Learning Python for COSC428 students

- You should be able to teach yourself Python fairly easily using the tutorial on the Python website: https://docs.python.org/3/tutorial
- The main resource for COSC121 is our departmental quiz server, which contains all the training quizzes. Although we can't give direct access to that course, Richard Lobb has set up a separate "course" on our quiz server that contains all the same lab quizzes as COSC121, plus the lecture notes, so that should fit needs without all the extraneous material that wouldn't be relevant.
- So contact Richard Lobb who is happy to enrol you

(which means logging you in on his quiz server, https://quiz2018.csse.canterbury.ac.nz, so that your name gets added to the quiz server's database of students)

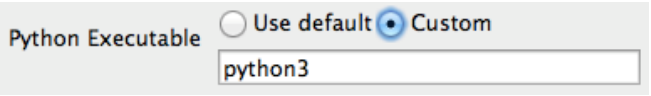
Getting started with COSC428 labs

- How to see the lab material?
  - Log in to UC Learn
    - Use your normal Uni username/password
  - Select COSC428
  - Select the lab and download files
- Do Lab Lab 01 (Working with Cameras) in the first week.
  - Launch *Wing101* and start doing the lab
  - Follow the Lab 01 instructions such as completing missing lines in the code.

Getting Python and Wing set up at home

At home, to get ready for the rest of the labs:

- **Download and install Python 3.5** from www.python.org/downloads/
- **Download and install the most recent version of Wing 101** (6.0.1 as of now) from: *www.wingware.com/downloads/wingide-101*
  - NB: Get the *free* Wing IDE 101, not Wing IDE Personal or Wing IDE Professional
  - On Mac OSX in Wing IDE 101 set Python to version 3 by choosing Edit->Configure Python... then for Python Executable choose Custom and type python3 in the box:



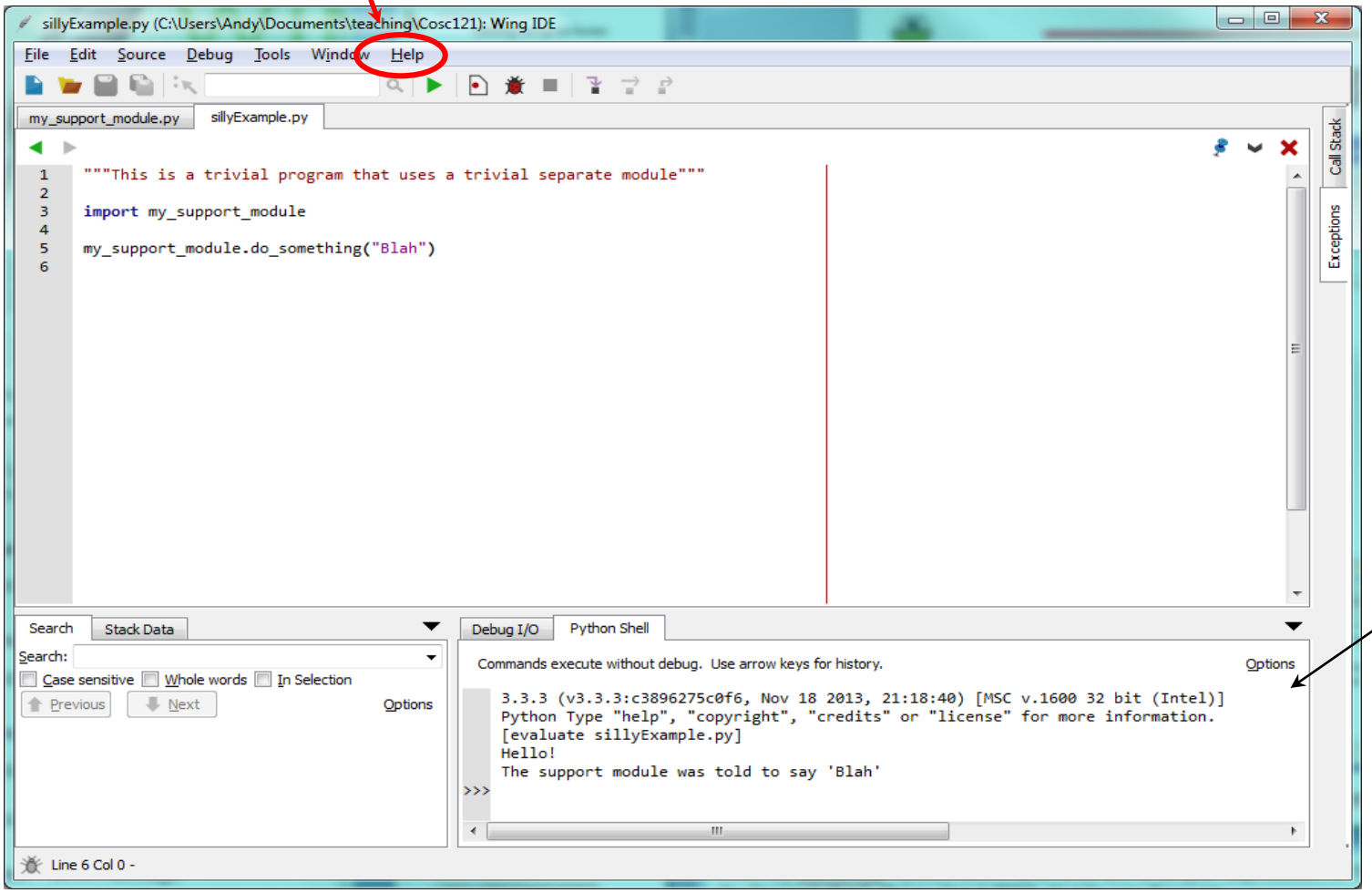
The above items are already installed in our labs; you don't have to install them.

NB!

Wing 101 IDE

DEMO

Program editing area



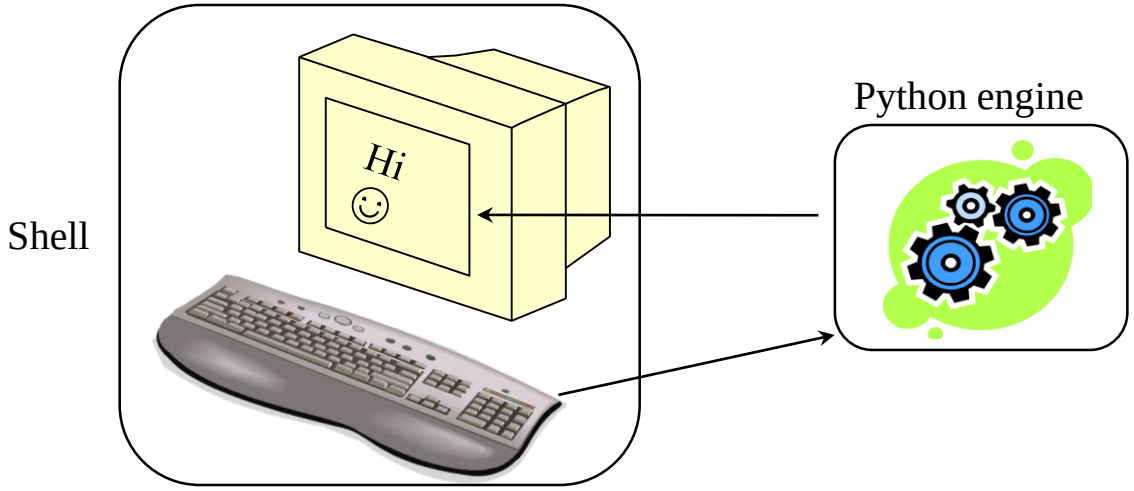
Integrated Development Environment

Python Shell pane

The Python Engine and Shell

DEMO

- The "Python Engine" is the program that executes ("interprets") Python instructions (programs)
  - A "virtual machine" or "scripting engine"
- Python Shell (bottom right pane in Wing) is a "terminal" interface to the Python engine



How Python shell works

- Python shell repeatedly:
  - "reads the input" (what you type on the keyboard)
  - "evaluates" what you entered using the Python engine
  - "prints" the results of evaluation (if anything is "returned")

```
>>> 5
5
>>> 2 + 7
9
```

Running programs in Wing

- Typing functions directly into the Python shell is clumsy
- Instead we enter them into a *program file*
- Then we can:
  - Run the file
  - Edit it easily
  - Come back to it days later
  - Re-use the functions in other programs
- We are now *programming* ☺

DEMO

Overview of programming in Python

Expressions

DEMO

- An *expression*: something that can be *evaluated* to *yield* a value
  - Typically a sequence of *operands* and *operators*
  - E.g. (25 * 3 – 5) / 7
    - Operands here are: 25, 3, 5, 7
    - Operators: * , –, /
    - Evaluates to 10.0
- Arithmetic operators (in lab 1):

  +, -, *, /, //, **, %

  Last three are *integer division*, *exponentiation* and *modulus*

Expressions (cont'd)

- Exponentiation: 2 ** 3 is 8 (i.e. 2^3)
- Division:
  - 26 / 3 is 8.666666666666666
  - 26 // 3 is 8 [Integer division]

NB: Different from other languages and even from Python 2. Beware, if you're using the old textbook: in Python 2, '/' was Python 3's '//'

- Modulo reduction: 26 % 3 is 2
  - the remainder after dividing 26 by 3
- *Operator precedence* determines order of evaluation
  - ** highest then *, /, //, % then +, – [but more operators later]
    - Left-to-right (usually) if operators have same precedence
  - Parentheses used to change default order
    - 2 + 3 * 5 is 17, (2 + 3) * 5 is 25

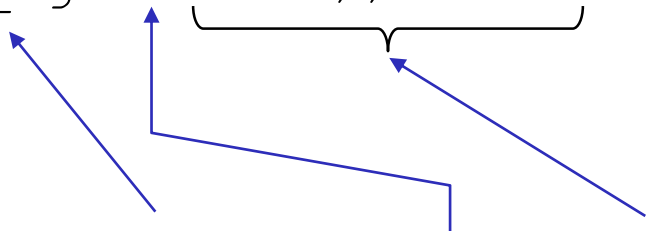
Introduction to Values and Types

- Values (or “objects”)
  - Example: 1, 2.3, -82
  - More examples (we’ll see later): “Hello”, [“Pink”, “Rock”]
- *Types* are classification of values. Each type allows certain operations.
- We have seen two Number types so far:
  - *int*: whole numbers, e.g. 28196
    - Exact
    - Any arbitrary size/accuracy
  - *float*: numbers with fractional bit, e.g. 5.15296
    - Approximate: ~16 digits accuracy
    - Stored in *binary representation* so some numbers like 1.1 are approximate and some like 1.5, 1.25, 1.125 have exact representations.

Assignment statements

DEMO

- Python shell *executes* any *statements* you enter
- One type of statement is the *assignment statement*
  - E.g. `my_age = 200 // 2 - 1`



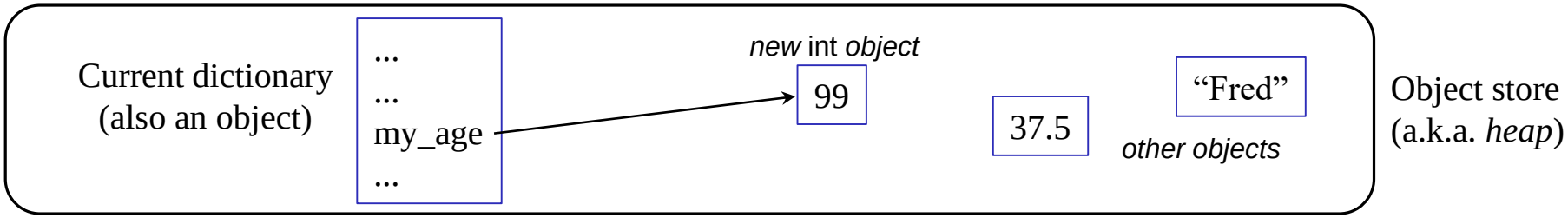
- Of form: *variable_name* = *expression*
- A variable name must be a letter (or underscore) followed by any number of *alphanumeric characters* (letters, underscores or digits)

What Python does with that

1. Works out the value of the Right Hand Side (RHS)
   - The value resides somewhere in the "object store" (or *heap)*
   - In the example, the object is of type *int* (i.e.,"Integer") with the value 99
2. Assigns a name to the object:

   a) Adds the *variable name* to the current "dictionary" of variables (unless it's already there)

   b) Sets the dictionary entry to point to the new object (a *reference* to the object)



Demo: http://www.pythontutor.com/. Use live programming mode with "render all objects on the heap". [It calls the current dictionary a "frame".]

What a reference actually is

- The computer has lots of *random access memory* (RAM)
  - e.g. 4 gigabytes (GB) where a *byte* is 8-bits, e.g. 01101101
- Bytes are numbered 0, 1, 2, 3, 4, ... 4 GB
  - The number of each byte is called its *address*
- A *reference* to an object is the address in memory at which the object is stored (i.e., where it starts)
  - In Python it's called the object's *identity*
- Thus a dictionary entry consists of a variable name (called a "string") together with the identity of the object it references
  - Shown as an arrow in the figures

Using variables

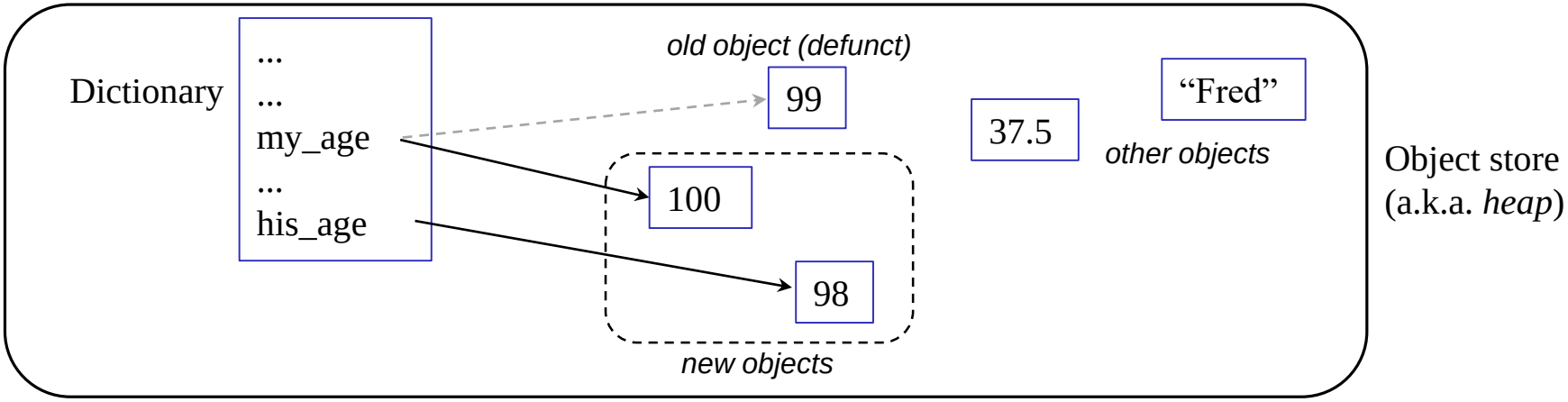
- When a variable name appears in an expression, the associated object's value is used in its place, e.g.

```
my_age = 99

his_age = my_age - 1

my_age = my_age + 1
```

- '=' is **not** "equals"!!! It means "assign to" or "bind name to"



- Python is case-sensitive. `My_age` is different from `my_age`.

Combined Operators

- We often use operations like

```
count = count + 1   # Increment the count
size = 2 * size     # Double the size
```

- So Python provides short-cut "combined operators", e.g.

```
count += 1
size *= 2
```

Functions

- The key to programming is *abstraction*
  - *Abstraction is a process by which concepts are derived from the usage and classification of literal ("real" or "concrete") concepts.* **- *Wikipedia***
  - *Naming* a concept is a key part of abstraction
- Example: "Hey, I often need to multiply a number by itself. I know, let's call that *squaring* a number"
- In Python, *functions* are used for abstracting common procedures (i.e., sequences of operations) and as building blocks ("divide and conquer")
  - We'll see other abstraction methods – modules and classes – later.

Using functions

- Example:
  - Function call `round(x)` returns the nearest *int* to the *float* value *x*
  - `round(45.6)`
    - Here round is the name of the function
    - 45.6 is the argument
- We can use functions in expressions
  - `round(4.4) + 3  # Evaluates to 7`

Note term *returns*. In full, we say "When **called** with an argument of 45.6 the *round* function **returns** the value 46"

Built-in functions

- Some built-in functions:
  - round(x) returns the nearest *int* to the *float* value *x*
  - abs(x) returns the absolute value of *x*
  - int(x) converts *x* into an *int*
    - If *x* is a *float*, it truncates.
    - Later we'll see *x* can also be a string.
- You'll meet lots more in due course
  - A lot of functions in Python libraries (modules) – see later
  - A lot of functions as *methods* (not simple functions) – see later.

Defining new functions

```
def square(x):              # x is called a "parameter"
    return x * x    # the "body" is indented
```

- Used by *calling* (or *invoking*) it, e.g.

```
square(3)                   # 3 is called the "argument"
square(37.5)                # Here 37.5 is the argument
square(2 + 3 * 5)  # The argument is an expression
```

- The parameter is set to the value of the argument and then the *body* of the function is executed
- In this case (but not always) it explicitly returns a value
  - The *value of the function*

What type *is the parameter?*

- In many languages we have to specify the parameter *type*
  - e.g. specify whether we are squaring *int*s or *float*s
  - That restricts the allowable argument types
- Python has "Duck Typing"
  - "*If it walks like a duck and quacks like a duck, it's a duck*"
  - In this case: if the argument allows `x * x` it's OK
    - If not, it crashes *when we run it*
- So you can square *ints* and *floats*
  - And any other objects we might define that allows '*'
    - We'll do more on this later (week 10)

Another program example

```
def fahrenheit(degrees_c):
    degrees_f = (9.0 / 5.0) * degrees_c + 32.0
    return degrees_f


print(fahrenheit(0))   # What answer do we get?
print(fahrenheit(100)) # What answer here?
print(fahrenheit(451.0)) # And here?
print(fahrenheit("Fred")) # What does this do?
```

Note multiline body. All lines indented by same amount. Also note *local variable.*

- The above is a *program* in a separate file
- Now we can't just write expressions and have them "printed"
- We have to use the *print* function. Covered in detail later.

Local variables

- *degrees_f* is a “local variable” of the fahrenheit function
- Goes in a new dictionary belonging to that function
  - That dictionary exists only while the function is running
    - So variable disappears when function returns
- We say the *scope* of a local variable is the body of the function in which it is used
  - Scope is where a variable can be “seen” from

When do I use functions?

- Always!
- Programming is the art of breaking a problem into small "obviously correct" functions
  - "Divide and conquer"
- Each can be separately debugged
  - To "debug" is to remove the "bugs", i.e., errors, from a program
- Most functions should be less than 10 lines
- No function may be longer than 40 lines in COSC121
  - Break big functions into smaller functions

Don't expect to understand all this properly yet!

Function docstrings

```
def fahrenheit(degrees_c):
    """Converts a given fahrenheit temperature to celsius"""
    degrees_f = (9.0 / 5.0) * degrees_c + 32.0
    return degrees_f
```

- Every function should have a descriptive ‘docstring’ that starts and ends with `"""`
- Helps people read & understand programs
- Also provides online documentation: e.g., `help(fahrenheit)`

The two sorts of functions

Procedures

("Write a function that prints …")

- **Don't** return a value to caller
  - No return statement (in Python they implicitly return None)
- **Do** print output (or write files etc)
- Names start with a verb
  - *print_table*, *display_summary*
- Called as, e.g.
  - `print_table(names, marks)`
  - `display_summary(data)`

Real functions

("Write a function that returns …")

- **Do** return a value to caller
  - Must have a return statement
- **Don't** print output or write files (usually)
- Names are nouns
  - *standard_error*, *max_rainfall*
- Called as, e.g.,
  - `error = standard_error(data)`
  - `print(max_rainfall(data))`