Introduction

Type	📒 Lecture
Date	@December 27, 2021
Lecture #	1
Lecture URL	https://youtu.be/3a1FXBR6QXY
Notion URL	https://21f1003586.notion.site/Introduction-95ec26f2dc934c96a69762cad8582900
Week #	1

Programming Languages

- A language is a medium for communication
- Programming langauges communicate computational instructions
- Originally, directly connected to architecture
  - Memory locations store values, registers allow arithmetic
  - Load a value from memory location M into register R
  - Add the contents of the register R_1 and R_2 and store the result back in R_1
  - Write the value in R_1 to memory location M'
- Tedious and Error prone process

Abstraction

- Abstractions used in computational thinking
  - Assigning values to named variables
  - Conditional execution
  - Iteration
  - Functions/Procedures, recursion
  - Aggregate data structures — arrays, list, dictionaries
- Express such ideas in the programming language
  - Translate “high level” programming language to “low level” machine language
  - Compilers, interpreters

- Trade off expressiveness for efficiency
    - Less control over how code is mapped to the architecture
    - But fewer errors due to mismatch between intent and implementation

Styles of programming

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

Imperative vs Declarative Programming, by example

- Add values in a list
- **Imperative** (in Python)

```
def sum_list(l):
  sum = 0
  for x in l:
    sum += x
  return sum
```

- Intermediate values `sum, x`
- Explicit iteration to examine each element in the list
- **Declarative** (in Python)

```
def sum_list(l):
  if l == []:
    return 0
  else:
    return l[0] + sum_list(l[1:])
```

- Describe the desired output by induction
    - Base case $\rightarrow$ Empty list has sum 0
    - Inductive step $\rightarrow$ Add the first element to the sum of the rest of the list
- No intermediate variables

- Sum of squares of even numbers upto `n`
- **Imperative** (in Python)

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

- We can code functionally in an imperative language
- Helps us identify natural units of (reusable) code
- **Declarative** (in Python)

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

def square(x):
  return x * x

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

Names, types and values

- Internally, everything is stored as a sequence of bits
- No difference between data and instructions, let alone numbers, characters, booleans
  - For a compiler or interpreter, our code is its data
- We impose a notion of type to create some discipline
  - Interpret bit strings as "high level" concepts
  - Nature and range of allowed values
  - Operations that are permitted on these values
- Strict type-checking helps catch bugs early
  - Incorrect expression evaluation — like dimension mismatch in science
  - Incorrect assignment — expression value does not match variable type

Abstract datatypes, object-oriented programming

- Collections are important
  - Arrays, lists, dictionaries
- Abstract data types
  - Structured collection with fixed interface
  - Stack, for example, is a sequence but only allows `push` and `pop`
  - Separate implementation from interface
    - Priority queue allows `insert` and `delete-max`
    - Can implement a priority queues using sorted or unsorted lists, or using a heap
- Object-Oriented Programming
  - Focus on data types
  - Functions are invoked through the object rather than passing data to the functions
  - In python, `my_list.sort()` vs `sorted(my_list)`

What is yet to come ...

- Explore concepts in programming languages
  - Object-oriented programming
  - Exception handling, concurrency, event-driven programming
- Use Java as the illustrative language
  - Imperative, object-oriented
  - Incorporates almost all the features
- Discuss design decisions where relavant
  - Every language makes some compromises
- Understand and appreciate why there is a zoo of programming languages out there, *lol*
- And why new ones are still being created

Types

Type	📒 Lecture
Date	@December 27, 2021
Lecture #	2
Lecture URL	https://youtu.be/0Gl9ygUk4K8
Notion URL	https://21f1003586.notion.site/Types-e3dd9b780c9d410086c44c63bffb6356
Week #	1

The role of types

- Interpreting data stored in binary in a consistent manner
    - View sequence of bits as integers, floats, characters, ...
    - Nature and range of allowed values
    - Operations that are permitted on these values
- Naming concepts and structuring our computation
    - Especially at a higher level
    - `Point` vs `(Float, Float)`
    - Banking applications → accounts of different types, customers, ...
- Catching bugs early
    - Incorrect expression evaluation
    - Incorrect assignment

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*** → derive type from the current value
    - `x = 10` — `x` is of type `int`
    - `x = 7.5` — now `x` is of type `float`

- An uninitialized name has no type
- ***Static typing*** → associate a type in advance with a name
  - Need to declare names and their types in advance
  - `int x, float a, ...`
  - Cannot assign an incompatible value — `x = 7.5` is illegal

- It is difficult to catch errors, such as typos

```
def factors(n):
  factorlist = []
  for i in range(1, n + 1):
    if n % i == 0:
      factorlst = factorlist + [i]  # Typo here!
  return factorlist
```

- Empty user defined objects
  - Linked list is sequence of objects of type `Node`
  - Convenient to represent empty linked list by `None`
  - Without declaring type of `l`, Python cannot associate type after `l = None`

Types of organizing concepts

- Even simple type "synonyms" can help clarify code
  - 2D point is a pair `(float, float)`, 3D point is triple `(float, float, float)`
  - Create new type names `point2d` and `point3d`
  - These are synonyms for `(float, float)` and `(float, float, float)`
  - Makes the intent more transparent when writing, reading and maintaining code
- More elaborate types — abstract datatypes and object-oriented programming
  - Consider a banking application
  - Data and operations related to accounts, customers, deposits, withdrawals, transfers
  - Denote accounts and customers as separate types
  - Deposits, withdrawals, transfers can be applied to accounts, not to customers
  - Updating personal details applies to customers, not accounts

Static analysis

- Identify errors as early as possible — saves cost & effort
- In general, compilers cannot check that a program will work correctly
  - Halting problem — Alan Turing
- With variable declarations, compilers can detect type errors at compile time - ***static analysis***
  - Dynamic typing would catch these errors only when the code runs
  - Executing code also shows down due to simultaneous monitoring for type correctness
- Compilers can also perform optimizations based on static analysis
  - Re-order statements to optimize reads and writes
  - Store previously computed expressions to re-use later

Summary

- Types have many uses
  - Making sense of arbitrary bit sequences in memory
  - Organizing concepts in our code in a meaningful way
  - Helping compilers catch bugs early, optimize compiled code

- Some languages also support automatic type inference
  - Deduce the types of variable statically, based on the context in which they are used
  - `x = 7` followed by `y = x + 15` implies `y` must be `int`
  - If the inferred type is consistent across the program, everything will go fine

Memory Management

Type	📒 Lecture
Date	@December 27, 2021
Lecture #	3
Lecture URL	https://youtu.be/b4nsGWXNm2c
Notion URL	https://21f1003586.notion.site/Memory-Management-bf48fedf690e4df5991b9b7e9b9e1480
Week #	1

Keeping track of variables

- Variables store intermediate values during computation
  - Typically these are local to a function
  - Can also refer to global variables outside the function
  - Dynamically created data, like nodes in a list
- Scope of a variable
  - When the variable is available for use
  - In the following code, the `x` in `f()` is not in scope withing call to `g()`

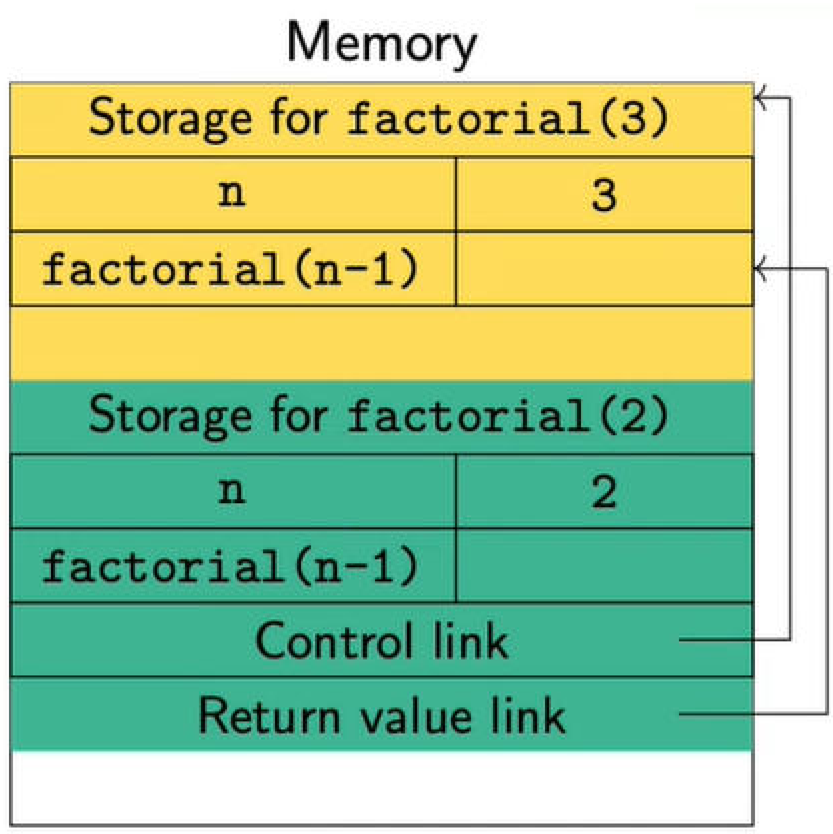
```
def f(l):
    ...
    for x in l:
      y = y + g(x)
    ...
```

```
def g(m):
    ...
    for x in range(m):
      ...
```

- Lifetime of a variable
  - How long the storage remains allocated
  - Above, the lifetime of `x` in `f()` is till `f()` exists
  - “Hole in the scope” — variable is alive but not in scope

Memory stack

- Each function needs storage for local variables
- Create ***activation record*** when function is called
- Activation record are stacked
  - Popped when function exits
  - **Control link** points to start of previous record
  - **Return value link** tells where to store result



Call `factorial(3)`

`factorial(3)` calls `factorial(2)`

- 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

Passing arguments to a function

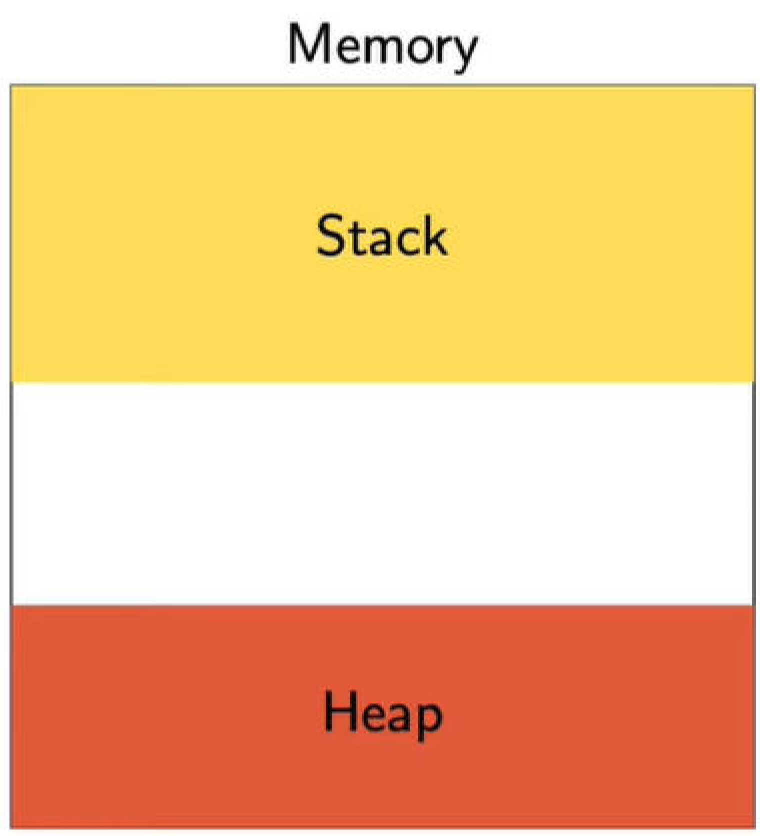
- When a function is called, arguments are substituted for formal parameters

```
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 not side effect
  - ***Call by reference*** → parameter points to same location as the argument
    - Can have side-effects
    - It can update the contents, but cannot change the reference itself

Heap

- Function that inserts a value in a linked list
  - Storage for new node allocated inside function
  - Node should persist after function exits
  - Cannot be allocated within activation record
- We need a separate storage for the persistent data
  - Dynamically allocated vs statically declared
  - Usually called the heap
    - Not the same as heap data structure
  - Conceptually, allocate heap storage from “opposite” end with respect to the stack



- Heap store outlives activation record
  - Access through some variable that is in scope

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 is now dead storage, unreachable
- Manual memory management
  - Programmer explicitly requests and returns heap storage
    - `p = malloc(...)` and `free(p)` in C language
  - Error-prone — memory leaks, invalid assignments
- Automatic garbage collection (Java, Python, ...)
  - Run-time environment checks and cleans up dead storage
    - Mark all storage that is reachable from program variables
    - Return all unmarked memory cells to free space
  - Convenience for programmer vs peformance penalty

Summary

- Variables have ***scope*** and ***lifetime***
  - Scope → whether the variable is available in the program

    - Lifetime → whether the storage is still allocated
- Activation records for functions are maintained as a stack
    - Control link points to previous activation record
    - Return value link tells where to store results
- Heap is used to store dynamically allocated data
    - Outlives activation record of function that created the storage
    - Need to be careful about deallocating heap storage
    - Explicit deallocation vs automatic garbage collection

Abstraction and Modularity

Type	Lecture
Date	@December 27, 2021
Lecture #	4
Lecture URL	https://youtu.be/8ciDI5cUhS
Notion URL	https://21f1003586.notion.site/Abstraction-and-Modularity-d39c0b22b59a4238b7173fef6074ebdc
Week #	1

Stepwise Refinement

- Begin with a high level description of the task
- Refine the tasks 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 amount?
    - Only need the current balance
    - Overall, an array of balance

- 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 the 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!)

Programming language support for abstraction

- Control abstraction
  - Functions and procedures
  - Encapsulate a block of code, re-use 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 the internal elements
- Object-Oriented programming
  - Organize ADTs in a hierarchy
  - Implicit reuse of implementations — subtyping, inheritance

Summary

- Solving a complex task requires breaking it down into manageable components
  - ***Top-down:*** refine the tasks into subtasks
  - ***Bottom-up:*** combine simple building blocks
- Modular description of components
  - Interface and specification
  - Build prototype implementation to validate design
  - Reimplement the components independently preserving interface and specification
- Programming Language support for abstraction
  - Control flow: functions and procedures
  - Data: Abstract data types, object-oriented programming

Object-Oriented Programming

Type	📒 Lecture
Date	@December 27, 2021
Lecture #	5
Lecture URL	https://youtu.be/NmYcNMPUlzY
Notion URL	https://21f1003586.notion.site/Object-Oriented-Programming-3bb75c2bcc92484d96ff97ef71bffed9
Week #	1

Objects

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

History of object-oriented programming

- Objects first introduced in Simula — simulation language, $1960s$
- Event-based simulation follows a basic pattern
    - Maintain a queue of events to be simulated
    - Simulate the event at the head of the queue
    - Add all events it spawns to the queue
- ***Challenges***

  - Queue must be well-types, yet hold all types of events
  - Use a generic simulation operation across different types of events
    - Avoid elaborate checking of cases

Abstraction

- Objects are similar to abstract datatypes
  - Public interface
  - Private implementation
  - Changing the implementation should not affect interactions with the object
- Data-centric view of programming
  - Focus on what data we need to maintain and manipulate
- Recall that stepwise refinement could affect both code and data
  - Tying methods to data makes this easier to coordinate
  - Refining data representation naturally tied to updating methods that operate on the data

Subtyping

- Recall the Simula event queue
  - A well-typed queue holds values of a fixed type
  - In practice, the queue holds different types of objects
  - How can this be reconciled?
- Arrange types in a hierarchy
  - 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`

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 runtime type `A`

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 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`

Summary

- Objects are like abstract datatypes
- Uniform way of encapsulating different combinations of data and functionality
- Distinguishing features of object-oriented programming
  - Abstraction
    - Public interface, private implementation, like ADTs
  - Subtyping
    - Hierarchy of types, compatibility of interfaces
  - Dynamic lookup
    - Choice of method implementation is determined at runtime
  - Inheritance
    - Reuse of implementations

Classes and Objects

Type	📒 Lecture
Date	@December 27, 2021
Lecture #	6
Lecture URL	https://youtu.be/TJC0WhS6FNo
Notion URL	https://21f1003586.notion.site/Classes-and-Objects-a6d38f8b11b44c269cb11bb0eb209107
Week #	1

Programming with Objects

- Objects are like abstract datatypes
  - Hidden data with set of public operations
  - All interactions through operations — methods ...
- ***Class***
  - Template for a data type
  - How a data is stored
  - How public functions manipulate the data
- ***Object***
  - Concrete instance of the above mentioned template
  - Each object maintains its separate copy of local data
  - Invoke methods on objects — Equivalent to "***send a message to the object***"

Example: 2D points

- A point has coordinates `(x, y)`
  - Each point object stores its own internal values `x` and `y` — these are called instance variables
  - For a point `p`, the local values are `p.x` and `p.y`
  - `self` is a special name referring to the current object — `self.x, self.y`
- When we create an object, we need to set it up
  - Implicitly call a constructor function with a fixed name

- In Python, constructor is called `__init__()`
- Parameters are used to set up internal values
- In Python, the first parameter is always `self`

```
class Point:
  def __init__(self, a=0, b=0):
    self.x = a
    self.y = y
```

Adding methods to a class

- Translation: shift a point by $(\Delta x, \Delta y)$
  - $(x, y) \mapsto (x + \Delta x, y + \Delta y)$
  - Update instance variables

  ```
  # This will go inside the Point class
  def translate(self, dx, dy):
    self.x += dx
    self.y += dy
  ```

- Distance from the origin
  - $d = \sqrt{x^2 + y^2}$
  - Does not update instance variables
  - state of object is unchanged

  ```
  # This will go inside the Point class
  def odistance(self):
    import math
    d = math.sqrt(self.x*self.x + self.y*self.y)
    return d
  ```

Changing the internal implementation

- Polar coordinates: (r, θ), not (x, y)
  - $r = \sqrt{x^2 + y^2}$
  - $\theta = tan^{-1}(y/x)$
- Distance from origin is just r

  ```
  import math
  class Point:
    def __init__(self, a=0, b=0):
      self.r = math.sqrt(a*a + b*b)
      if a == 0:
        self.theta = math.pi/2
      else:
        self.theta = math.atan(b/a)

    def odistance(self):
      return self.r
  ```

- Translation
  - Convert (r, θ) to (x, y)
  - $x = rcos\theta, y = rsin\theta$
  - Recompute r, θ from $(x + \Delta x, y + \Delta y)$
- Interface has not changed
  - User need not be aware whether representation is (x, y) or (r, θ)

  ```
  def translate(self, dx, dy):
    x = self.r * math.cos(self.theta)
  ```

```
    y = self.r * math.sin(self.theta)
    x += dx
    y += dy
    self.r = math.sqrt(x*x + y*y)
    if x == 0:
      self.theta = math.pi/2
    else:
      self.theta = math.atan(y/x)
```

Abstraction

- Users of our code should not know whether `Point` uses `(x, y)` or `(r, theta)`
  - Interface remains indentical, or should remain identical
  - Even constructor is the same
- Python allows direct access to instance variables from outside the class

  `p = Point(5, 7)`

  `p.x = 4  # Point is now (4, 7)`
  - This defeats the purpose of abstraction
  - Changing the internal implementation of `Point` can have impact on other code
    - Usually called breaking changes

```
class Point:
  def __init__(self, a=0, b=0):
    self.x=a
    self.y=b

class Point:
  def __init__(self, a=0, b=0):
    self.r = math.sqrt(a*a + b*b)
    if a==0:
      self.theta = math.pi/2
    else:
      self.theta = math.atan(b/a)
```

Subtyping and inheritance

- Define `Square` to be a subtype of `Rectangle`
  - Different constructor
  - Same instance variables

  ```
  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
  ```

- The following code is legal

  `s = Square(5)`

  `a = s.area()`

  `p = s.perimeter()`
  - `Square` inherits definitions from `area()` and `perimeter()` from `Rectangle`

Subtyping and inheritance

- Can change the instance variable in `Square`
  - `self.side`

```
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
```

- The following code gives a runtime 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

---

- Subclasses and parent class are usually developed separately
- Implementor of `Rectangle` changes the instance variables

```
class Rectangle:
  def __init__(self, w=0, h=0):
    self.wd = w
    self.ht = h

  def area(self):
    return self.wd * self.ht

  def perimeter(self):
    return 2 * (self.wd + self.ht)

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

- The following code gives a runtime error

  `s = Square(5)`

  `a = s.area()`

  `p = s.perimeter()`

  - `Square` constructor sets `s.width` and `s.height`
  - But the instance variable names have changed

- Need a mechanism to hide private implementation details
  - Declare component private or public
- Working within privacy constraints
  - Instance variables `wd` and `ht` of `Rectangle` are private
  - How can the constructor for `Square` set these private variables?
  - `Square` does (and should) not know the names of the private instance variables
- Need to have elaborate declarations
  - Type and visibility of variables
- Static type checking catches errors early

Summary

- A class is a template describing the instance variables and methods for an abstract datatype

- An object is a concrete instance of a class
- We should separate the public interface from the private implementation
- Hierarchy of class to implement subtyping and inheritance
- A language like Python has no mechanism to enforce privacy etc.
    - Can illegally manipulate private instance variables
    - Can introduce inconsistencies between subtypes and parent types
- Use strong declarations to enfore privacy, types
    - Do not rely on programmer disciplines
    - Catch bugs early through type checking