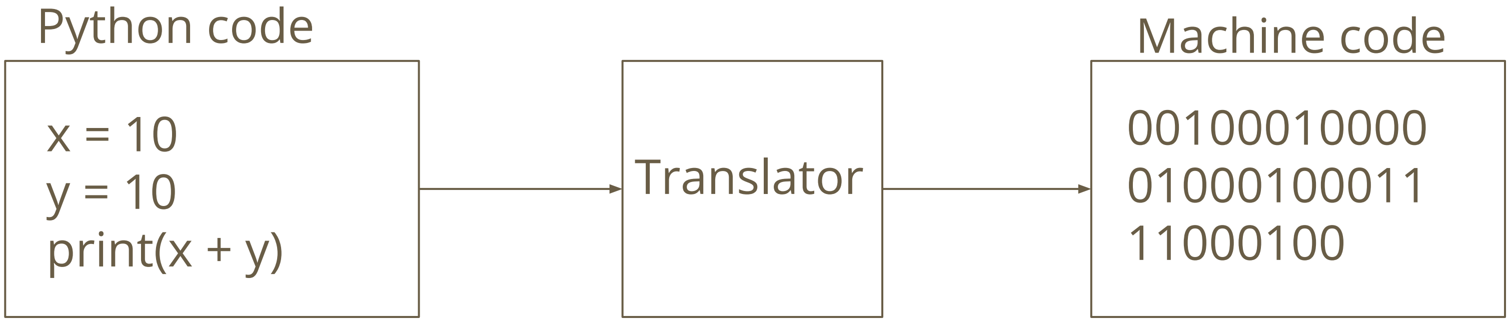
WEEK 1

Programming languages

- A language is a medium for communication
- Programming languages communicate computational instructions
- Translate high level language to low level language
- Java
- Binary language(0’s and 1’s)
- Compilers and Interpreters



Styles of programming

- Imperative vs declarative
- Imperative
  - How to compute
  - Step by step instructions on what is to be done
- Declarative
  - What the computation should produce
  - Often exploit inductive structure, express in terms of smaller computations
  - Typically avoid using intermediate variables
  - Combination of small transformations — functional programming

Imperative vs Declarative Programming, by example, ...

- Sum of squares of even numbers upto `n`

- Imperative (in Python)

```
def sumsquareeven(n):
  mysum = 0
  for x in range(n+1):
    if x%2 == 0:
      mysum = mysum + x*x
  return(mysum)
```

- Can code functionally in an imperative language!

- Helps identify natural units of (reusable) code

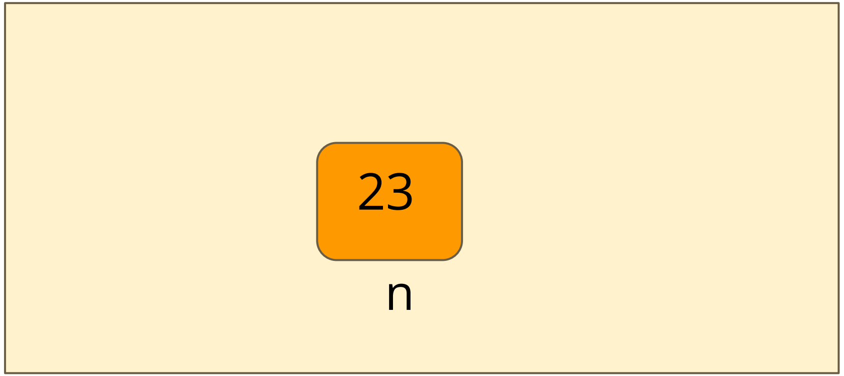
- Declarative (in Python)

```
def even(x):
  return(x%2 == 0)

def square(x):
  return(x*x)

def sumsquareeven(n):
  return(
    sum(map(square,
            filter(even,
                   range(n+1)))))
```

Names,values,types

- Variables

int n=23;



Dynamic vs static typing

- Every variable we use has a type
- How is the type of a variable determined?
- Python determines the type based on the current value
  - Dynamic typing — names derive type from current value
  - `x = 10` — `x` is of type `int`
  - `x = 7.5` — now `x` is of type `float`
  - An uninitialized name as no type
- Static typing — associate a type in advance with a name
  - Need to declare names and their types in advance value
  - `int x`, `float a`, ...
  - Cannot assign an incompatible value — `x = 7.5` is no longer legal

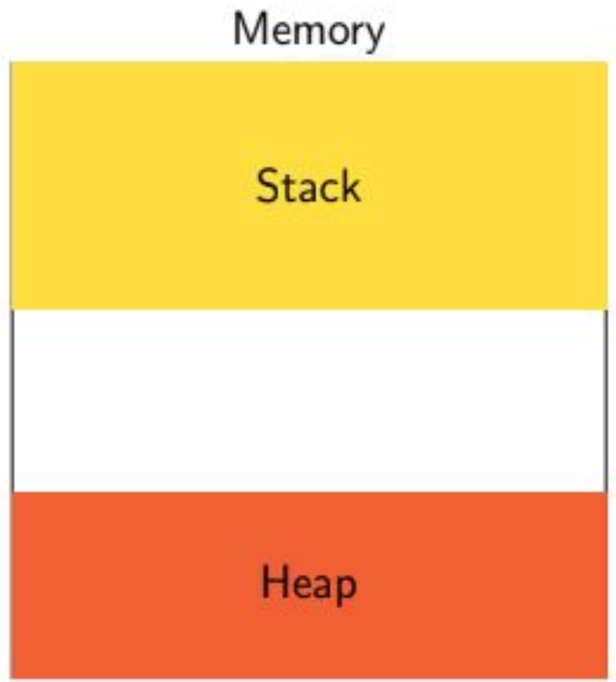
Collections,Abstract data types,Object oriented programming

- Collections like array,list etc..

2	3	4	6

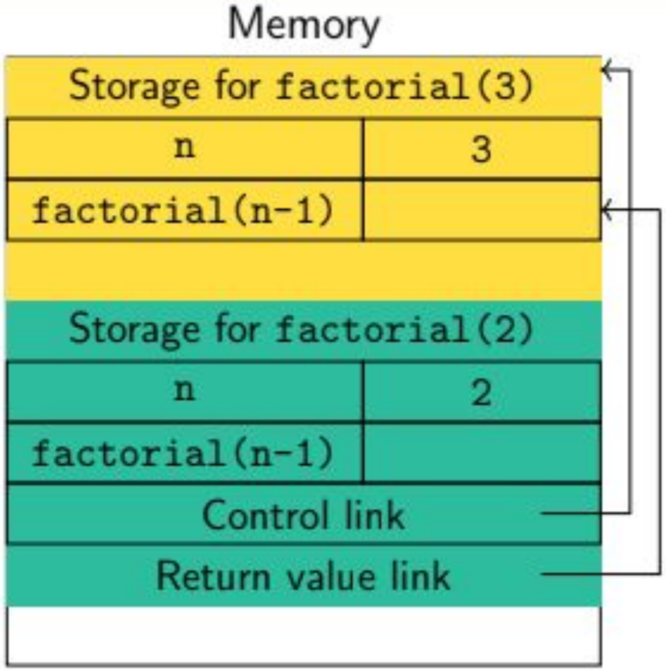
- Abstract data types
    - Stack ADT,Queue ADT etc
    - Structured collection with fixed interface
    - Stack is a sequence, but only allows push and pop
    - Separate implementation from interface

Memory management


Memory
Stack
Heap

Memory stack

- Each function needs storage for local variables
- Create activation record when function is called
- Activation records are stacked
  - Popped when function exits
  - Control link points to start of previous record
  - Return value link tells where to store result
- Scope of a variable
  - Variable in activation record at top of stack
  - Access global variables by following control links
- Lifetime of a variable
  - Storage allocated is still on the stack



- Call `factorial(3)`
- `factorial(3)` calls `factorial(2)`

- Scope of variable

  The part of the program in which the **variable** can be accessed.

- Lifetime of variable

  It indicates how long the variable stays alive in the memory.

Passing arguments to functions

- When a function is called, arguments are substituted for formal parameters

```
def f(a,l):          x = 7                a = x
    ...              myl = [8,9,10]       l = myl
    ...              f(x,myl)             ... code for f() ...
```

- Parameters are part of the activation record of the function
  - Values are populated on function call
  - Like having implicit assignment statements at the start of the function
- Two ways to initialize the parameters
  - Call by value — copy the value
    - Updating the value inside the function has no side-effect
  - Call by reference — parameter points to same location as argument
    - Can have side-effects
    - Be careful: can update the contents, but cannot change the reference itself

Managing heap storage

- On the stack, variables are deallocated when a function exits
- How do we "return" unused storage on the heap?
  - After deleting a node in a linked list, deleted node i now dead storage, unreachable
- Manual memory management
  - Programmer explicitly requests and returns heap storage
    - `p = malloc(...)` and `free(p)` in C
  - Error-prone — memory leaks, invalid assignments
- Automatic garbage collection (Java, Python, . . .)
  - Run-time environment checks and cleans up dead storage — e.g., mark-and-sweep
    - Mark all storage that is reachable from program variables
    - Return all unmarked memory cells to free space
  - Convenience for programmer vs performance penalty

Modularity

Stepwise refinement

- Begin with a high level description of the task
- Refine the task into subtasks
- Further elaborate each subtask
- Subtasks can be coded by different people
- Program refinement — focus on code, not much change in data structures

```
begin
  print first thousand prime numbers
end
```

```
begin
  declare table p
  fill table p with first thousand primes
  print table p
end
```

```
begin
  integer array p[1:1000]
  for k from 1 through 1000
    make p[k] equal to the kth prime number
  for k from 1 through 1000
    print p[k]
```

Data refinement

- Banking application
  - Typical functions: `CreateAccount()`, `Deposit()/Withdraw()`, `PrintStatement()`
- How do we represent each account?
  - Only need the current balance
  - Overall, an array of balances
- Refine `PrintStatement()` to include `PrintTransactions()`
  - Now we need to record transactions for each account
  - Data representation also changes
  - Cascading impact on other functions that operate on accounts

Modular software development

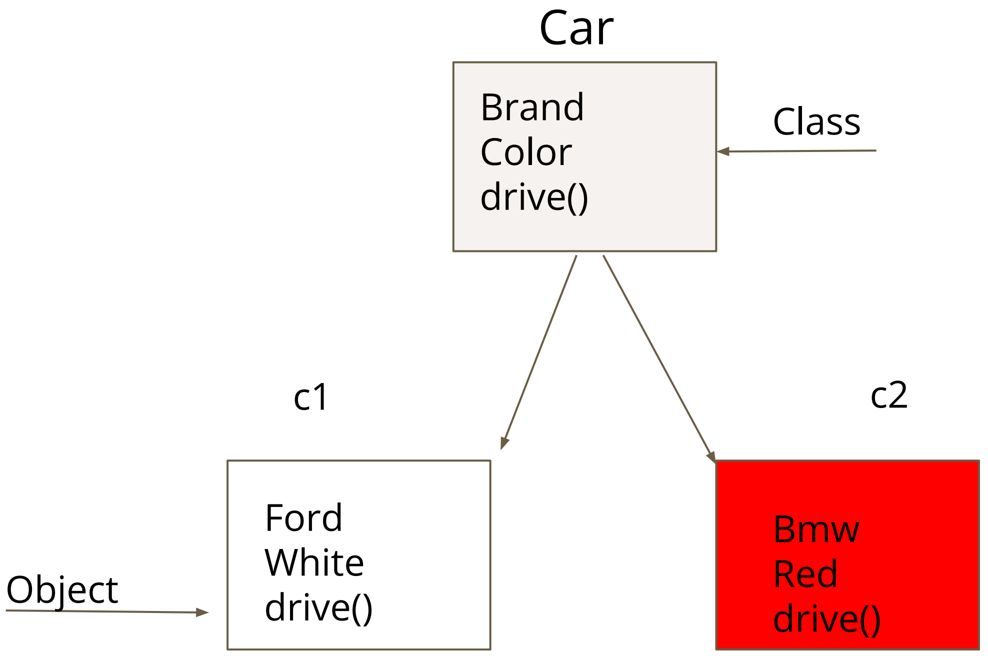
- Use refinement to divide the solution into components
- Build a prototype of each component to validate design
- Components are described in terms of
  - Interfaces — what is visible to other components, typically function calls
  - Specification — behaviour of the component, as visible through interface
- Improve each component independently, preserving interface and specification
- Simplest example of a component: a function
  - Interfaces — function header, arguments and return type
  - Specification — intended input-output behaviour
- Main challenge: suitable language to write specifications
  - Balance abstraction and detail, should not be another programming language!
  - Cannot algorithmically check that specification is met (halting problem!)

Object oriented programming

Programming with objects

- Object are like abstract datatypes
  - Hidden data with set of public operations
  - All interaction through operations — messages, methods, member-functions, ...
- Class
  - Template for a data type
  - How data is stored
  - How public functions manipulate data
- Object
  - Concrete instance of template
  - Each object maintains a separate copy of local data
  - Invoke methods on objects — send a message to the object

```python
class Car:
	def __init__(self,brand,color) :
		self.brand=brand
		self.color=color
	def drive(self):
		print('driving')
c1=Car('Ford','White')
print(c1.brand)
print(c1.color)
c2=Car('Bmw','Red')
print(c2.brand)
print(c2.color)
```



Objects

- An object is like an abstract datatype
  - Hidden data with set of public operations
  - All interaction through operations — messages, methods, member-functions, . . .
- Uniform way of encapsulating different combinations of data and functionality
  - An object can hold single integer — e.g., a counter
  - An entire filesystem or database could be a single object
- Distinguishing features of object-oriented programming
  - Abstraction
  - Subtyping
  - Dynamic lookup
  - Inheritance

Programming language support for abstraction

- Control abstraction
  - Functions and procedures
  - Encapsulate a block of code, reuse in different contexts
- Data abstraction
  - Abstract data types (ADTs)
  - Set of values along with operations permitted on them
  - Internal representation should not be accessible
  - Interaction restricted to public interface
    - For example, when a stack is implemented as a list, we should not be able to observe or modify internal elements

Inheritance

- Re-use of implementations
- Example: different types of employees
  - `Employee` objects store basic personal data, date of joining
  - `Manager` objects can add functionality
    - Retain basic data of `Employee` objects
    - Additional fields and functions: date of promotion, seniority (in current role)
- Usually one hierarchy of types to capture both subtyping and inheritance
  - `A` can inherit from `B` iff `A` is a subtype of `B`
- Philosophically, however the two are different
  - Subtyping is a relationship of interfaces
  - Inheritance is a relationship of implementations

Subtyping

- A subtype is a specialization of a type
- If A is a subtype of B, wherever an object of type B is needed, an object of type A can be used.
- Every object of type A is also an object of type B
- Think subset — if $X \subseteq Y$, every $x \in X$ is also in Y
- If f() is a method in B and A is a subtype of B, every object of A also supports f().
- Implementation of f() can be different in A.
- Dequeue is subtype of stack,queue

- Define `Square` to be a subtype of `Rectangle`
  - Different constructor
  - Same instance variables
- The following is legal

```
s = Square(5)
a = s.area()
p = s.perimeter()
```

  - `Square` inherits definitions of `area()` and `perimeter()` from `Rectangle`

```
class Rectangle:
  def __init__(self,w=0,h=0):
    self.width = w
    self.height = h

  def area(self):
    return(self.width*self.height)

  def perimeter(self):
    return(2*(self.width+self.height))

class Square(Rectangle):
  def __init__(self,s=0):
    self.width = s
    self.height = s
```

- Can change the instance variable in `Square`
  - `self.side`
- The following gives a run-time error

```
s = Square(5)
a = s.area()
p = s.perimeter()
```

  - `Square` inherits definitions of `area()` and `perimeter()` from `Rectangle`
  - But `s.width` and `s.height` have not been defined!
  - Subtype is not forced to be an extension of the parent type

```
class Rectangle:
  def __init__(self,w=0,h=0):
    self.width = w
    self.height = h

  def area(self):
    return(self.width*self.height)

  def perimeter(self):
    return(2*(self.width+self.height))


class Square(Rectangle):
  def __init__(self,s=0):
    self.side = s
```

Subtyping vs inheritance

- A deque is a double-ended queue
  - Supports `insert-front()`, `delete-front()`, `insert-rear()` and `delete-rear()`
- We can implement a stack or a queue using a deque
  - Stack: use only `insert-front()`, `delete-front()`,
  - Queue: use only `insert-rear()`, `delete-front()`,
- `Stack` and `Queue` inherit from `Deque` — reuse implementation
- But `Stack` and `Queue` are not subtypes of `Deque`
  - If `v` of type `Deque` points an object of type `Stack`, cannot invoke `insert-rear()`, `delete-rear()`
  - Similarly, no `insert-front()`, `delete-rear()` in `Queue`
- Interfaces of `Stack` and `Queue` are not compatible with `Deque`
  - In fact, `Deque` is a subtype of both `Stack` and `Queue`

Dynamic lookup

- Whether a method can be invoked on an object is a static property — type-checking
- How the method acts is a dynamic property of how the object is implemented
  - In the simulation queue, all events support a simulate method
  - The action triggered by the method depends on the type of event
  - In a graphics application, different types of objects to be rendered
  - Invoke using the same operation, each object "knows" how to render itself
- Different from overloading
  - Operation `+` is addition for `int` and `float`
  - Internal implementation is different, but choice is determined by static type
- Dynamic lookup
  - A variable `v` of type `B` can refer to an object of subtype `A`
  - Static type of `v` is `B`, but method implementation depends on run-time type `A`