 2, 3), t2 -> (1, 2, 3),
#          l1 -> [5, 2, 3], l2 -> [1, 2, 3]}
print(l1, l2)
```

Modelling mutability

Let's consider our programs state before and after each of these statements. For space we will always just show the next statement and the immediately prior statement that was just executed at a time.

```
print(l1, l2)
# iota : ""
# omega : "(5, 2, 3) (1, 2, 3)\n[5, 2, 3] [1, 2, 3]\n"
# delta : {t1 -> (5, 2, 3), t2 -> (1, 2, 3),
#          l1 -> [5, 2, 3], l2 -> [1, 2, 3]}
```

Our model does not produce the result that Python does! There's nothing in our model that can account for what is happening in this case... as such we must expand our model to understand mutability.

Modelling mutability

In order to be able to model mutability we need to update our model of our program's state in two ways. Currently, to resolve an identifier's value we are simply using δ which maps identifiers directly to values. This model is insufficient to model what is happening in our previous example though. So, one of our changes must be to δ.

Our other change will be to add a new portion of our state μ, and now our programs state will now be a quintet of $(\phi, \delta, \mu, \iota, \omega)$. But what does μ represent, and how are we changing δ?

Introducing memory

In our computers our values must actually be stored physically, the location in which our values are stored is called *memory*. Our μ will be a very abstract approximation of our computer's memory.

Introducing memory

In our computers our values must actually be stored physically, the location in which our values are stored is called *memory*. Our μ will be a very abstract approximation of our computer's memory.

In our abstract view of memory we will define memory to be a series of boxes, and we'll assume for simplicity's sake that each box can hold any[3] value. Each box of our memory abstraction will have a unique integer by which it can be identified.

[3] This is a lie, but a simplification to aid our understanding

Introducing memory

In our computers our values must actually be stored physically, the location in which our values are stored is called *memory*. Our μ will be a very abstract approximation of our computer's memory.

In our abstract view of memory we will define memory to be a series of boxes, and we'll assume for simplicity's sake that each box can hold any[3] value. Each box of our memory abstraction will have a unique integer by which it can be identified.

Since in our abstraction each memory box has a unique integer that identifies it which we will call its location, and each memory box can hold any value our μ will be a mapping from integers that represent memory locations to values which are the data of our program.

[3] This is a lie, but a simplification to aid our understanding

Updating δ

Since we now have μ which maps memory locations to values it is no longer the case that values should be stored directly in δ. Instead, now, δ must map from identifiers to memory locations. Now to resolve an identifier's value one must find the memory location the identifier maps to in δ and use that memory location to find the value it maps to in μ.

Additionally, we must update our understanding of our collection data types. Previously, when representing our collections in δ we represented them as a collection of values, we must now instead represent them as a collection of memory locations.

Modelling memory

We must also update for our new model how Python statements affect our μ and δ. Again, this is an abstract simplification to model the behaviour of Python.

- Whenever a non-identifier expression evaluates to a value a new memory location is created to store the resultant value. When choosing new memory locations any unused value is fine, starting from zero and incrementing by one each new value is valid.
- When an identifier is assigned to an expression then δ is updated to map the identifier to the memory location the result of that expression was placed in.
- Whenever assignment is made to index *i* of a `list` then the *ith* item of the `list` in μ is updated to refer to the memory location of the value it is being assigned to.

Applying our model

We will now apply our new model to the program we could not explain before, to see what is happening. We will omit ι and ω for space, but properly we should always model all four of our state components.

```
# delta : {}
# mu : {}
t1 = (1, 2, 3)
```

Applying our model

```
t1 = (1, 2, 3)
# delta : {t1 -> 116}
# mu : {104: 1, 108: 2, 112: 3,
#         116: (104, 108, 112)}
t2 = t1
```

Applying our model

```
t2 = t1
# delta : {t1 -> 116, t2 -> 116}
# mu : {104 -> 1, 108 -> 2, 112 -> 3,
#         116 -> (104, 108, 112)}
t1 = (5,) + t[1:]
```

Applying our model

```
t1 = (5,) + t[1:]
# delta : {t1 -> 124, t2 -> 116}
# mu : {104 -> 1, 108 -> 2, 112 -> 3,
#        116 -> (104, 108, 112),
#        120 -> 5,
#        124 -> (120, 108, 112)}
l1 = [1, 2, 3]
```

Applying our model

```
l1 = [1, 2, 3]
# delta : {t1 -> 124, t2 -> 116,
#          l1 -> 128}
# mu : {104 -> 1, 108 -> 2, 112 -> 3,
#       116 -> (104, 108, 112),
#       120 -> 5,
#       124 -> (120, 108, 112)}
#       128 -> [104, 108, 112]
l2 = l1
```

Applying our model

```
l2 = l1
# delta : {t1 -> 124, t2 -> 116,
#          l1 -> 128, l2 -> 128}
# mu : {104 -> 1, 108 -> 2, 112 -> 3,
#       116 -> (104, 108, 112),
#       120 -> 5,
#       124 -> (120, 108, 112)}
#       128 -> [104, 108, 112]
l1[0] = [5]
```

Applying our model

```
l1[0] = [5]
# delta : {t1 -> 124, t2 -> 116,
#          l1 -> 128, l2 -> 128}
# mu : {104 -> 1, 108 -> 2, 112 -> 3,
#       116 -> (104, 108, 112),
#       120 -> 5,
#       124 -> (120, 108, 112)}
#       128 -> [120, 108, 112]
```

Here, we see what it means for a piece of data to be mutated. The `list` which resides at memory location 128 had its first item changed from 104 to 120 — rather than creating a *new* `list` we have *mutated* the existing one in memory!

Applying our model

```
l1[0] = [5]
# delta : {t1 -> 124, t2 -> 116,
#          l1 -> 128, l2 -> 128}
# mu : {104 -> 1, 108 -> 2, 112 -> 3,
#       116 -> (104, 108, 112),
#       120 -> 5,
#       124 -> (120, 108, 112)}
#       128 -> [120, 108, 112]
```

However, our model now demonstrates why `l2` has changed as well! We can see that `l1` and `l2` in δ both map to 128, the same memory location. Since the `list` located at this memory location has been mutated the result is observable in both `l1` and `l2`!

Applying our model

```
l1[0] = [5]
# delta : {t1 -> 124, t2 -> 116,
#          l1 -> 128, l2 -> 128}
# mu : {104 -> 1, 108 -> 2, 112 -> 3,
#       116 -> (104, 108, 112),
#       120 -> 5,
#       124 -> (120, 108, 112)}
#       128 -> [120, 108, 112]
```

This is what is meant by mutation, when a value stored in memory is changed in place to a new value. All of our data types before lists have been immutable, which means they *cannot* be modified, instead only new values could be created and placed in new places in μ.

Aliasing

What we have observed, when two identifiers map to the same memory location, is called *aliasing*. Aliasing has been happening all the time throughout this course as whenever identifiers are assigned to other identifiers the only change made is to update δ such that the one on the left-hand side maps to the same location as the one on the right-hand side.

However, since we have only had immutable data types so far aliasing has not mattered — if two identifiers map to the same location that stores an immutable piece of data it is not observable, since that data is immutable and cannot be changed!

Redefining mutability

Now that we have our mental model we can properly define mutation, mutability, and immutability.

- Mutation is when a value in μ is modified *in place*, not when a new value is created, nor when δ is updated.
- A data type is mutable if there are mechanisms by which we can mutate it.
- A data type is immutable if there are no mechanisms by which we can mutate it.

Revisting `double`

Now we reconsider the function `double` we discussed earlier:

```
def double(x):
  x = x*2
x = 10
double(x)
print(x) # prints 10
```

This function could not modify the variable `x` outside itself for multiple reasons. When can a function modify the state of our program outside its own scope? When it mutates data.

New `double`

```python
def double(l):
  l[0] = l[0]*2

x = [10]
double(x)
print(x[0]) # prints 20
```

Here, the function `double` is called on the list `x`, and it is able to mutate it. What has happened?

New double

```
def double(l):
  l[0] = l[0]*2

x = [10]
double(x)
print(x[0]) # prints 20
```

Before `double` is called our μ and δ may look like the following:

- μ $\{224 \rightarrow 10, 228 \rightarrow [224]\}$
- δ $\{x \rightarrow 228\}$

New `double`

```
def double(l):
  l[0] = l[0]*2

x = [10]
double(x)
print(x[0]) # prints 20
```

When `double` is called the parameter `l` must be added to δ, but what should it map to? Since our argument was simply x we use 228.

- μ $\{224 \rightarrow 10, 228 \rightarrow [224]\}$
- δ $\{x \rightarrow 228, l \rightarrow 228\}$

New `double`

```
def double(l):
  l[0] = l[0]*2

x = [10]
double(x)
print(x[0]) # prints 20
```

Inside the function we execute `l[0] = l[0]*2`, since `l` maps to memory location 228 we are operating on the first item in the `list` located at that memory location. This is a side effect!

- $\mu\ \{224 \rightarrow 10, 228 \rightarrow [232], 232 \rightarrow 20\}$
- $\delta\ \{x \rightarrow 228, l \rightarrow 228\}$

New `double`

```
def double(l):
  l[0] = l[0]*2

x = [10]
double(x)
print(x[0]) # prints 20
```

After the function returns the parameter `l` is no longer in scope, and is removed from δ. However, the the data was mutated and so the change persists and is observable in `x`.

- μ $\{224 \rightarrow 10, 228 \rightarrow [232], 232 \rightarrow 20\}$
- δ $\{x \rightarrow 228, \}$

Reading our model

Practice Question: Consider the following model of a running program's state:

- ι — `"2\n0\n4\n3\n"`
- ω — `""`
- δ — $\{l \rightarrow 332, z \rightarrow 272\}$
- μ — $\{192 \rightarrow 77, 196 \rightarrow 500, 200 \rightarrow 7, 204 \rightarrow 8,$
  $208 \rightarrow 10, 272 \rightarrow [192, 196], 332 \rightarrow [200, 240, 272, 208]\}$

If that is the state of our program before the following Python statements are executed, what is the state after?

```
x = int(input())
y = int(input())
l[x][y] = 17
print(l)
print(z)
```

Avoiding aliasing

What if we want to assign an identifier to another without aliasing? Well, one of the rules of our model was the following:

Avoiding aliasing

What if we want to assign an identifier to another without aliasing? Well, one of the rules of our model was the following:

- Whenever a non-identifier expression evaluates to a value a new memory location is created to store the resultant value. When choosing new memory locations any unused value is fine, starting from zero and incrementing by one each new value is valid.

So, if we want to avoid aliasing between identifiers but assign to the same value we need an expression that simply evaluates back to the same value.

Avoiding aliasing

What if we want to assign an identifier to another without aliasing? Well, one of the rules of our model was the following:

- Whenever a non-identifier expression evaluates to a value a new memory location is created to store the resultant value. When choosing new memory locations any unused value is fine, starting from zero and incrementing by one each new value is valid.

So, if we want to avoid aliasing between identifiers but assign to the same value we need an expression that simply evaluates back to the same value.

Avoiding aliasing

Since aliasing only matters when we have mutable data types the only type we know about where we need to worry about this is lists. What expression would evaluate back to the same `list` it was given?

Avoiding aliasing

Since aliasing only matters when we have mutable data types the only type we know about where we need to worry about this is lists. What expression would evaluate back to the same `list` it was given?

Several! One expression we can use is slicing from beginning to end.

Avoiding aliasing

Since aliasing only matters when we have mutable data types the only type we know about where we need to worry about this is lists. What expression would evaluate back to the same list it was given?

Several! One expression we can use is slicing from beginning to end.

```
l = [1, 2, 3]
z = l[::]
l[0] = 5
print(l) # [5, 3, 3]
print(z) # [1, 2, 3]
```

Avoiding aliasing

Since aliasing only matters when we have mutable data types the only type we know about where we need to worry about this is lists. What expression would evaluate back to the same `list` it was given?

Several! One expression we can use is slicing from beginning to end.

```
l = [1, 2, 3]
z = l[::]
l[0] = 5
print(l) # [5, 3, 3]
print(z) # [1, 2, 3]
```

This is not foolproof, however...

Beware shallow copying

What we did in the previous example was to make `z` be a *shallow* copy of `l`. A shallow copy is one in which we created a new list, but for each memory location that was an item in the original list we copied it into our new list.

So, let us model what happened.

Beware shallow copying

```
l = [1, 2, 3]
```

In the above line first the literal expression `[1, 2, 3]` is evaluated which must make space in memory for each of the values 1, 2, 3 and a list of those three items.

- μ
    - $762 \to 1$
    - $768 \to 2$
    - $772 \to 3$
    - $776 \to [762, 768, 772]$
- δ

Beware shallow copying

```
l = [1, 2, 3]
```

After the expression is evaluated and its result, 776, has been resolved it can be assigned to `l` in δ.

- μ
  - $762 \to 1$
  - $768 \to 2$
  - $772 \to 3$
  - $776 \to [762, 768, 772]$
- δ
  - `l` $\to 776$

Beware shallow copying

```
z = l[::]
```

In the above line first the list slice expression `l[::]` is evaluated, which specifically says to create a new list that contains all the values in `l` from the start to the end — that means the exact memory locations stored in that list are copied.

- μ
  - $762 \rightarrow 1$
  - $768 \rightarrow 2$
  - $772 \rightarrow 3$
  - $776 \rightarrow [762, 768, 772]$
  - $782 \rightarrow [762, 768, 772]$
- δ

Beware shallow copying

```
z = l[::]
```

After the expression is evaluated and its result, 782, has been resolved it can be assigned to `z` in δ.

- μ
  - $762 \rightarrow 1$
  - $768 \rightarrow 2$
  - $772 \rightarrow 3$
  - $776 \rightarrow [762, 768, 772]$
  - $782 \rightarrow [762, 768, 772]$
- δ
  - `l` $\rightarrow 776$
  - `z` $\rightarrow 782$

Beware shallow copying

```
l[0] = 5
```

In this case, the value 5 must have space made for it in memory, and then the first item of `l` is updated to store the memory location of 5. Note, that since `z` refers to memory location 782 and not 776 this change does not affect `z`.

- μ
    - $762 \to 1$
    - $768 \to 2$
    - $772 \to 3$
    - $776 \to [786, 768, 772]$
    - $782 \to [762, 768, 772]$
    - $786 \to 5$
- δ
    - `l` $\to 776$
    - `z` $\to 782$

Beware shallow copying

In our example, the shallow copy seems to have done the trick, the mutation of `l` didn't affect `z`. A shallow copy works if our `list` contains only immutable data (and any collections inside of it also contain only immutable data). However, if our `list` contains mutable data a shallow copy is insufficient.

We will now consider such a case.

Beware shallow copying

```python
l = [[10, 20], [30, 40]]
z = l[::]
l[0] = [5, 3]
l[1][0] = 33
print(l) # [[5, 3], [33, 40]]
print(z) # [[10, 20], [33, 40]]
```

In the above example we see that modifying `l[0]` did not change z, yet modifying `l[1][0]` *did*. Luckily, our model demonstrates why this happened. This issue is exactly the problem with a shallow copy.

Beware shallow copying

```
l = [[10, 20], [30, 40]]
z = l[::]
```

After these two statements have executed our state may look like the following

- μ — $\{144 \rightarrow 10, 148 \rightarrow 20, 152 \rightarrow 30, 156 \rightarrow 40,$ $160 \rightarrow [144, 148], 164 \rightarrow [152, 156], 168 \rightarrow [160, 164],$ $172 \rightarrow [160, 164]\}$
- δ — $\{\texttt{l} \rightarrow 168, \texttt{z} \rightarrow 172\}$

Beware shallow copying

```
l[0] = [5, 3]
```

The above statement must create the values 5, and 3, and the list that contains them in memory and update `l[0]` to that location, this update to μ is shown below.

By following our definition of `z` we can see none of the memory locations referenced by `z` or the items contained within them have changed — so `z` remains unchanged.

- μ — $\{144 \rightarrow 10, 148 \rightarrow 20, 152 \rightarrow 30, 156 \rightarrow 40, 160 \rightarrow [144, 148], 164 \rightarrow [152, 156], 168 \rightarrow [184, 164], 172 \rightarrow [160, 164], 176 \rightarrow 5, 180 \rightarrow 3, 184 \rightarrow [176, 180]\}$
- δ — $\{\texttt{l} \rightarrow 168, \texttt{z} \rightarrow 172\}$

Beware shallow copying

```
l[1][0] = 33
```

When the statement above is executed space is made for 33 and the first item of the list that is the second item in `l` is updated to refer to it.

However, since we shallow copied `l` even though `l` refers to the list at 168 and `z` refers to the list at 172 *both* of those lists have the list at `164` as their second item.

- μ — $\{144 \rightarrow 10, 148 \rightarrow 20, 152 \rightarrow 30, 156 \rightarrow 40, 160 \rightarrow [144, 148], 164 \rightarrow [152, 156], 168 \rightarrow [184, 164], 172 \rightarrow [160, 164], 176 \rightarrow 5, 180 \rightarrow 3, 184 \rightarrow [176, 180]\}$
- δ — $\{\texttt{l} \rightarrow 168, \texttt{z} \rightarrow 172\}$

Beware shallow copying

```
l[1][0] = 33
```

So, when we updated `l[1][0]` we are updating the first item of the list located at memory location 164. Since the second item of `z` is also the list located at 164 we can observe the change in `z` a well!

- μ — $\{144 \to 10, 148 \to 20, 152 \to 30, 156 \to 40, 160 \to [144, 148], 164 \to [152, 188], 168 \to [184, 164], 172 \to [160, 164], 176 \to 5, 180 \to 3, 184 \to [176, 180], 188 \to 33\}$
- δ — $\{\texttt{l} \to 168, \texttt{z} \to 172\}$

Beware shallow copying

When we make a shallow copy of a list we are simply copying the memory locations that each item of that list refers to into a new list.

A deep copy would need to take into account that any items that are lists (no matter how nested) would have to be copied as well.

Since our list could have any level of nested lists beneath it, it makes sense to write such a function recursively.

The deepCopy function

```
def deepcopy(l):
  '''
  deepcopy returns a DEEPCOPY of the list l

  l       - X
  returns - X

  Side effects: None

  Examples:
    deepcopy([[1, 2], [3, 4]])
      -> [[1, 2], [3, 4]] BUT ALL ALIASING IS BROKEN!
  '''
```

The deepCopy function

```
def deepcopy(l):
  if type(l) != list:
    return l
  newL = []
  for item in l:
    newL.append(deepcopy(item))
  return newL
```

Classes and Objects

Python is what is called an object-oriented language. In an object-oriented language there are data types called *classes*.

One of the defining features of classes is that they are data types that contain within them functions. Functions contained with a class are called *methods*.

A piece of data that is of a class type is called an *object*.

In Python *all* types are classes and as such all data are objects.

Methods

Methods are simply functions, so we should be able to understand them. The only way methods differ from regular functions is how we call them.

Methods are typically called “on” an object through the “member access operator”.

Member access operator

The form of the member access operator is as follows:

```
expr.identifier
```

Where `expr` is an expression that evaluates to an object and `identifier` is the name of a member of the class type that expression evaluates to.

Using methods

The most common use of the member access operator in Python is to call methods.

For example, the class `str` has a method `upper` that returns the upper case version of a string. For example:

```
s = "foo"
"hello".upper() -> "HELLO"
s.upper() -> "FOO"
```

Understanding methods

It may seem from our example that the method `upper` takes has no parameters, since we provided it no arguments — however that is not true! All methods take at least one parameter — the object on which the method is called. This parameter just appears implicit because we are accessing the member *of* that object.

In fact, the methods of a type can be referenced from the type's name itself. In this case, there *is* no object which the method is being called on so the implicit parameter must be explicitly passed:

```
s = "foo"
str.upper("hello") -> "HELLO"
str.upper(s) -> "FOO"
```

`list` methods

We address methods now as we need to understand some `list` methods that are commonly used in order to mutate lists. For ease of describing the methods we refer to sample function calls.

- `l.append(elem)` — mutates l by adding `elem` to the back, returns `None`. Has an amortized runtime of $\Theta(1)$
- `l.pop()` — mutates l by removing the last element of the list and removes the removed value. Has a runtime of $\Theta(1)$
- `l.pop(i)` — mutates l by removing the item of the list at index i, and returns the removed value. Has a runtime of $\Theta(n)$
- `l.insert(i, elem)` — mutates l by inserting `elem` as a new item before the item at position `i`, returns `None`. Has a runtime of $\Theta(n)$

Sorting a list in place

Practice Question: Previously when we learned about sorting we created new sorted copies of our data. With mutable lists we can perform what is called an *in-place* sorting.

An in-place sorting is a sorting that does not new sequence as a result and typically does not even create any additional sequences to aid in sorting. Instead, it mutates the list it is sorting.

Write a function `inPlaceSort` that takes a single `list` parameter sorts the list in place, thus mutating it to be sorted in non-decreasing order. Your function should not return anything. You may not create any other sequence in any point of your function.

More to learn

There are many more features and data types of Python to learn about should one continue on working in Python.

However, over this course we have built a strong foundation of the basics that allow us to learn and understand how new features can work when as we discover them.

Soon, in CMPUT 275 we will move on from Python and away from our abstracted view of how our code works and instead bring our view closer to reality.