# Module 01: Introduction to Programming in Python

Topics:

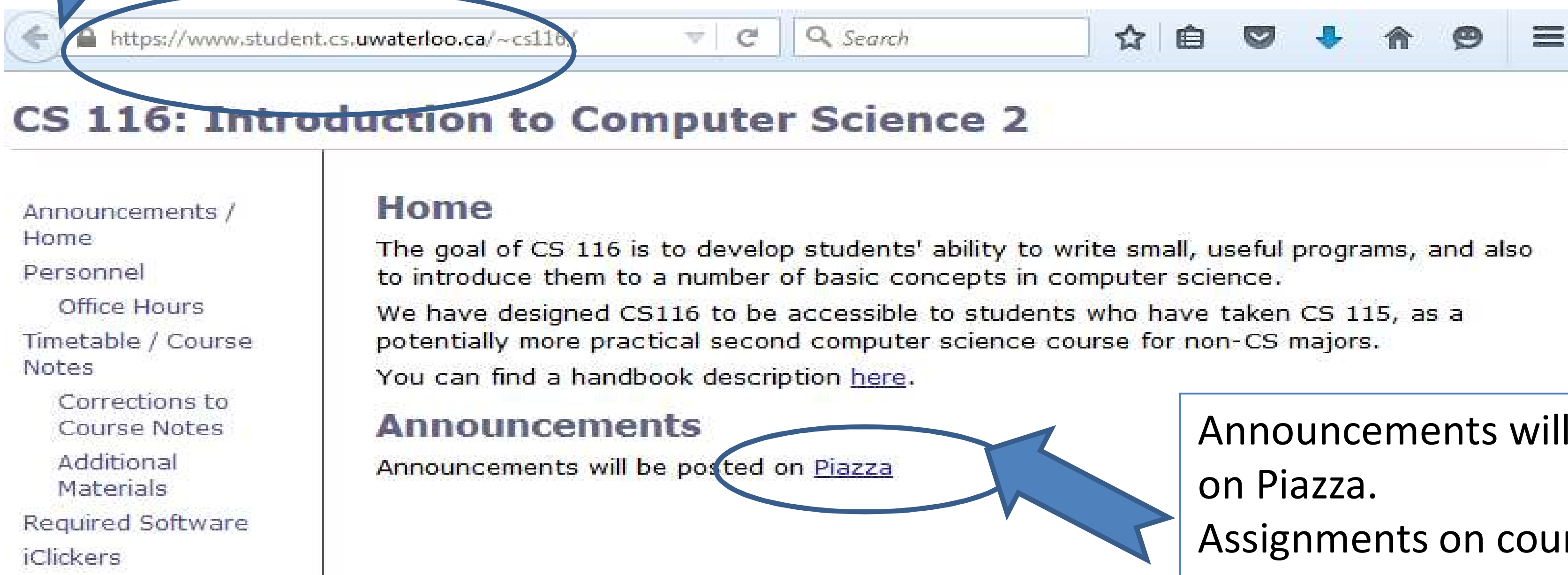
- Course Introduction
- Introduction to Python basics

Readings: ThinkP 1,2,3

Practice: Self study exercises

# Finding course information

- https://www.student.cs.uwaterloo.ca/~cs116/




Announcements will be on Piazza.
Assignments on course web page and submitted through MarkUs.

# Important Administrative Details

- Announcements
- Weekly Tutorials
- (almost) Weekly Assignments
  - No extensions
  - Remark Policy: 2 weeks
  - **Submit code early and often**
  - Check your basic tests emails
  - Will drop lowest assignment grade (some restrictions)
- Academic Integrity Policy
- AccessAbility Services

# Grading

- Assignments 20%
- Participation 5%

  (clicker questions with tutorial bonus)
- Midterm 30%
- Final 45%

**Note: You must pass the weighted average of the midterm and final in order to pass the course.**

# Major Themes from CS115

- Design
- Common Patterns
- Verification
- Communication

CS115 was not a course *just* about Racket!

# Major Themes for CS116

- Design
- Common Patterns
- Verification
- Communication
- ***Algorithms***

CS116 is not *just* a course about Python!

# Introducing Python …

- We will learn to do the things we did in Racket
- We will learn to do new things we didn’t do in Racket
- Why change?
  – A different programming paradigm
    - Racket is a functional programming language
    - Python is an imperative programming language
  – Design recipe still applies

# What can Python programs do?

- Everything we did with Racket programs
- Lots of things we didn't cover in Racket

# Functional vs Imperative languages in problem solving

- Much is the same: determine needed data types, variables, and helper functions.
- With a functional language like Racket:
  - Calculations are nested to show precedence
  - Calculated value is returned
- With an imperative language like Python:
  - Steps are separated, and ordered (similar to **`local`** in Racket)
  - Data values change as the program executes
  - Calculated values may (or may not) be returned by a function

# Running a Python Program

- Uses an interpreter like Racket (unlike most imperative languages)
  - Translates one statement at a time
  - Stops when one error is found
- Most imperative languages use a compiler
  - Translates entire program into machine code
  - Finds all errors in entire program
- Generally, harder to debug with a compiler but code typically runs faster.

# What does a Python program look like?

- A series of statements
  - *Assignment statements*
  - *Control statements*
  - Function calls
- May include function definitions
  - Made up of statements
- May include new type definitions

# Some Python Basics

- Written using regular mathematical notation

```
3 + 4
5 * (3 + 4) - 1
```

- Two numeric types (integers and floating point numbers) instead of one
- Strings, Booleans, lists
- No character or symbol type used in CS116.

# Assignment Statements

# `v = expr`

- Similar to Racket's **`define`**
- **`=`** is the assignment operator ("becomes" or "is")
- **`v`** is any variable name
- **`expr`** is any Python expression
- How it works:
  1. Evaluate **`expr`**
  2. "Assign" that value to **`v`**
- Assignment statements do not return a value. They only have an effect.

# A very simple Python program

```
x = 2 * (4 + 12)
y = x + 8
z = y * y
q = 2.5 + 4.2
w = "hi"
u = w + w
```

What are the values of **`x, y, z, q, w, u`**?

# Racket vs Python: Numeric types

- Numeric calculations in Racket were exact, unless involving irrational numbers
  - no real difference between **`3`** and **`3.0`**
- Integers in Python are stored exactly
- Other numbers are approximated by floating point values → Representation error
- Two different numeric types:
  - **`3`** is of type **`Int`**, but **`3.0`** is of type **`Float`**

# Racket vs Python: Numeric types

| | Racket | | Python | |
|---|---|---|---|---|
| *Value* | *Representation* | *Type* | *Representation* | *Type* |
| *natural* | exact | **Nat** | exact | **Nat** |
| *integer* | exact | **Int** | exact | **Int** |
| *rational* | exact | **Num** | inexact | **Float** |
| *irrational* | inexact | **Num** | inexact | **Float** |

Use these type names in Python contracts

Recall, in Racket:

- **`check-expect`** for testing exact values
- **`check-within`** for testing inexact values

# Basic Mathematical Operations

- Addition (+), Subtraction (-), Multiplication (*):
  - If combining two **`Int`** values, the result is an **`Int`**
  - If combining two **`Float`** values, or a **`Float`** and an **`Int`**, the result is a **`Float`**

# Basic Mathematical Operations

- Division: **`x / y`**
  - The result is a **`Float`** for any numerical values **`x`** and **`y`** (even if both are **`Int`**)
- Integer division: **`x // y`**
  - The result is the integer part of the division
  - If **`x`** and **`y`** are both **`Int`**, the result is an **`Int`**
  - If either **`x`** or **`y`** is a **`Float`**, the result is a **`Float`**
  - Returns quotient when **`x`** and **`y`** are **`Nat`**

# Other Mathematical Operations

- Remainder: **`x % y`**
  - **`x`** and **`y`** should both be **`Nat`**
  - returns the **`Nat`** remainder when **`x`** divided by **`y`**
- Exponents: **`x ** y`**
  - **`(anyof Int Float) (anyof Int Float) -> (anyof Int Float)`**
  - returns **`x`** raised to the power of **`y`**

# More useful things to know

- Python precedence operations are standard math precedence rules (BEDMAS)
- Use ## or # for comments (from beginning or middle of line)
- Do not use dash in variable names
  - Use underscore instead

# Calling functions in Python

**`fn_name (arg1, arg2, …, argN)`**

- built-in function or a user-defined **`fn_name`**
- must have correct number of arguments
- separate arguments by single comma
- examples:

```
abs(-3.8) ⇨ 3.8
len("Hello There") ⇨ 11
max(3,5.2,9) ⇨ 9
min("ABC", "AA") ⇨ "AA"
```

# The `math` Module

- A Python module is a way to group together information, including a set of functions
- The `math` module includes constants and functions for basic mathematical calculations
- To use functions from `math`
  - Import the `math` module into your program
  - Use `math.fn` or `math.const` to reference the function or constant you want

# Type in the interactions window

```
import math
math.sqrt(25)
math.log(32,2)
math.log(32.0, 10)
math.floor(math.log(32.0, math.e))
math.factorial(10)
math.cos(math.pi)
sqrt(100.3)
```

Error!! Must use `math.sqrt(100.3)`

# More **math** functions

```
>>> import math
>>> dir(math)
[…, 'acos', 'asinh', 'atan', 'ceil', 'cos',
  'cosh', 'degrees', 'e', 'exp',
  'factorial', 'floor', 'log', 'log10',
  'pi', 'pow', 'radians', 'sin', 'sqrt',
  'tan', 'trunc', …]
>>> help(math.floor)
Help on built-in function floor in module
  math:
floor(...)
    floor(x)
    Return the floor of x as an integer.
    This is the largest integral value <= x.
```

# Creating new functions in Python

```
def fname (p1, p2, …, pN):
  statement1
  …
  statement
```

- Indent each statement the same amount
- For a function to return a value, include
  **`return answer`**
  where **`answer`** is the value the function returns
- If there is no **`return`** statement or a **`return`** statement without an expression for **`answer`**, the function will return **`None`** (which is the default with a Python function).

Example: Write a Python function that consumes 3 different integers and returns the middle value.

```
def middle(a,b,c):
    largest = max(a,b,c)
    smallest = min(a,b,c)
    mid = (a+b+c)-largest-smallest
    return mid
```

# Review: Design Recipe for Functions

When writing functions in Racket, we included:

- Purpose statement
- Contract
- Examples
- Function body
- Test cases

We'll continue with these steps for Python programs, but there will be some changes.

# Python's docstring

- Python provides a convenient way to associate documentation with a function.
- A string included after the function header becomes the help documentation for that function.
- As this will require more than one line, we will use special string delimiters: three single quotes before and after our design recipe steps.

# Basic docstring usage

*Definitions window*

```
def middle(a,b,c):
    ''' returns middle
        value of
        a,b,c
    '''
    lr = max(a,b,c)
    sm = min(a,b,c)
    mid = (a+b+c)-lr-sm
    return mid
```

*Interactions window*

```
>>> help(middle)
Help on function
middle in module
__main__:

middle(a, b, c)
    returns middle
    value of
    a,b,c
```

# Design Recipe: Some things remain basically the same

Some steps are the same in Python as in Racket:

- Purpose statement:
  - Explicitly indicate what the function does, including how the parameters are used
  - New style: Use "returns" rather than "produces"
- Contract
  - Types of consumed and returned values
  - Include any needed requirements on parameters
  - Most type names are the same as in Racket, except for Num; Use Nat, Int, Float as appropriate

# Design Recipe: Some things have to change

Examples …

- We cannot write our examples as tests as we did in Racket, so a different approach is needed here.
- Our new approach (inside our docstring)

**`fn(arg1, arg2, …)`** ⇨ **`expected`**

- For example:

**`middle(4,2,8)`** ⇨ **`4`**

**`middle(3,2,1)`** ⇨ **`2`**

```
def middle(a,b,c):
    ''' returns the middle (median) value of a,b,c

        middle: Int Int Int -> Int
        requires: a,b,c are all different

        Examples:
        middle(4,2,8) => 4
        middle(3,2,1) => 2
    '''

    largest = max(a,b,c)
    smallest = min(a,b,c)
    mid = (a+b+c) - largest - smallest
    return mid
```

We've dropped the function call at the beginning of the purpose statement, since it is included in the header.

Blank lines between parts of the design recipe improve readability.

# More on design recipe

- We will soon see that testing is similar, but different
- While templates will not be a focus in CS116, you may still find them helpful, and we will try to point out common code patterns when it might be helpful.

# Why we use the Design Recipe

Program design still involves creativity, but the design recipe can be very helpful:

- It provides a place to start.
- Contracts and purpose can reduce simple syntax errors.
- Good design and template choices can
  - reduce logical errors
  - provide better solutions

# What goes in the body of a Python function?

- Assignment statements
  - May introduce new, local variables
- Calls to other functions
  - Built-in functions
  - User-defined functions
- `return` statement
  - Will be last code executed when present

We will learn more Python statements as we progress.

# Using local variables in Python

In **`middle`**,

- **`largest`**, **`smallest`**, **`mid`**

are local variables.

They do not exist outside of middle.

# More on local variables and functions

- A variable initialized inside a function only exists in that function
- If  your function calls a helper function, the helper function cannot access the caller's variables
- Helper functions can be defined locally, but we will learn about that later
- Need only provide contract and purpose for helper functions

# Example: Write a Python function to compute the area of a circle with nonnegative radius **r**

```
import math
def area_circle (radius):
    ''' returns the area of a circle with
        the given radius

        area_circle: Float -> Float
        requires: radius >=0

        Examples:
        area_circle(0.0) => 0.0
        area_circle(1.0) => 3.14159265
        '''
    return math.pi * radius * radius
```

# Picky, picky, picky … Indentation in Python

A small change in indentation will lead to error

```
def tens_digit(n):
    '''returns the tens digit in n
       tens_digit: Nat -> Nat
       Examples:
       tens_digit(1234) => 3
       tens_digit(4) => 0
       '''
    div10 = n // 10
       tens = div10 % 10
  return tens
```

***WARNING!! This example contains indentation errors!***

# Design Recipe: Testing in Python

- Our Python functions must still be tested
- Choosing test cases will be similar to before
  - Black box tests
    - Based on problem description
  - White box test
    - Based on actual implementation
- The mechanics of testing in Python will be different as Python does not have built-in **`check-expect`** or **`check-within`**

# CS116 "**`check`**" Module

- Download the file: **`check.py`** from the CS116 web pages. Put a copy in the same folder as your **`.py`** files for each assignment.
- Add the following line to each assignment file:

```
import check
```

- You do NOT need to submit **`check.py`** when you submit your assignment files.
- A message is displayed for each test.

# check.expect

```
## Question 1, Test 1: description
check.expect(
    "Q1T1",
    expr,
    value_expected)
```

Label the test – can put description here

Actual result - usually a function call

Expected result;
Calculate it yourself

- This function performs the test:

Does **expr** exactly equal **value_expected**?

- Use for checking exact values (integer or strings).

# Testing `middle`

```
check.expect(
 "middle is first value",
  middle(3,10,1),3)
check.expect(
 "middle is second value",
  middle(2,5,9), 5)
```

***Note: Your examples should be coded as tests as well.***

# Testing functions that return `Float` values

- Recall that Python has two numeric types: **`Int`** and **`Float`**.
- **`Float`** values may not be stored exactly, and additional errors may be introduced in floating point calculations.
  - Answers may not be exact
- Do not use **`check.expect`** for testing functions that return a **`Float`**. Always use **`check.within`** instead.

# check.within

```
## Question 2, Test 2: description
check.within(
     "Q2T2",
     expr,
     value_expected,
     tolerance)
```

Label the test

Actual result from function

Expected result

Maximum allowed error

- This function performs the test:
  **`abs(expr – value_expected) <= tolerance`**
- Use for checking inexact values (floating point numbers only).

# Testing `area_circle`

`area_circle` returns a floating point

→Don't test for exact equality

```
check.within("zero radius",
  "area_circle(0.0), 0.0, 0.00001)

check.within("positive radius",
  area_circle(1.0), 3.14159, 0.00001)
```

Note: 0.00001 is typically a good **`threshold`** for our tests.

# Investigating **return** further

```
def total_digits(secret):
    ones = secret % 10
    tens = secret // 10
    sum = ones + tens
    return
```

*Assume*
*10≤secret≤99*

```
>>> d = total_digits(74)
```

What is the value of **d**?

How would you write the contract of **total_digits**?

# And even further

```
def total_digits(secret):
    ones = secret % 10
    tens = secret // 10
    sum = ones + tens

def calculation(secret):
    s = total_digits(secret)
    return secret - s

c = calculation(74)
```

*Assume*
*10≤secret≤99*

# Warning: Continuing a Python statement over multiple lines

- Python expects each line of code to be an entire statement
  - Can be a problem with line length and readability due to indentation
- If a statement is not done, use a \ (backslash) character to show it continues on next line
  - \ is not needed if you have an open bracket on the unfinished line

# More on Basic Types in Python

- Remember that the differences between integers and floating point numbers can complicate calculations
- Python has many built-in conversion functions from one basic type to another

# How to get the type we want: More Casting and Conversion Functions

- **`float: Int → Float`**
  - `float(1)` ⇨ `1.0`, `float(10)` ⇨ `10.0`
- **`float: Str → Float`**
  - `float("34.1")` ⇨ `34.1`,
  - `float("2.7.2")` ⇨ `Error`
  - `float("23")` ⇨ `23.0`
- **`float: Float → Float`**
  - `float(23.4)` ⇨ `23.4`

# More Casting Functions

- **`int: (anyof Float Str Int)`** → **`Int`**
  - **`int(4.7)`** ⇨ **`4`**, **`int(3.0/4)`** ⇨ **`0`**,
  - **`int(-12.4)`** ⇨ **`-12`**
  - This is a truncation operation (not rounding)
  - **`int("23")`** ⇨ **`23`**
  - **`int("2.3")`** ⇨ **`Error`**

- **`str: (anyof Int Float Str)`** → **`Str`**
  - **`str(3)`** ⇨ **`"3"`**, **`str(42.9)`** ⇨ **`"42.9"`**

# Goals of Module 1

- Become comfortable in Python
  - Basic types and mathematical operations
  - Calling functions
  - Defining functions
  - Using **`return`** correctly
  - Design recipe in Python