Niklaus Wirth

- After receiving his Ph.D. in 1963 from Berkeley and then teaching at Stanford, Niklaus Wirth accepted a professorship at the Eidgenössische Technische Hochschule in Zurich (invariably shortened to ETH).
- Wirth designed several important programming languages, including Algol-W, Pascal, Euler, Modula, and Oberon. Pascal was for many years the primary language used to teach introductory computer science and was the first programming language with full support for records.
- Niklaus Wirth received the ACM Turing Award in 1984.

The Idea of a Record

- Records are a very old idea, dating back at least to the first Babylonian dynasty from the 19th to the 16th centuries BCE. These records were written in cuneiform on clay tablets.
- To give you a sense of age, this tablet contains part of the Gilgamesh epic that some of you read last year in HUM 110.

Employees at Scrooge and Marley

- The example in the text uses the literary example of Charles Dickens’s *A Christmas Carol* to represent the payroll data for the two-employee firm Scrooge and Marley, in which each record contains the name of the employee, the job title, and the weekly salary in shillings:

<table>
<thead>
<tr>
<th>name</th>
<th>title</th>
<th>salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Ebenezer Scrooge&quot;</td>
<td>&quot;founder&quot;</td>
<td>1000</td>
</tr>
<tr>
<td>&quot;Bob Cratchit&quot;</td>
<td>&quot;clerk&quot;</td>
<td>15</td>
</tr>
</tbody>
</table>
Tuples

• In Python, the simplest strategy for representing a record uses the built-in Python data type `tuple`, which is an ordered, immutable sequence of values. The term comes from words like quintuple and sextuple that denote fixed-size collections but also encompasses pairs, triples, and quadruples that don’t actually end with `tuple`.
• In terms of their implementation, tuples are like lists except for their immutability, although they are used very differently.
• To use tuples effectively, you need to think of them as being more like records than like lists.

Tuples in Python

• You can create a tuple in Python by enclosing a list of its elements in parentheses. For example, the statement
  ```python
employee = ("Bob Cratchit", "clerk", 15)
  ```
creates a three-element tuple containing the employment data for Bob Cratchit and assigns it to the variable `employee`.
• A Python tuple is stored internally in precisely the same way that a list is. Thus, the internal representation of the variable `employee` looks like this:
  ```
  ["Bob Cratchit", "clerk", 15]
  ```

Selection in Tuples

• Just as with lists, you can select elements from a tuple using integer indices, but doing so has the disadvantage that the numeric order is usually not relevant to the record model, which uses names instead of indices for selection.
• If you use tuples, you can make your programs more readable using destructuring assignment, which breaks up a triple into named components, as follows:
  ```python
  name, title, salary = employee
  ```
• Although modern versions of Python provide support for named tuples that associate names with the immutable fields, those structures are beyond the scope of CSCI 121. A more general strategy is to define new classes that represent objects with a particular structure.

Review: Classes and Objects

• When I introduced the Portable Graphics Library early in the semester, I emphasized the distinction between classes and objects:
  - A class is a pattern that defines the structure and behavior of values with that particular type.
  - An object is a value that belongs to a class. A single class can be used to create any number of objects, each of which is an instance of that class.
• The Portable Graphics Library, for example, defines a `GRect` class. In the Breakout project, you created many different instances of the `GRect` class—one for each brick and one for the paddle—each of which was a separate object.

The Purposes of Objects

• Python uses the concepts of objects and classes to achieve at least three different goals:
  - Aggregation. Objects make it possible to represent collections of independent data values as a single unit. Such collections are traditionally called records.
  - Encapsulation. Classes make it possible to store data values together with the operations that manipulate them. In Python, the data values are called attributes, and the operations are called methods.
  - Inheritance. Class hierarchies make it possible for a class that shares some attributes and methods from a previously defined class to inherit those definitions without explicitly repeating the definitions.
• The lectures this week will examine each of these aspects of object-oriented programming in turn.
Classes as Templates
- Since they share the same attributes, it is natural to regard the two employees at Scrooge and Marley as instances of the same class. The class itself is defined by the template:

- Class definitions in Python, however, are sufficiently complex that it helps to start by using an empty template that creates blank-slate objects and then filling in the necessary fields.

Defining a Blank-Slate Class
- Class definitions in Python start with a header line consisting of the keyword `class` followed by the class name.
- Although the body of a class will later contain definitions of attributes and methods, it is possible to define a blank-slate version of the `Employee` class by leaving the body empty:

- Python’s syntactic rules do not allow an empty body. You can either use a docstring as in this example or the keyword `pass`.
- Once you have defined the `Employee` class, you can create an empty `Employee` object like this:

Object Values are References
- It is important to keep in mind that objects—like all values in Python—are stored as `references`. The blank-slate template created by the preceding slide therefore looks like this:

- Any code that has access to this reference can manipulate the contents of the object. In particular, the reference allows code to get and set the contents of existing attributes or to create new ones.

Selecting Attributes of an Object
- Given an object, you can select an individual attribute by writing an expression denoting the object and then following it by a dot and the name of the attribute. For example, in its initialized form, the expression `clerk.name` will have the value “Bob Cratchit”, and the expression `clerk.salary` will have the value 15.
- Attributes are assignable. For example, the statement `clerk.salary *= 2;` doubles poor Mr. Cratchit’s salary.
- In Python, you can create a new attribute simply by assigning it a value, in much the same way that assigning a value to a variable creates a new local variable in the current frame.

Creating an Employee by Assignment
- Although the strategy from the preceding slide creates an `Employee` object with the correct contents, forcing the client to create the attributes violates the spirit of object-oriented programming. The details of the data structure are the property of the implementation rather than the client.
- A better strategy for creating a new `Employee` object is to define a method called a `constructor`, which is responsible for initializing the attributes to the object.
- In Python, you define a constructor by implementing a special method called `__init__`, which is automatically called when a client uses the class name as a function.
- The first parameter to the `__init__` method is called `self`, which is a reference to the new object. Any other arguments provided by the client are passed as additional parameters.
A Constructor for the Employee Class

```python
def ConstructEmployee():
    def __init__(self, name, title, salary):
        self.name = name
        self.title = title
        self.salary = salary

name = 'Bob Cratchit'
title = 'clerk'
salary = 15
```

Defining Additional Methods

- In addition to the constructor, most classes define additional methods that allow clients to read or update attributes of the object or to manipulate the object in some way.

- Methods always declare an explicit parameter `self` at the beginning of the parameter list, just as the constructor does. A method definition therefore looks like this:

```python
def name(self, other parameters):
    ... body of the method ...
```

- Whenever the client calls a method on an object, Python initializes `self` to be a reference to the receiver, which is the object to which the method is applied.

Getters and Setters

- The simplest methods to describe are those that retrieve the value of an attribute, which are called getters, and those that set an attribute to a new value, which are called setters.

- The following definitions show the getter and setter for the salary attribute of an Employee object:

```python
def getSalary(self):
    return self.salary

def setSalary(self, salary):
    self.salary = salary
```

- Getters are much more common than setters. You need to think carefully before providing a setter as to whether you want clients to be able to change the attribute.

Lists of Objects

- Because lists can contain values of any type, the elements of a list can be objects. For example, a list of the employees at Scrooge and Marley can be initialized like this:

```python
SCROOGE_AND_MARLEY = [
    Employee("Ebenezer Scrooge", "founder", 1000),
    Employee("Bob Cratchit", "clerk", 15)
]
```

- The following function prints the payroll for the roster of employees supplied as an argument:

```python
def printPayroll(roster):
    for emp in roster:
        print(emp.getName() + " (" + emp.getTitle() + ") : " + str(emp.getSalary()))
```

Converting Objects to Strings

- If for no other reason than that doing so simplifies debugging, it is good practice to include a `__str__` method in each class to convert an object to a string.

- The `__str__` method for the Employee class looks like this:

```python
def __str__(self):
    return self.name + " (" + self.title + ") : " + str(self.salary)
```

- This definition allows you to simplify the `printPayroll` function to this much shorter form:

```python
def printPayroll(roster):
    for emp in roster:
        print(emp)
```

The GPoint Class

- The Portable Graphics Library defines a class called GPoint, which encapsulates an x and a y coordinate, along with getter methods to return these attributes and a `__str__` method to convert a GPoint to a string.

- To discourage clients from referring to the coordinates directly the GPoint class uses the names `_x` and `_y` to refer to these attributes. By convention, names beginning with a single underscore are considered to be private to the implementation.

- The code for the GPoint class appears on the next slide.
Points and Graphics

- Points turn up often in graphical applications, particularly when you need to store the points in an array or an object.
- As an aesthetically pleasing illustration of the use of points and the possibility of creating dynamic pictures using nothing but straight lines, the text presents the program **YarnPattern.py**, which simulates the following process:
  - Place a set of pegs at regular intervals around a rectangular border.
  - Tie a piece of colored yarn around the peg in the upper left corner.
  - Loop that yarn around the peg a certain distance \( \text{DELTA} \) ahead.
  - Continue moving forward \( \text{DELTA} \) pegs until you close the loop.

**Code for the GPoint Class**

```python
class GPoint:
    """This class represents a location on the graphics plane."""
    def __init__(self, x, y):
        """Initializes a point with the specified coordinates."
        self.x = x
        self.y = y
    def getX(self):
        """Returns the x component of the point."""
        return self.x
    def getY(self):
        """Returns the y component of the point."""
        return self.y
    def __str__(self):
        """Returns the string representation of a point."""
        return "(\%i, \%i)" % (self.x, self.y)
```

**A Larger Sample Run**

**Code for the YarnPattern Program**

```python
# File: YarnPattern.py
...

# This program uses the GLine class to simulate winding a piece of yarn around a list of pegs along the edges of the graphics window.
...

from pgl import GWindow, GLine, GPoint

# Constants
PEG_SEP = 12  # The separation between pegs in pixels
N_ACROSS = 50  # Number of PEG_SEP units horizontally
N_DOWN = 50  # Number of PEG_SEP units vertically
DELTA = 113  # Number of pegs to skip on each cycle

# Derived constants
GWINDOW_WIDTH = N_ACROSS * PEG_SEP
GWINDOW_HEIGHT = N_DOWN * PEG_SEP

# Implementation notes
# The YarnPattern program first calls createPegs to create a list of coordinates of the pegs. It then starts at peg 0 and advances by DELTA pegs on each loop cycle, using modular arithmetic to wrap around to the beginning. On each cycle, it draws a GLine connecting the last point and the current one.

def createPegs():
    """Creates the list of pegs around the edges."""
    pegs = createPegs()
    thisPeg = 0
    nextPeg = -1
    while thisPeg < 0 or nextPeg == -1:
        nextPeg = (thisPeg + DELTA) % len(pegis)
        p_next = pegs[nextPeg]
        line = GLine(p_next.getX(), p_next.getY(), p_next.getX(), p_next.getY())
        line.setColor("Magenta")
        pen.addLine(line)
        thisPeg = nextPeg
    return pegs

# Startup code
if __name__ == "__main__":
    YarnPattern()
```