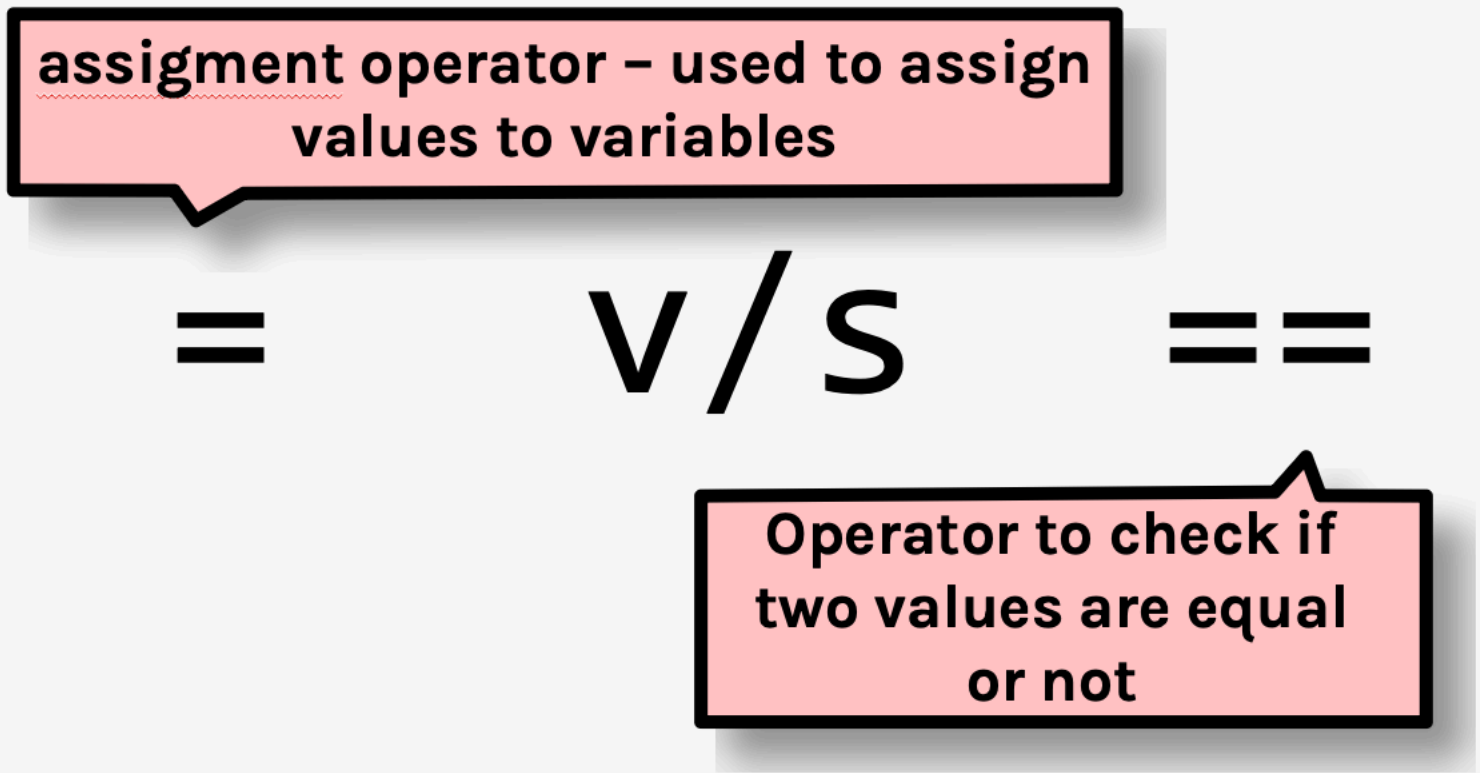
Python as a calculator

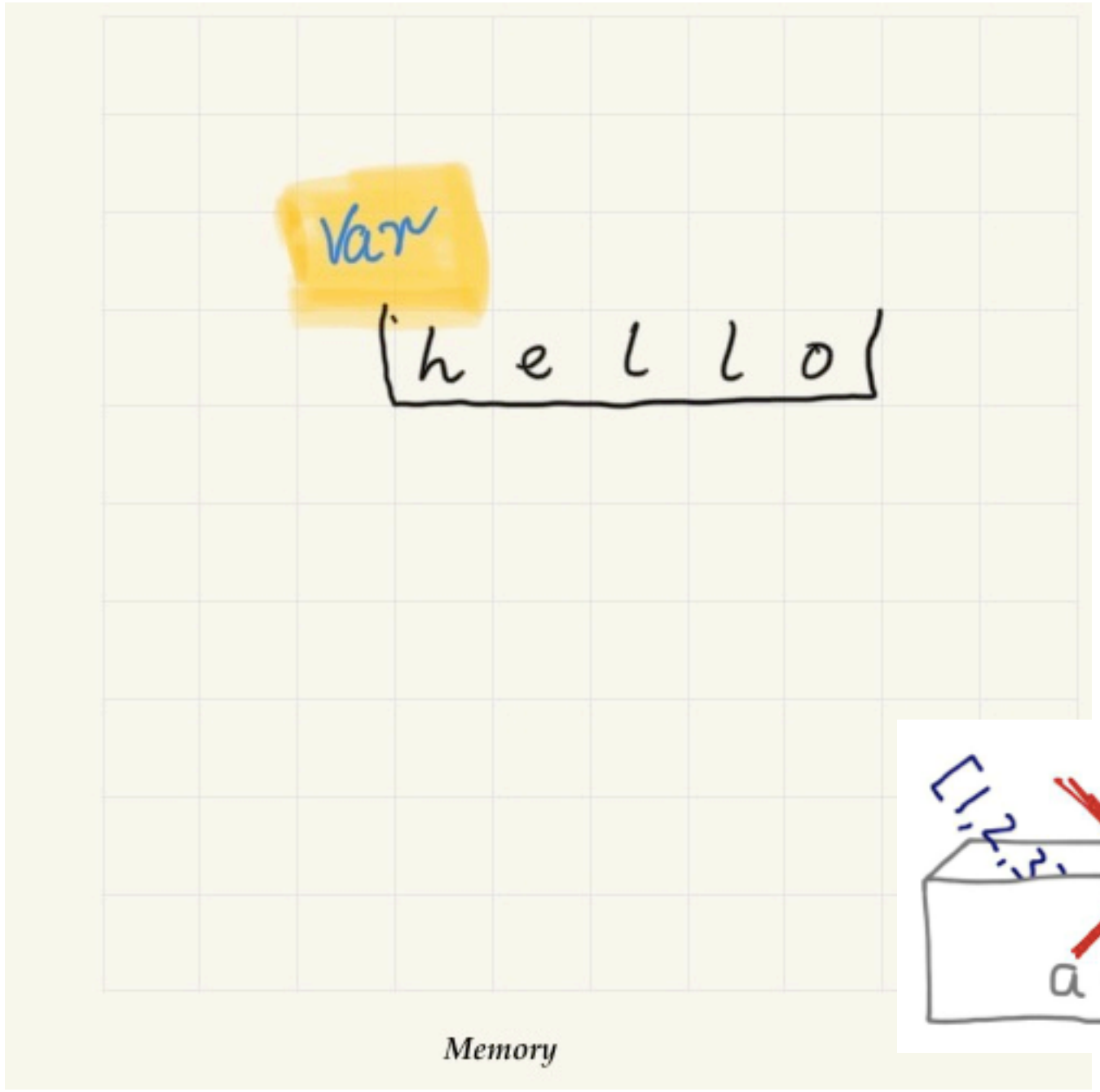
Operator	Description	Example
$+$	adds values on either side	$1.2 + 2 = 3.2$
$-$	subtracts the right value from the left	$1.2 - 0.2 = 1.0$
$\star$	multiplies values on either side	$1.2 * 2 = 2.4$
$/$	divides the left value by the right	$5/2 = 2.5$
$\%$	divides the left value by the right and returns the remainder	$5\%2 = 1$
$\star\star$	exponentiate the left value by the right	$3 \star \star 2 = 9$
$//$	divides the left value by the right and removes the decimal part	$5//2 = 2$

Comparison vs Assignment

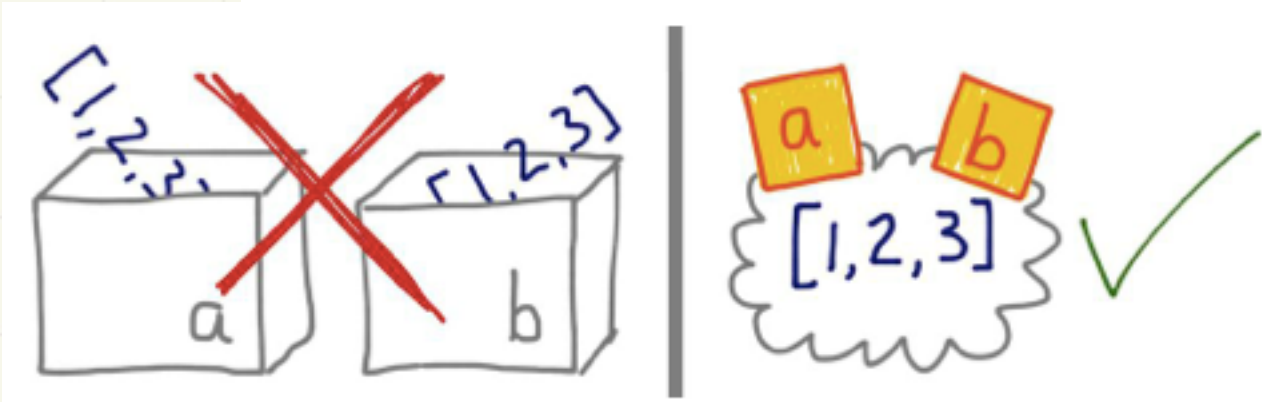


Conditionals

Operator	Description	Example
==	checks if values on either side are equal	$1 == 2$ is False
!=	checks if values on either side are unequal	$1! = 2$ is True
>	checks if left value is greater	$1 > 2$ is False
<	checks if left value is smaller	$1 < 2$ is True
>=	checks if left value is greater or equal	$2 \geq 2$ is True
<=	checks if left value is smaller or equal	$1 \leq 2$ is True



Think of variables as post-its or lookups in a dictionary



Lists

Python is 0-indexed. This means that the first index is 0 not 1.

Create a lazy sequence of numbers from 0 to 9 (Why lazy?):

```
seq = range(10)
```

Create a list:

```
seq = range(10)
lst = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
lst = list(seq) # same as above
```

Add to a list:

```
lst.append(10)
print(lst)
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Get the third element above(value 2):

```
lst[2] # the 2 inside is called an index.
```

List Indexing and Ops

`lst = ['hi', 7, 'c']`. The indices into this list are 0, 1, 2.

Operator	Description	Example
$+$	appends right list to end of left	$['\mathrm{H}'] + [2]$ is $['\mathrm{H}', 2]$
[n]	returns the n -th item	lst $[0]$ is 'hi' (0th item)
$[-\mathrm{n}]$	returns the n -th item from end	lst $[-2]$ is 7
[n : m]	returns items from n up to m	lst $[0:1]$ is $['\mathrm{hi}']$
[n :]	returns items from n on	lst $[1:]$ is $[7,' \mathrm{c}']$
[: n]	returns items up to n	lst $[:2]$ is $['\mathrm{hi}', 7]$

Create a dictionary:

```
d = dict(
    name = 'Alice',
    age  = 18,
    gender = 'F'
)
```

Get a value:

```
print("age", d['age'])
```

```
age 18
```

Set a value:

```
d['job] = 'scientist'
```

Iterate over keys:

```
for key in d.keys():
    print(key, d['key'])
```

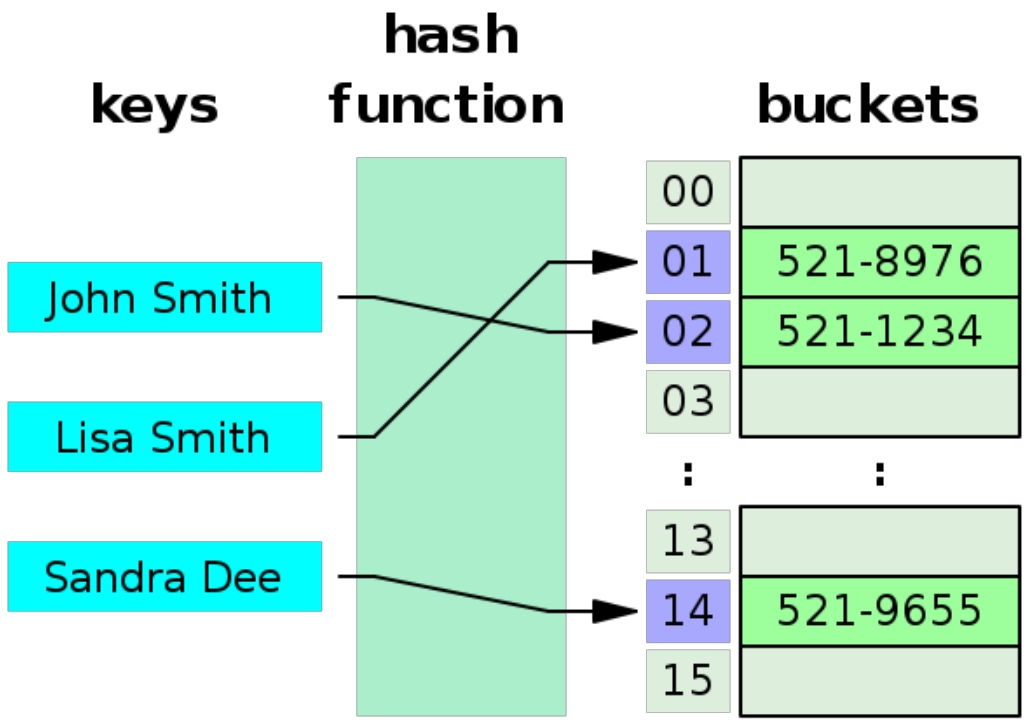
```
name Alice
age 18
gender F
```

Iterate over keys and values:

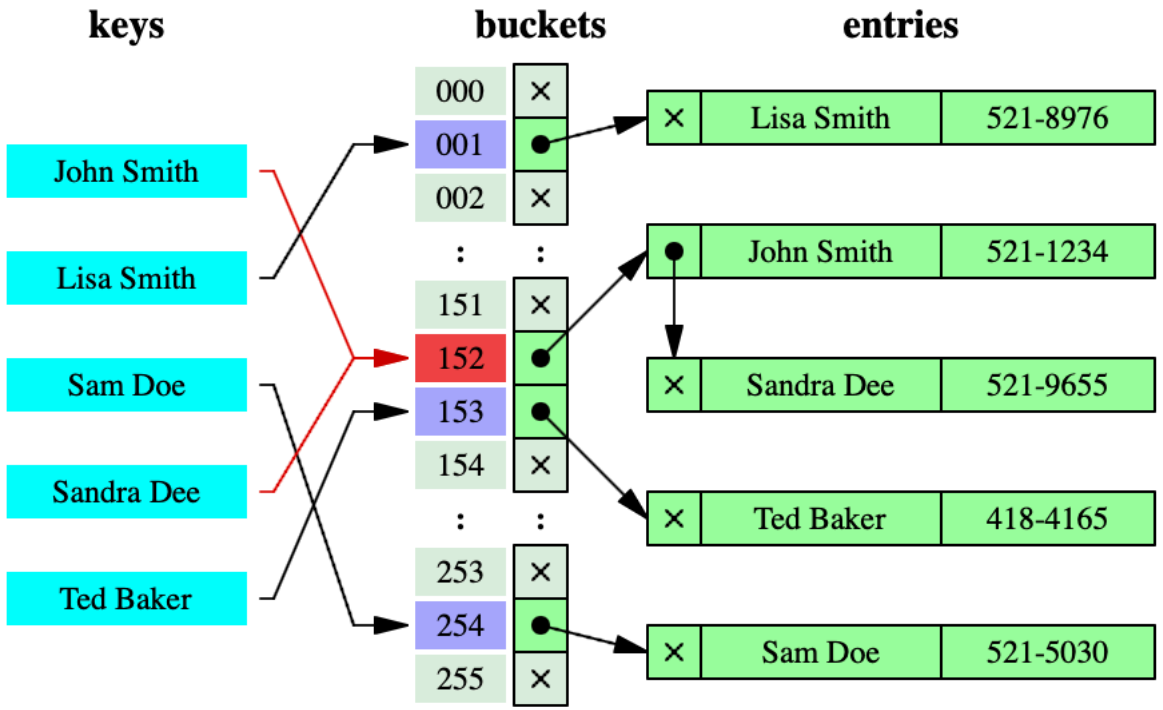
```
for key, value in d.items():
    print(key, value)
```

```
name Alice
age 18
gender F
```

The idea for dictionaries is to try and map to a unique bucket



If the unique mapping fails we are willing to incur a slight performance penalty



Dictionary methods

Python Code	What it does
`d = dict(name='Alice', age=18)`, `d = {'name' : 'Alice', 'age' : 18 }`	Create a dictionary using the `dict` constructor or the "literal" braces notation
`d['name']`, `d.get('name', 'defaultname')`	Access value at key. Second form returns a default if `name` is not in `d`
`d2 = dict(gender='F')`, `d1.update(d2)`	Update from another dictionary
`d['gender'] = 'F'`	Set a value associated with a key
`d.setdefault('gender', 'F')`	If there is a value associated with gender, return it, else **set** that value to default `F` and return it.
`del d['gender']`	delete a key-value pair from the dictionary.
`'gender' in d`	Returns true if the key `gender` is in the dictionary
`d.keys()`	Returns a view over the keys in the dictionary that can be iterated or looped over
`d.values()`	Returns a view over the values in the dictionary which can be iterated or looped over
`d.items()`	Returns a view over pairs (tuples) of type key, value which can be iterated or looped over

Tuples and immutability

Tuples are like lists but they cannot be changed in place. They are often used for storing *different kinds* of data, while lists are used for the same kind. They are **immutable**. Why use them? They are fast! More about this later.

```
tup = ('Alice', 18, 'F')
```

This looks similar to our dictionary, but you cant add any entries to it. Its a *final object*, so to speak.

Why immutability?

- Strings are immutable too!
- Immutability makes things faster.
- Immutable objects can be used as keys in dictionaries!

```
tup = (1, 2, 3)
tup[1] = 4 # replace 2 by 4
```

```
TypeError: 'tuple' object does not support item assignment
```

```
mystr = "Hello World"
mystr[2] = 'k' # replace l with k
```

```
TypeError: 'str' object does not support item assignment
```

String methods

`'capitalize()'`	`'index()'`	`'islower()'`
`'center()'`	`'isalnum()'`	`'isnumeric()'`
`'count()'`	`'isalpha()'`	`'isspace()'`
`'endswith()'`	`'isdecimal()'`	`'istitle()'`
`'find()'`	`'isdigit()'`	`'isupper()'`
`'format()'`	`'lower()'`	`'join()'`
`'replace()'`	`'lstrip()'`	`'ljust()'`

And a lot more …

Data types review

```
['Apple', 'Orange', 'Mango', 'Grape']
```

LIST

```
'Apple'
```

STRING

```
('Apple', 'Orange', 'Mango', 'Grape')
```

TUPLE

```
{'Apple': 5, 'Orange': 12, 'Mango': 20, 'Grape': 3}
```

DICTIONARY

Data types review

Means of Abstraction: Modular Development

- The most important aspect of structured programs is that they are designed in a modular way. These modules are called procedures or functions, and often combined into groups, called libraries. Modular design brings great **productivity enhancement**.
- Firstly, small modules can be coded easily and quickly, with less error
- Secondly, general purpose modules can be re-used, leading to faster development of new programs
- Thirdly, individual modules can be individually tested.

From: Why Functional Programming Matters, by John Hughes

Functions

We want to encapsulate code, that is combine code together so that we can reuse it at multiple places. This is a **function**.

```
from math import pi # importing python builtins
def circle_area(radius):
    area = pi*radius*radius # calculate area
    return area # return the area
```

A function takes 0 or more arguments and returns a value. If `return` is not specified, python sneaks in one for you, its the special value `None`.

Where are functions defined? Importing from modules

A module is a collection of values like π and functions like `sqrt` which have a common purposes. We saw earlier the symbol `pi` being imported. We can also do:

```
from math import sqrt
sqrt(4)
```

returns 2.0 . Or:

```
import math
math.sqrt(4)
```

We can import functions from modules to do our work:

```
from math import sqrt
hypot = lambda x, y : sqrt(x*x + y*y)
hypot(3, 4) # returns 5
```

Where are functions defined? lambda functions

Besides functions using the `def` syntax, you can define your own **anonymous** functions and assign them to variables. Then you can call them with the variable name. These are great for math functions.

Anonymous function: `lambda x: 5*x + 4`

```
square = lambda x: x*x
square(2)
```

gives 4

```
sos = lambda x, y : x*x + y*y
sos(3, 4)
```

gives 25

Where are functions defined? Built-in functions.

Python defines a lot of built-in functions. Examples you have seen are `set`, `list`, `dict`, and `id`. Here's two useful examples.

```
days = ['M', 'T']
for i, day in enumerate(days):
    print(i, day)
```

gives

```
0 M
1 T
```

```
days = ['M', 'T']
letters = ['A', 'B']
for letter, day in zip(letters, days):
    print(letter, day)
```

gives

```
A M
B T
```

Functions can take multiple arguments and return multiple things.

```
def f(a,b):
    return a+b, a-b
print(f(1,2))
```

Returns a tuple: `(3, -1)`

Functions can have default arguments, which can be named. See on right side.

```
def f(a, b=5):
    return a+b, a-b
```

- `f(1,2)` returns `(3, -1)`
- `f(1,b=2)` returns `(3, -1)`: this form is for self-documentation
- `f(1)` returns `(6, -4)`, the default argument has applied.

Input arguments to a function

Function declaration

```
def average(num1, num2, num3):
    ...:     num_sum = num1 + num2 + num3
    ...:     num_values = 3
    ...:     return num_sum / num_values
```

Function call

```
In [19]: average(1,2,3)
Out[19]: 2.0
In [20]: average(num1 =1,num2 = 2,num3 = 3)
Out[20]: 2.0
In [21]: average(1,2,num3=3)
Out[21]: 2.0
```

Python Function Scope

In python, variables are visible in the *scope* they are defined in, and all scopes inside.

The scope of the jupyter notebook is the **global scope**.

Functions can use variables defined in the global scope.

```
c = 1
def f(a, b):
    return a + b + c, a - b + c
```

f(1,2) returns (4, 0)

Variables defined *locally* will shadow globals.

```
c = 1
def f(a, b):
    c = 2
    return a + b + c, a - b + c
```

f(1,2) returns (5, 1)

Python Function Scope

Variables defined *locally* are not available outside.

```
def f(a, b):
    return a + b, a - b
f(1,2)
print(a)
```

Output:

```
NameError: name 'a' is not defined
```

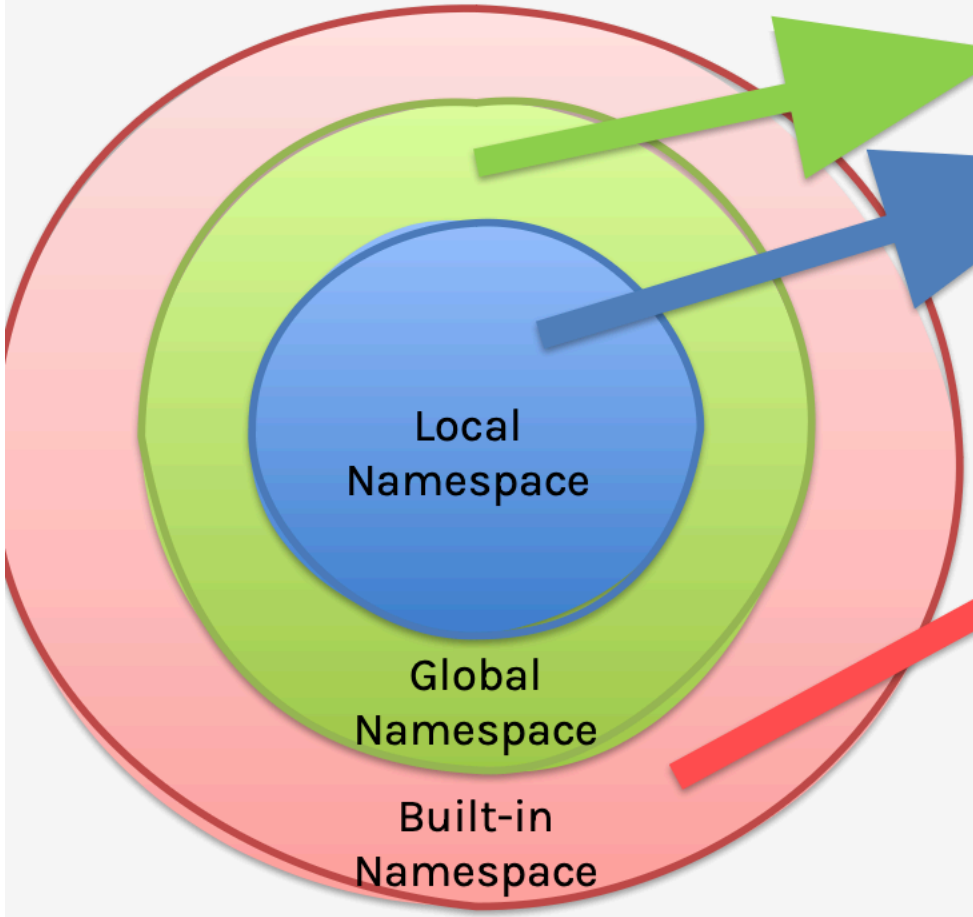
Variables defined in loops (including loop index) are available after.

```
for m in range(2):
    print(m)
print(m)
```

gives us:

```
0
1
1
```

Namespaces




```
def square(x):
    newvar = x**2
    return newvar
square_list = list()
for numbe      range(10):
    squar_    st.append(square(number))
squa    e_list
>>>
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

First Class Functions

Functions can be assigned to variables: `hypot = lambda x, y : sqrt(x*x + y*y)`.

Functions can also be passed to functions and returned from functions.

Functions passed:

```
def mapit(aseq, func):
    return [func(e) for e in aseq]
mapit(range(3), lambda x : x*x) # [0, 1, 4]
```

map and reduce are famous built-in functions.

Functions returned:

```
def soa(f): # sum anything
    def h(x, y):
        return f(x) + f(y)
    return h
sos = soa(lambda x: x*x)
sos(3, 4) # returns 25 like before
```

Pure functions

- **return the same values for the same arguments**, thus being like mathematical functions,
- **no side effects**, by which some state is changed during the execution of the function, and this might not be repeatable, and
- **referential transparent**, by which one may replace the function call by the resultant return values in the code.

Decorators

Decorators use the @ syntax and are a shortcut for a function wrapping another.

```
def factorial(n):
    return n*factorial(n-1)

def check_posint(f):
    def checker(n):
        if n > 0:
            return f(n)
        elif n == 0:
            return 1
        else:
            raise ValueError("Not a positive int")
    return checker

factorial = check_posint(factorial)
print(factorial(4)) # returns 24
print(factorial(-1)) # raises a ValueError
```

```
def check_posint(f):
    def checker(n):
        if n > 0:
            return f(n)
        elif n == 0:
            return 1
        else:
            raise ValueError("Not a positive int")
    return checker

@check_posint
def factorial(n):
    return n*factorial(n-1)

print(factorial(4)) # returns 24
print(factorial(-1)) # raises a ValueError
```

Closures: capturing state

Sometimes we want to capture some state:

```
def soaplusbias(f, bias): # sum anything
    def h(x, y):
        return f(x) + f(y) + bias
    return h
sosplusbias = soaplusbias(lambda x: x*x, 5)
sosplusbias(3, 4)
```

Returns 30

The last line is indentical to before, but we have additionally captured a `bias` from the enclosing scope. The bias is captured when we define `sosplusbias`, but gets used when we call it. This is called a **closure**. It is useful at all sorts of places where state must be captured and used later, as in deep learning callbacks, GUIs, etc

Classes

In Python, a class is simply defined as:

```
class MyClass:
    pass
```

which is equivalent to saying

```
class MyClass(object):
    pass
```

```
class Curry:
    def set_info(self, name, desc):
        self.name = name
        self.desc = desc

    def print(self):
        print(f'{self.name} curry,
              {self.desc} is Yummy!')
```

attribute method self

Mutating State

```
class BankAccount:

    def __init__(self, balance):
        self.balance = balance

    def withdraw(self, amount):
        self.balance = self.balance - amount
```

We make an *instance* of a *class* BankAccount called myaccount with a balance of 100 and withdraw 20 from it. Both the "function" withdraw and the variable representing the balance are called as if they belong to the instance myaccount.

```
myaccount = BankAccount(100)
print(myaccount.balance) # 100
myaccount.withdraw(20)
print(myaccount.balance) # 80
```

- `__init__`: a constructor for the class. This is the function called when we say `BankAccount(100)`.
- Why does `__init__(self, balance)` have 2 arguments then? This is because it is a very special kind of function called a **method**, in which the first argument is the *instance* of the class. By convention, it is always called `self` in python. Thus we will use `BankAccount(balance)` to call the *constructor method*.
- `withdraw(self, amount)`: another *method*. Once again `self` is the existing account object. You can think of your program's `myaccount` instance `withdraw`-ing the `amount` and thus write it `myaccount.withdraw(amount)`, the implicit `self` having been moved to the left of the dot.
- `myaccount.balance`: this is an *instance variable*, some data 'belonging" to the instance, just as the previous method did. Thus we'll use `self.balance` inside the methods to denote it.

Class Variables and Class Methods

How do we share variables and functionality across all instances of a class?

```
class BankAccount:
    max_balance = 1000

    @classmethod
    def make_account(cls, balance):
        if balance <= cls.max_balance:
            return cls(balance)
        else:
            raise ValueError(f"{balance} too large")

    def __init__(self, balance):
        self.balance = balance

    def withdraw(self, amount):
        self.balance = self.balance - amount

ba = BankAccount.make_account(100)
ba.balance # 100
ba.withdraw(20)
ba.balance # 80

bb = BankAccount.make_account(10000) # ValueError: 10000 too large
```

- `max_balance` is a **class variable** since this max value needs to be shared by all accounts. Note it is not preceded with a `self`.
- The **classmethod** `make_account`, announced thus by *decorating* it with `@classmethod`, takes the class, NOT the instance, as its "implicit" (as in not written) first argument, moving the class over to the left side of the dot.
- It calls the constructor as `cls(balance)` if the class variable `cls.max_balance` is reasonable.

Pseudo-code is an abstraction for real code. Lets see some real code!

Lets see even more python

https://github.com/hult-cm3-rahul/LearningPython