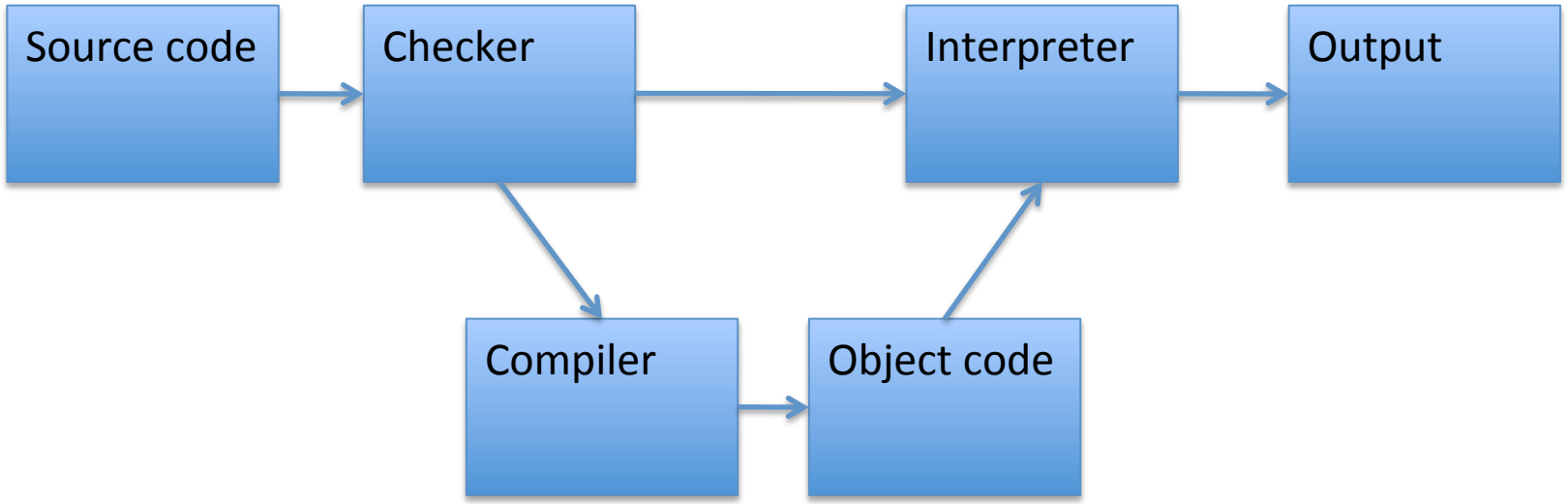
Programming languages

- Goal:
  - Need a way to describe algorithmic steps such that computer can use them to execute process
  - Programming language defines syntax and semantics needed to translate our computational ideas into mechanical steps

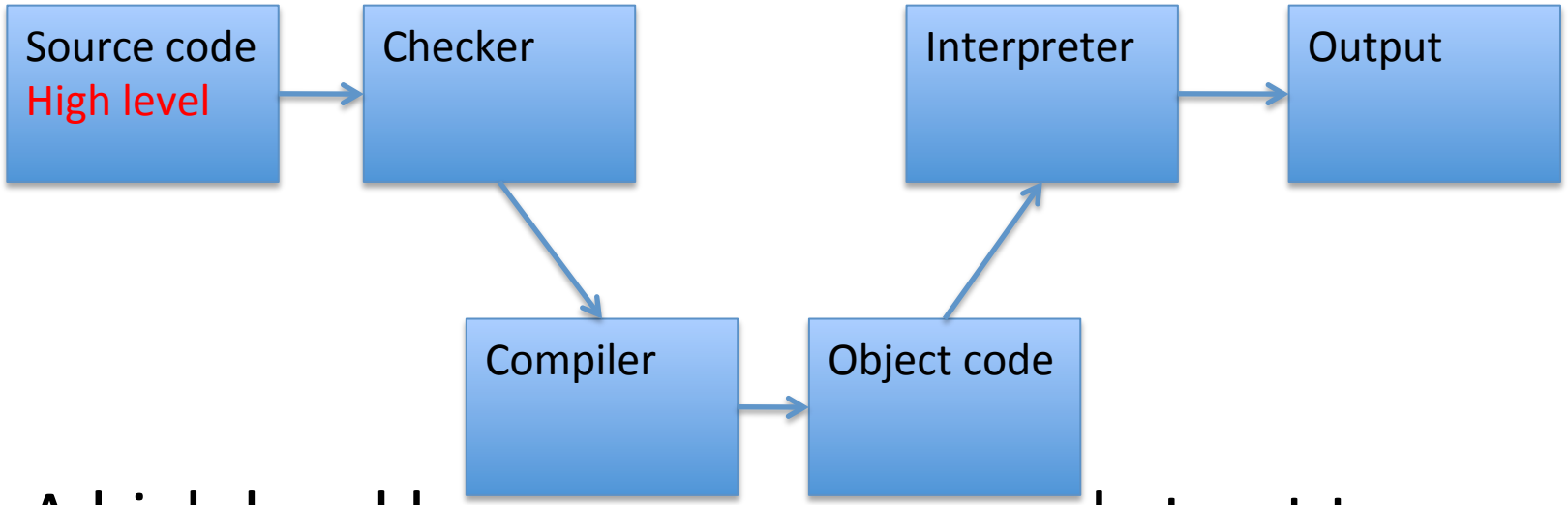
Options for programming languages



Options for programming languages

- Low level language uses instructions similar to internal control unit:
  - Move data from one location to another
  - Execute a simple ALU operation
  - Jump to new point in sequence based on test
- Checker confirms syntax, static semantics correct
- Interpreter just follows sequence of simple instructions

Options for programming languages



- A high level language uses more abstract terms – invert a matrix, compute a function
- In a compiled language, those abstractions are converted back into low level instructions, then executed

Options for programming languages

- In an interpreted language, special program converts source code to internal data structure, then interpreter sequentially converts each step into low level machine instruction and executes
- We are going to use Python, which belongs to this class of programming languages

Python programs

- Program (or script) is a sequence of **definitions** and **commands**
  - Definitions evaluated and commands executed by Python interpreter in a **shell**
  - Can be typed directly into a shell, or stored in a file that is read into the shell and evaluated
- **Command** (or **statement)** instructs interpreter to do something

Objects

- At heart, programs will manipulate data objects
- Each object has a **type** that defines the kinds of things programs can do to it
- Objects are:
  - **Scalar** (i.e. cannot be subdivided), or
  - **Non-scalar** (i.e. have internal structure that can be accessed)

Scalar objects

- `int` – used to represent integers, e.g., `5` or `10082`
- `float` – used to represent real numbers, e.g., `3.14` or `27.0`
- `bool` – used to represent Boolean values `True` and `False`
- The built in Python function `type` returns the type of an object

```
>>> type(3)
<type 'int'>
>>> type(3.0)
<type 'float'>
```

Expressions

- Objects and operators can be combined to form **expressions**, each of which denotes an object of some type
- The syntax for most simple expressions is:
  - <object> <operator> <object>

Operators on `ints` and `floats`

- `i + j` – sum – if both are `ints`, result is `int`, if either is `float`, result is `float`
- `i - j` – difference
- `i * j` – product
- `i/j` – division – if both are `ints`, result is `int`, representing quotient without remainder
- `i%j` – remainder
- `i**j` – `i` raised to the power of `j`

Some simple examples

```
>>> 3 + 5
8
>>> 3.14 * 20
62.8
>>> (2 + 3)*4
20
>>> 2 + 3*4
14
```

Performing simple operations

- Parentheses define sub-computations – complete these to get values before evaluating larger expression
  - (2+3)*4
  - 5*4
  - 20
- Operator precedence:
  - In the absence of parentheses (within which expressions are first reduced), operators are executed left to right, first using **, then * and /, and then + and –

Comparison operators on `ints` and `floats`

- `i > j` – returns `True` if strictly greater than
- `i >= j` – returns `True` if greater than or equal
- `i < j`
- `i <= j`
- `i == j` – returns `True` if equal
- `i != j` – returns `True` if not equal

Operators on `bool`s

- `a and b` is `True` if both are `True`
- `a or b` is `True` if at least one is `True`
- `not a` is `True` if `a` is `False`; it is `False` if `a` is `True`

Type conversions (type casting)

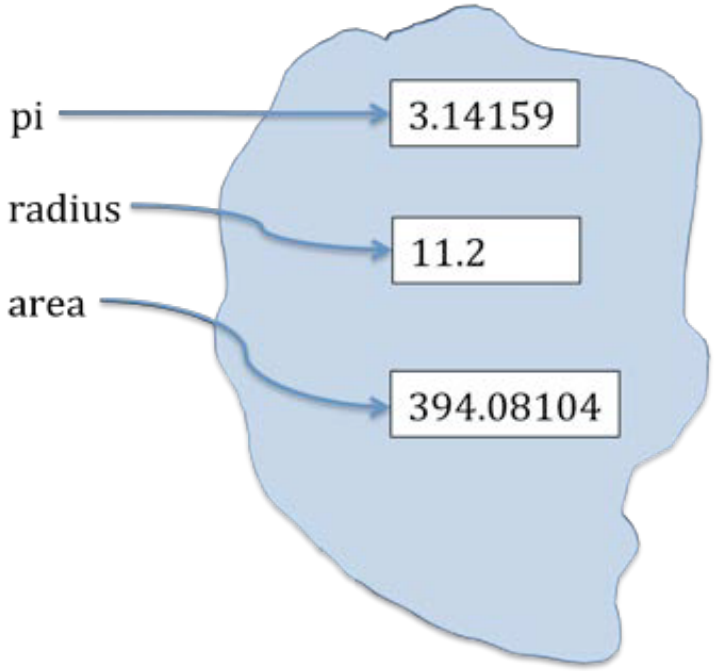
- We can often convert an object of one type to another, by using the name of the type as a function
  - `float(3)` has the value of `3.0`
  - `int(3.9)` truncates to `3`

Simple means of abstraction

- While we can write arbitrary expressions, it is useful to give names to values of expressions, and to be able to reuse those names in place of values
- `pi = 3.14159`
- `radius = 11.2`
- `area = pi * (radius**2)`

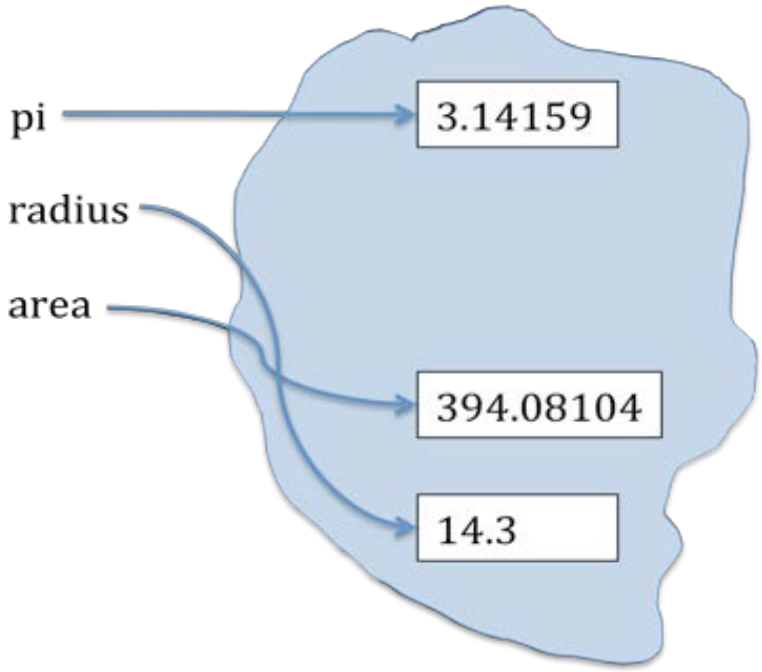
Binding variables and values

- The statement `pi = 3.14159` assigns the name `pi` to the value of the expression to the right of the =
- Think of each assignment statement as creating a binding between a name and a value stored somewhere in the computer
- We can retrieve the value associated with a name or variable by simply invoking that name, e.g., `pi`



Changing bindings

- Variable names can be rebound, by invoking new assignments statements.
- For example, if we now execute:
  - `radius = 11.2`
- we get the diagram shown here.
- Note that this doesn't change the value associated with `area`



Non-scalar objects

- We will see many different kinds of compound objects
- The simplest of these are strings, objects of type `str`
- Literals of type string can be written using single or double quotes
  - `'abc'`
  - `"abc"`
  - `'123'` – this is a string of characters, not the number

Operators on strings

```
>>> 3 * ‘a’
‘aaa’
>>> ‘a’ + ‘a’
‘aa’
>>> ‘a’ + str(3)
‘a3’
>>> len(‘abc’)
3
```

Extracting parts of strings

- Indexing:
  - `'abc'[0]` returns the string `'a'`
  - `'abc'[2]` returns the string `'c'`
  - `'abc'[3]` is an error (as we cannot go beyond the boundaries of the string)
  - `'abc'[-1]` returns the string `'c'` (essentially counting backwards from the start of the string)
- Slicing:
  - If `s` is a string, the expression `s[start:end]` denotes the substring that starts at `start`, and ends at `end-1`
    - `'abc'[1:3]` has the value `'bc'`

Programs (or scripts)

- While we can type expressions directly to a Python interpreter (for example using an interface such as an IDLE shell), in general we will want to include statements in a program file
- Executing an expression from a script will not produce any output; for that we need statements (not expressions), such as
  - `print('ab')`
  - `print('3'*3)`

Providing input

- If we are going to write programs or scripts, we will need a way to incorporate input from a user.
- We use the Python function `raw_input`, as in:

```
>>> name = raw_input('Enter your name: ')
Enter your name: Eric Grimson
>>> print('Are you ' + name + '?')
Are you Eric Grimson?
```

Some simple code

One can use variable names anywhere you might use the expression whose value it holds

```
>>> myString = 'Too much'
>>> weather = 'snow'
>>> print(myString + ' ' +
  weather)
Too much snow
```

A straight line program

Suppose we type the following into a file, and load it into a Python IDLE window

```
x = 3
x = x*x # square value of x
print(x)
y = float(raw_input('Enter a number: '))
print(y*y)
```

Then we observe the following behavior (where I type a 4 below)

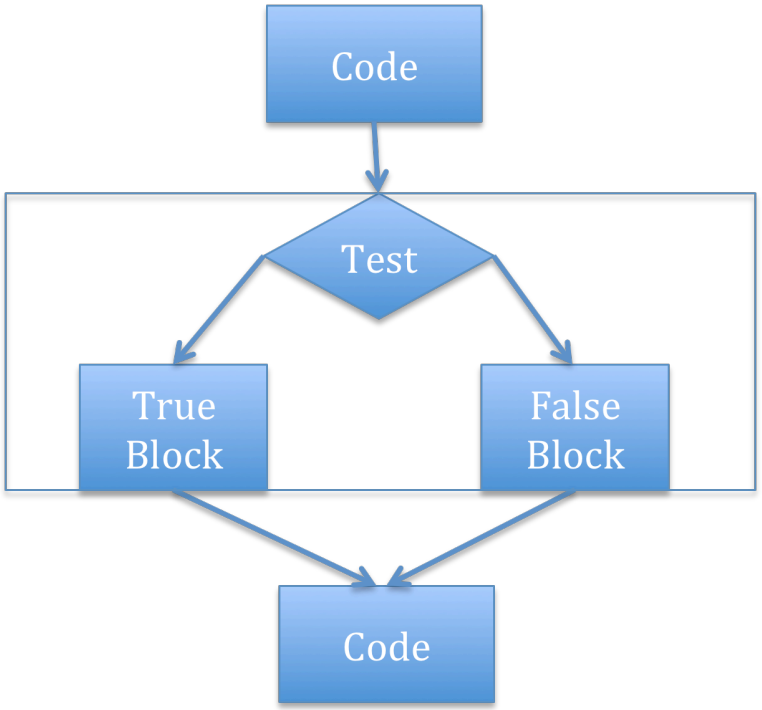
```
9
Enter a number: 4
16.0
```

Some observations

- Comments appear after a #
  - These are very valuable, as they help a user understand decisions the programmer has made in creating the program
  - Well commented code should be very readable by a user
- A straight line program simply executes each statement in order, with no variation in order
- Most programs require more sophisticated flow control

Branching programs

- The simplest branching statement is a **conditional**
  - A test (expression that evaluates to `True` or `False`)
  - A block of code to execute if the test is `True`
  - An optional block of code to execute if the test is `False`



A simple example

```
x = int(raw_input('Enter an integer: '))
if x%2 == 0:
    print(‘’)
    print('Even')
else:
    print(‘’)
    print('Odd')
print(’Done with conditional')
```

Some observations

- The expression `x%2 == 0` evaluates to `True` when the remainder of `x` divided by `2` is 0
- Note that `==` is used for comparison, since `=` is reserved for assignment
- The indentation is important – each indented set of expressions denotes a block of instructions
  - For example, if the last statement were indented, it would be executed as part of the `else` block of code
- Note how this indentation provides a visual structure that reflects the semantic structure of the program

We can have nested conditionals

```
if x%2 == 0:
    if x%3 == 0:
        print('Divisible by 2 and 3’)
    else:
        print('Divisible by 2 and not by 3’)
elif x%3 == 0:
    print('Divisible by 3 and not by 2’)
```

And we can use compound Booleans

```
if x < y and x < z:
    print('x is least')
elif y < z:
    print('y is least')
else:
    print('z is least')
```

What have we added?

- Branching programs allow us to make choices and do different things
- But still the case that at most, each statement gets executed once.
- So maximum time to run the program depends only on the length of the program
- These programs run in **constant time**