Lecture 13
Python

```c
#include <stdio.h>
int main(void)
{
    int count;
    for (count = 1; count <= 500; count++)
        printf("I will not throw paper airplanes in class.");
    return 0;
}
```
The Phish Bowl

Latest Phishing Attempts

YOUR COVID-19 PANDEMIC RELIEF FUND IS READY
10/20/2021 PM

Amazon Shipment Update
10/19/2021 PM

HTML
10/18/2021 PM

Action Required
10/16/2021 AM

Tutoring
10/13/2021 PM

"STUDENT EMPLOYMENT" alternate subject "JOB POSTING"
10/12/2021 AM

Weekly Position Announcement PT/FT
10/12/2021 AM

Forward suspicious messages to: phishbowl@princeton.edu

View The Phish Bowl video.

View our handout, "Phishing - Don't Take the Bait!"
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
What is Python?

• a comparatively simple language that scales well to large(ish) programs

• designed & implemented in 1990 by Guido van Rossum at CWI in Amsterdam

• very widely used
  – standard language for many intro courses (though not here)
  – standard language for data science
  – arguably the best choice for a first language

• use version 3, not version 2
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, ...

- **libraries, built-in functions**
  - 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 are not expected to write code in exams**
  (though a bit in problem sets and labs)

- **you are expected to understand the ideas**
  - how programming and programs work
  - figure out what a tiny program does or why it's broken
Basic example 0: Hello world (hello.py)

- this is the basic example for most programming languages
  
  ```python
  print("Hello, world")
  ```

- how we run it:
  
  - commandline interactive
  - commandline from a file
  - browser with local files
  - on the web with Colab or other cloud service
Basic example 1: echo a name (name.py)

- read some input, print it back

```python
name = input("What's your name? ")
print("hello,", name)
```
Basic example 2: join 2 names (name2.py)

- shows variables

```python
firstname = input("Enter first name: ")
secondname = input("Enter lastname: ")
result = firstname + secondname
print ("hello," , result)
```
Adding up lots of numbers: addup.py

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

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

```python
max = 0
num = float(input("Enter new value, or 0 to end: "))
while num != 0:
    if num > max:
        max = num
    num = float(input("Enter new value, or 0 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 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` or "hello"
- **an expression** uses operators, variables and constants to compute a value
  - `3.14 * rad * rad`

- **operators** include `+`, `-`, `*`, `/`
Computing area: area.py

```python
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 type
  - 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 we 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 and goto

  while condition is true:
    do this group of statements
if-else examples  (sign.py)

• can include else-if sections for a series of decisions:

```python
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: ")
```
while-loop examples

- counting or "indexed" loop:
  ```python
  i = 1
  while i <= 10:
      # do something (maybe using current value of i)
      i = i + 1
  ```

- "nested" loops (while.py):
  ```python
  n = input("Enter number: ")
  while n != ":
      i = 0
      while i <= int(n):
          print(i, i * i)
          i = i + 1
  n = input("Enter number: ")
  ```
Another kind of loop: "for"

```c
#include <stdio.h>
int main(void)
{
    int count;
    for(count = 1; count <= 500; count++)
        printf("I will not throw paper airplanes in class.");
    return 0;
}
```

```python
for i in range(0, 500):
    print("I will not throw paper airplanes in class.")
```
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
  – can give it values to use in its computation (arguments or parameters)
  – the function computes 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
Function examples

- syntax
  ```python
def name(list of "arguments"):
    the statements of the function
  ```

- example of a function definition:
  ```python
def area(r):
    return math.pi * r ** 2;
  ```

- using ("calling") the function:
  ```python
r = input("Enter radius ");
print("radius =", r, ", area =", area(r))
  ```

- calling it twice in one expression:
  ```python
print("CD recording surface =", area(2.3) - area(0.8))
  ```
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
• defines an interface
  – implementation details can be changed as long as it still does the same job
  – different implementations can interoperate
• multiple people can work on the program
• 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
• a good library encourages use of the language
Summary: elements of (most) programming languages

• constants: literal values like 1, 3.14, "Error!"
• 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
• functions: package a group of statements so they can be called / used 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

• simulator is often called an "interpreter" or a "virtual machine"
  – probably written in C or C++ but could be written in anything
The process of programming

• what we saw with Python or Toy is like reality, but very small

• figure out what to do
  – start with a broad specification
  – break into smaller pieces that will work together
  – spell out precise computational steps in a programming language

• build on a foundation (rarely start from scratch)
  – a programming language that's suitable for expressing the steps
  – components that others have written for you
    functions from libraries, major components, ...
  – which in turn rest on others, often for several layers
  – runs on software (the operating system) that manages the machine

• it never works the first time
  – test to be sure it works, debug if it doesn't
  – evolve as get a better idea of what to do, or as requirements change
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 criteria for reliability, safety, security, 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