Lecture 12: Python programming

- **a (formerly) simple language that scales well to large(ish) programs**
- **originally designed & implemented in 1990 by Guido van Rossum at CWI in Amsterdam**
- **very widely used**
  - standard language for many intro courses (though not CS here)
  - standard language for data science
  - arguably the best choice for a first language
- **use version 3, not version 2**

Programming language components

- **syntax: grammar rules for defining legal statements**
  - what's grammatically legal? how are things built up from smaller things?
- **semantics: what things mean**
  - what do they compute?
- **statements: instructions that say what to do**
  - compute values, make decisions, repeat sequences of operations
- **variables: places to hold data in memory while program is running**
  - numbers, text, ...

- **most languages are higher-level and more expressive than the assembly language for the toy machine**
  - statements are much richer, more varied, more expressive
  - variables are much richer, more varied
  - grammar rules are more complicated
  - semantics are more complicated
- **but it's basically the same idea**

Python components

- **Python language**
  - statements that tell the computer what to do
    get user input, display output, set values, do arithmetic,
    test conditions, repeat groups of statements, …
- **built-in functions, libraries**
  - pre-fabricated pieces that you don't have to create yourself
    `print`, **`input`**, math functions, text manipulation, ...
- **access to the environment**
  - file system, network, ...

- **you are not expected to remember syntax or other details**
  you can look them up as needed
- **you are not expected to write code in exams**
  though you will have to write some code in problem sets and labs

- **you <u>are</u> expected to understand the ideas**
  - how programming and programs work
  - figure out what a tiny program does or why it's broken

Example 0: Hello world (hello.py)

- **this is the basic example for most programming languages**

```
print("Hello, world")
```

- **how you can run it:**
  - commandline interactive
  - commandline from a file
  - browser with local files
  - on the web with Colab or other cloud service

Example 1: echo a name (name.py)

- read some input, print it back

```
name = input("What's your name? ")
print("hello,", name)
```

Example 2: join 2 names (name2.py)

- variables, joining strings of characters together

```
firstname = input("Enter first name: ")
lastname = input("Enter last name: ")
result = firstname + lastname
print("hello,", result)
```

Example 3: add 2 numbers (add2.py)

- user input, variables, arithmetic, type conversion

```
num1 = input('Enter first number: ')
num2 = input('Enter second number: ')
sum = int(num1) + int(num2)
print('Sum =', sum)
```

`int(...)` converts a sequence of characters into its integer value

`float(...)` converts a sequence of characters into its floating point value

Example 4: add up lots of numbers (addup.py)

- **variables, operators, expressions, assignment statements**
- **while loop, relational operator (`!=` means "not equal to")**

```
sum = 0
num = input("Enter new value, or empty to end: ")
while num != "":
  sum = sum + float(num)
  num = input("Enter new value, or empty to end: ")
print(sum)
```

Example 5: find the largest number (max.py)

- needs an `if` to test whether new number is bigger
- needs another relational operator
- needs `int()` or `float()` to treat input as a number

```
max = 0
num = input("Enter new value, or empty to end: ")
while num != "":
  num = float(num)
  if num > max:
    max = num
  num = input("Enter new value, or empty to end: ")
print(max)
```

Variables, constants, expressions, operators

- **a *variable* is a place in memory that holds a value**
  - has a **name** that the programmer gave it, like **sum** or **Area** or **n**
  - in Python, can hold any of multiple types, most often

    numbers like **1** or **3.14**, or

    sequences (strings) of characters like **"Hello"** or **"Enter new value"**
  - always has a **value**
  - has to be set to some value initially before it can be used
  - its value will generally change as the program runs
  - ultimately corresponds to a location in memory
  - but it's easier to think of it just as a name for information
- **a *constant* is an unchanging literal value like 3.14 or "hello"**
- **an *expression* uses operators, variables and constants to compute a value**

  3.14 * rad * rad
- ***operators* include `+ - * / %`**

Example 6: compute area of a circle (area.py)

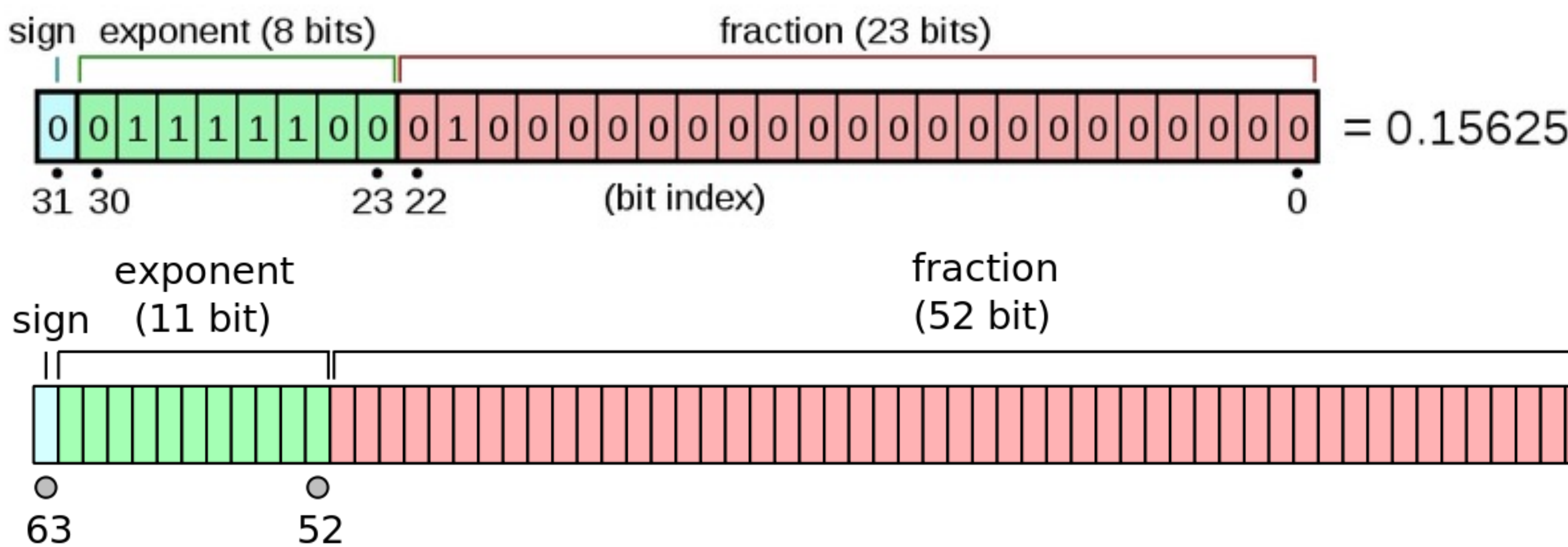
```
import math

r = input("Enter radius: ")
while r != "":
  area = math.pi * float(r) ** 2
  print("radius =", r, ", area =", area)
  r = input("Enter radius: ")
```

- **how do we terminate the loop?**
  - 0 is a valid data value
  - `input()` returns `""` for empty input so use that
- **exponentiation operator is `**`**
- **note use of the math library**

Types, declarations, conversions

- **each variable holds information of a specific <u>type</u>**
  - really means that bits are to be interpreted as info of that type
  - internally, 3 and 3.00 and "3.00" are represented differently



- **Python sometimes infers types from context and does conversions automatically**
- **usually you have to be explicit:**

```
int(...)
float(...)
str(...)
```

Making decisions and repeating statements

- **`if-else` statement makes decisions**
  - the Python version of decisions written with ifzero, ifpos, ...

`if` ***condition is true*****`:`**

do this group of statements

`else:`

do this group of statements instead

- **`while` statement repeats groups of statements**
  - a Python version of loops written with `ifzero`, `ifpos` and `goto`

`while` ***condition is true*****`:`**

do this group of statements

- **INDENTATION MATTERS**
  - indicates what statements are within the `if` or `while`

Example 7: `if-elif-else` sequence (sign.py)

- can include `else-if` ("`elif`") sections for a series of decisions:

```
num = input("Enter number: ")
while num != "":
  num = int(num)
  if num > 0:
    print(str(num) + " is positive")
  elif num < 0:
    print(str(num) + " is negative")
  else:
    print(str(num) + " is zero")
  num = input("Enter number: ")
```

Example 8: `while` loops

- counting or "indexed" loop:

```
i = 1
while i <= 10:
    # do something (maybe using current value of i)
    i = i + 1
```

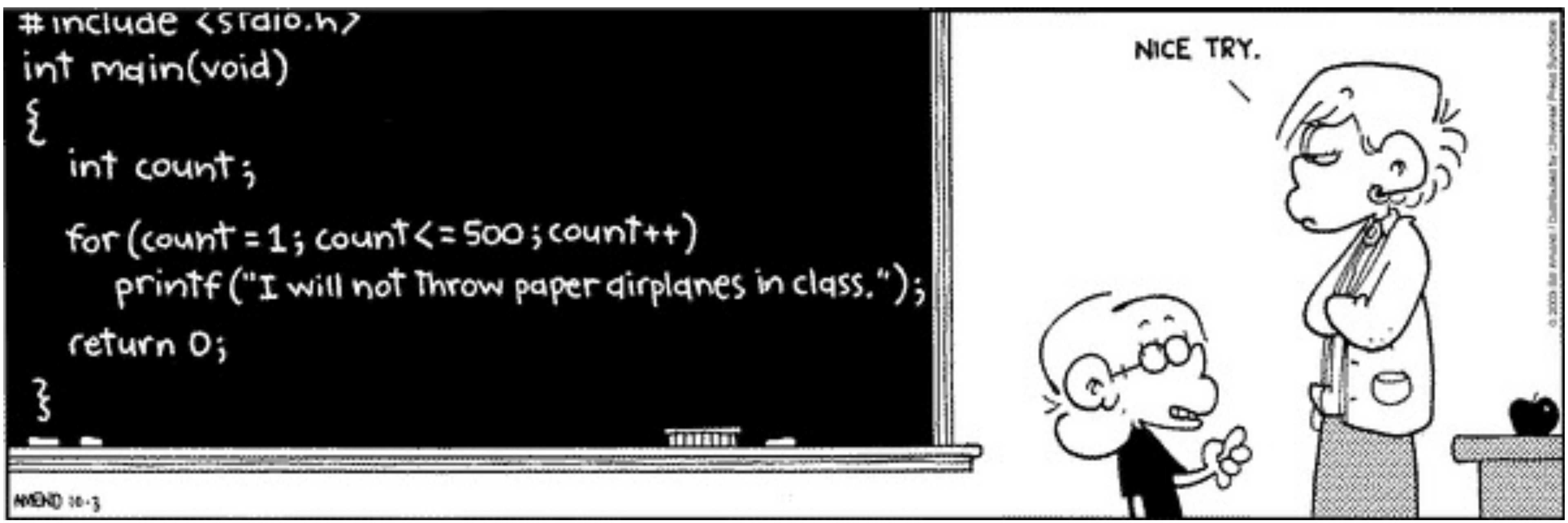
- "nested" loops (while.py):

```
n = input("Enter number: ")
while n != "":
  i = 0
  while i <= int(n):
    print(i, i * i)
    i = i + 1
  n = input("Enter number: ")
```

Example 9: `for` loop

```
for i in range(0, 500):
   print("I will not throw paper airplanes in class.")
```

C version:



Functions

- **a function is a group of statements that does some computation**
  - the statements are collected into one place and given a name
  - other parts of the program can "call" the function
    that is, use it as a part of whatever they are doing
  - you can supply the function with values to use in its computation
    ("arguments" or "parameters")
  - the function computes or "returns" a value that can be used in expressions
    (the value need not be used)

- **Python provides some useful built-in functions**
  - e.g., **`print`**, **`input`**, ...

- **you can write your own functions**

Example 10: functions

- **syntax**

  `def` *name* **(***list of arguments***):**

  the statements of the function

- **example of a function definition:**

```
def area(r):
    return math.pi * r ** 2
```

- **using ("calling") the function:**

```
r = input("Enter radius ");
print("radius =", r, ", area =", area(r))
```

- **calling it twice in one expression:**

```
print("CD recording surface =", area(2.3) - area(0.8))
```

Example 11: area of a ring (ring.py)

```
import math

def area(r):
  return math.pi * r ** 2

r1 = input("Enter radius 1: ")
while r1 != "":
  r2 = input("Enter radius 2: ")
  print("Area = ", area(float(r1)) - area(float(r2)))
  r1 = input("Enter radius 1: ")
```

Why use functions?

- **if a computation appears several times in one program**
  - a function collects it into one place

- **breaks a big job into smaller, manageable pieces**
  - that are separate from each other
  - multiple people can work on the program

- **defines an interface**
  - implementation can be changed as long as it still does the same job

- **a way to use code written by others long ago and far away**
  - most of Python's library of useful stuff is accessed through functions

- **good libraries encourage use of the language**
  - a major reason for the success of Python

Data structures

- **how to organize related data items in a program**
  - so they can be treated uniformly, e.g., in a loop

- **usually means groups of related data items, e.g.,**
  - info about a particular student
  - list of student names
  - list of info about all students

- **basic Python data structures**
  - object: a collection of one or more related variables
    e.g., info about a particular student
  - array: a sequence of items numbered from 0 to whatever
    (confusingly, Python calls this a list)
    indexed by number, like `a[n]`; elements are `a[0]` ... `a[n-1]`
  - dictionary: a set of items indexed by name
    `weight["A+"]`

Summary: elements of (most) programming languages

- **constants: literal values like 1, 3.14, "Hello, world!"**
- **variables: places to store data and results during computing**
- **declarations: specify name (and type) of variables, etc.**
- **expressions: operations on variables and constants to produce new values**
- **statements: assignment, conditional, loop, function call**
  - assignment: store a new value in a variable
  - conditional: compare and branch; if-else
  - loop: repeat statements while a condition is true
- **data structures: ways to organize related data items**
- **functions: package a group of statements so they can be used ("called")**

  from other places in a program
- **libraries: functions already written for you**

How Python works

- **recall the process for Fortran, C, etc.:**

  compiler => assembler => machine instructions

- **Python is analogous, but differs significantly in details**

- **Python compiler**
  - checks for errors
  - compiles the program into instructions for something like the toy machine, but richer, more complicated, higher level
  - runs a simulator program (like the toy) that interprets these instructions

- **the simulator is often called an "interpreter" or a "virtual machine"**
  - probably written in C or C++ but could be written in anything

Real-world programming

- **the same thing, but on a grand scale**
  - programs may be millions of lines of code

    typical productivity: 1-10K lines/year/programmer
  - thousands of people working on them
  - lifetimes measured in years or even decades
- **big programs need teams, management, coordination, meetings, …**
  - schedules and deadlines
  - constraints on how fast the program must run, how much memory it can use
  - external requirements for reliability, safety, security, legal compliance, interoperability with other systems, …

- **maintenance of old ("legacy") programs is hard**
  - programs must evolve to meet changing environments and requirements
  - machines and tools and languages become obsolete
  - expertise disappears