Practice
Week 1 project solutions are posted

FOCUS ON FUNCTIONS
- Python built-in functions
- Functions are an executable data type
- Functions may need one or more arguments to execute (run)
- Python modules
- Python user-defined functions
- Defining vs. calling a function
- Call stack and activation records
- Function parameters and arguments

Week 1 Project Solutions are posted
Review carefully – ask questions!
Notice, for example,
- style guidelines: file header, white space around operators, white space between sections of code
- use uppercase variable names for constants (e.g. hours in a day)
- “scaffolding code” to assist code development
- integer and floating point division
- round operation (not quite what was needed, perhaps)
- errors (e.g., what happens if you leave the closing paren off the print statement?)

Python Primitive Elements
Variables and assignment

```python
>>> example = 88
>>> example
```

- a) 88
- b) 99
- c) syntax error
- d) none of these

```python
>>> type(example)
```

- a) <class 'int'>
- b) <class 'float'>
- c) <class 'str'>
- d) all of these

What will happen when each line of code is executed?

```python
>>> example = 99
>>> nextvar = example / 9
>>> nextvar
```

- a) 11
- b) 11.0
- c) syntax error
- d) none of these

```python
>>> nextvar = example // 9
>>> nextvar
```

- a) 11
- b) 11.0
- c) syntax error
- d) none of these

```python
a) 10 + 4
b) 10 - 4
c) 10 + .4
d) 10 + 4.0
e) -.7
f) +7
g) 10 + 4 / 2
h) (10 + 4) / 2
```
Welcome to CIS 122
Intro to Programming and Computational Problem Solving

What will happen when each line of code is executed?

a) \( \frac{2}{3} \)
b) \( \frac{2}{3} \)
c) \( \frac{1}{3} \times 15 \)
d) \( \frac{1}{3} \times 15 \)
e) \((10 + 4) \mod 2\)
f) \(10 + 4 \mod 2\)
g) \( \frac{123}{10} \)
h) \( \frac{123}{10} \)
i) \( 2^6 \)
j) \(-2^2\)
k) \((-2)^2\)
l) \(1+3 \times 5\)
m) \(2^6 \)  # challenge

Python Functions

A function names an operation.

Python has built-in functions and Standard Library functions and user-defined functions

Why functions?

Functions support abstraction, or reducing complexity by hiding unnecessary detail
Why functions?

Functions support **abstraction**, or reducing complexity by hiding unnecessary detail.

Organizing a program into functions supports

- program organization
- code testing
- program readability
- code re-use

Python Functions

A function names an operation.

- built in functions
- functions in the Python Standard Library modules
- user-defined functions

```python
>>> abs
<built-in function abs>
```

Functions are an executable data type

Calling a function

```python
>>> abs()
notice parentheses
```

```python
>>> abs(7)
supply the required argument
7
```
Python Functions

General form of a function call

\[ \text{<function name>}(\text{<arguments>}) \]

```python
>>> abs(0)
0
```

```python
>>> locals()
a function may not require an argument
```

```python
>>> globals()
still need to include parens to execute
```

```python
>>> print()
arguments may be optional
```

```python
>>> round(123.456)
```

```python
>>> round(123.456, 2)
```

Function calls are expressions – they are evaluated and return a value.

```python
>>> round(3.5)  
>>> round(4.5)
4
```

```python
>>> pow(3, 2)
9
```

Python Functions

```python
>>> y = 2.001
```

```python
>>> round(y)
2
```

```python
>>> pow(3, round(y))
```

```python
>>> type(round(y))
```

Functions can be composed

```python
>>> y = 3.14159
```

```python
>>> y = 3.14159
```

```python
>>> z = type(round(y))
```

```python
>>> z = type(y)
```

✓ Review week 1 project set solutions
✓ Midterm demo practice questions

FOCUS ON FUNCTIONS
✓ Python built-in functions
✓ Functions are an executable data type
✓ Functions may need one or more arguments to execute (run)
  • Python modules
  • Python user-defined functions
  • Defining vs. calling a function
  • Call stack and activation records
  • Function parameters and arguments
Python modules

collection of related variables and functions grouped together in a single file

Python standard library

collection of handy modules included with Python

to gain access to the variables and functions in a module, you have to import the module

>>> import math

>>> type(math)
<class 'module'>

>>> math
<module 'math' from '/Library/Frameworks/Python.framework/Versions/3.8/lib/python3.8/lib-dynload/math.cpython-38-darwin.so'>

EXPLORE!

>>> help(math)

>>> help(math.factorial)

>>> math.factorial(5)

>>> help(math.ceil)

>>> math.ceil(26.4)

import turtle

✓ Review week 1 project set solutions
✓ Midterm demo practice questions

FOCUS ON FUNCTIONS

✓ Python built-in functions
✓ Functions are an executable data type
✓ Functions may need one or more arguments to execute (run)
✓ Python modules and Python standard library
  • Python user-defined functions
  • Defining vs. calling a function
  • Call stack and activation records
  • Function parameters and arguments

Python User-defined Functions

Defining a function associates a name with a user-defined operation/process.

def add_one(y):
    z = y + 1
    print(z)
    return

defining a function associates a name with a user-defined operation/process.
Defining a function associates a name with a user-defined operation/process.

```python
x = 99 >>> x
99
def add_one(y):
    >>> add_one
    <function add_one at 0x103dec680>
z = y + 1
print(z)
return
```

Function docstring – multiline comment directly after the function header – helpful documentation – integrated with development environment

```python
def add_one(y):
    '''
    increases value of y by 1 and prints the result
    brief description
    example of use
    (more later)
    '''
z = y + 1
print(z)
return
```

As with built-in functions, a user-defined function must be called by name to execute:

```python
>>> add_one(4)
5
>>> add_one() >>> help(add_one)
??
```

✓ Review week 1 project set solutions
✓ Midterm demo practice questions

FOCUS ON FUNCTIONS
✓ Python built-in functions
  ✓ Functions are an executable data type
  ✓ Functions may need one or more arguments to execute (run)
✓ Python modules and Python standard library
✓ Python user-defined functions (def, return, docstring)
✓ Defining vs. calling a function
  • Function parameters and arguments
  • What happens when a function is called – stack frame diagram

```python
>>> def add_one(y):
    z = y + 1
    print(z)
    return

y is a parameter
```
Function Parameters and Arguments

```python
>>> def add_one(y):
    z = y + 1
    print(z)
    return

y is a parameter
specifies input needed for function to run (execute)
```

4 and 99 are arguments — input values specified when the function is called

```python
>>> add_one(4)
4
>>> add_one(99)
99
```

 parameters (formal parameters) are variable names supplied when the function is defined.
arguments (actual parameters) are the values supplied when the function is called.

**Python — “call by assignment” parameter passing:**
parameter name = argument value when the function is called.

```python
def add_one(y):
    z = y + 1
    print(z)
    return

>>> add_one(4)
4
>>> add_one(99)
99
```

4 is an argument
99 is an argument

```python
>>> add_one(4)
4
>>> add_one(99)
99
```
Function Parameters and Arguments

```python
>>> def add_one(y):
    z = y + 1
    print(z)
    return

>>> x = 101
>>> add_one(x)
??

>>> y = 200
>>> add_one(y)
??
```

What happens when a function is called
When a function is called/executed/run, Python:

- Creates a frame (activation record) on a call stack to keep track of the function’s local variables and other information needed to execute the function
- Evaluates arguments left to right
- Assigns argument values to parameters
- Executes the function body until return statement
- Frame/activation record is (eventually) discarded
- Processing resumes where function was called
What happens when a function is called

```python
>>> def add_one(y):
    __main__ (global frame)
    z = y + 1
    print(z)
    return

>>> add_one(101)
arg is eval'd: 101
y = 101
execute the function until return
z = y + 1
```

CIS 122 Intro to Programming and Computational Problem Solving
What happens when a function is called

```python
>>> def add_one(y):
    global frame (__main__)
    z = y + 1
    print(z)
    return

>>> add_one(101)
✓ arg is eval’d: 101
✓ y = 101
✓ z = y + 1
y and z are local variables
```

Function Parameters and Arguments

```python
>>> def add_one(y):
    z = y + 1
    print(z)
    return

>>> x = -99
>>> add_one(x)
??
```

Exercise: draw the call stack diagram

```python
>>> def add_one(y):
    z = y + 1
    print(z)
    return

>>> y = 200
>>> add_one(y)
??
>>> y
??
```
What happens when a function is called/parameter passing

```python
def add_one(y):
    global frame (__main__)
    z = y + 1
    print(z)
    return

>>> y = 200
>>> add_one(y)
201
```

What happens when a function is called/parameter passing

```python
def add_one(y):
    return y + 1

>>> y = 200
>>> add_one(y)
201
```

What happens when a function is called/parameter passing

```python
def cube_it(n):
    n ** 3
    print(n)
    return

>>> cube_it(2)
4
>>> n = 4
>>> cube_it(n)
64
```

Exercise: draw the call stack diagram

Sum: Defining a function

```python
def add_one(y):
    function header keyword; function name; parms; colon
defining function
docstring increases value of y by 1 and prints the result
example of use
100 increases value of y by 1 and prints the result
function body
print(z)
return return keyword indicates end of function
```
What is the result when this code is executed?
(Draw the call stack diagram.)

```python
def calculate(w, x):
    a = x + 2
    b = w + 1
    print(a + b + 3)
    return
```

>>> calculate
>>> calculate(1)
>>> calculate(1, 5)
>>> w = 2
>>> calculate(w, 0)

67

What is the result after this code is executed?

```python
>>> def calculate(w, x):
    w = 1  # a common bug
    x = 5
    a = x + 2
    b = w + 1
    print(a + b + 3)
    return
```

>>> calculate(1, 5)
>>> calculate(2, 0)

68

What will happen when each line of code is executed?

(a) myvar = 4 + 3  (i) my-var = 10 // 3
(b) def = 1.0   (j) my-var = 10 % 3
(c) round = 5  
(d) my_var = 5 * 4  (k) myvar
(e) my_var2 = 'hi' + 4  (l) round
(f) 2my_var = 99 + 1.0  (m) my_var
(g) my_var_2 = 1 + 2.5  (n) my_var_2
(h) _ = 101  (o) _

69

Which are executable data types in Python?

a) int  b) float  c) string  d) function  e) both int and function

70

What will happen when each line of code is executed?

```python```
>>> x = 0
>>> x = x + 1
>>> x = x + 2
>>> x

(hint: draw the reference diagram)

71

72
Given the following Python code:

```python
1 >>> x = -2
2 >>> abs(x)
3 >>> x = round(abs(x))
4 >>> y = x / 2
5 >>> type(y)
```

1) a) what is the value of `x` at line 2?
   b) what is printed in the Shell at line 4?
   c) what is the value of `x` at line 4?
   d) what is the value of `x` at line 6?

```python
def q1(anum):
    def midterm_function:
        def q1(anum):
            print(anum * 100)
            return
    ...
```

1) a) what is the name of this function?
   b) `q1`
   c) `anum`
   d) midterm function
   e) none of these

2) a) def
   b) q1
   c) anum
   d) midterm function
   e) none of these

What is the result when `q1(3)` is executed?

```
HPD = 24  # hours per day
DPW = 7   # days per week
week_total = HPD * DPW
print('There are', week_total, 'hours in a week.')
```

1) a) NameError
   b) TypeError
   c) SyntaxError
   d) 168
   e) There are 168 hours in a week.

REFERENCES

- A structured approach to computational problem solving
- Python toolkit so far
- Programming concepts so far
- CIS 122 is a community of learners...
- What can you expect in CIS 122?
- Suggested time management for CIS 122
- CIS 122 learning outcomes
Welcome to CIS 122
Intro to Programming and Computational Problem Solving

A Structured Approach to Computational Problem Solving

**TASK/PROBLEM** ↔ Computational Thinking ↔
**EXAMPLES/DIAGRAMS/WORDS** ...
**DEVELOP AN ALGORITHM/COMPUTATIONAL PROCESS**
**ALGORITHM/COMPUTATIONAL PROCESS** ↔ Design/Coding ↔
**CODE SNIPPETS/AUX. FUNCTIONS/...**
**DEVELOP A COMPUTER PROGRAM**
**COMPUTER PROGRAM** ↔ Testing/Debugging ↔
**CONTINUE DEVELOPING A**
**HIGH QUALITY COMPUTER PROGRAM** → Execute →
**HIGH QUALITY COMPUTER PROGRAM** → Automatic, fast, reliable, reusable
**COMPUTER-GENERATED SOLUTION TO TASK**

Welcome to CIS 122
Intro to Programming and Computational Problem Solving

CIS 122
Python toolkit so far
- numeric data types (int, float) and operations (e.g., +, -, *, /, %, round, pow, abs)
- string data type and operations ("", ",
- variables (identifiers)
- assignment statement — associates an identifier (variable name) with an object
- reference diagrams — keep track of identifiers, which must be unique
- expressions — combination of values, variables, and operations; a value or variable all by itself is an expression
- user-defined functions — def, docstring, return
- built-in functions — print, type, help, eval, exit

Python standard library — math
- IDLE interactive development environment to support Python coding
- IDE Shell is a REPL (read-evaluate-print loop) — code is executed immediately, returned value, if any, is automatically printed in the shell

Welcome to CIS 122
Intro to Programming and Computational Problem Solving

CIS 122 is a community of learners where
- **Everyone is welcome**
- **Everyone is respected**
- **We value intellectual challenges and deliberate practice in pursuit of new knowledge and skills**
- **We support and encourage each other**
- **We celebrate our own and each other’s accomplishments**

Welcome to CIS 122
Intro to Programming and Computational Problem Solving

What can you expect in CIS 122?
- ✓ Weekly projects and exercises to support learning of computational problem solving in a variety of areas
- Supported by
  - ✓ Class — large group — big picture concepts, exercises, Q/A
  - ✓ Lab — small group — exercises, computers/whiteboards, Q/A
  - ✓ Class notes (posted), project solutions (posted), text readings — review, practice, explore
  - ✓ Help hours — daily drop-in help, tutorials, code reviews
  - ✓ Comprehensive assessments — 2 midterm + 1 final “demo”
  - ✓ Individual feedback on weekly projects

Welcome to CIS 122
Intro to Programming and Computational Problem Solving

Welcome to CIS 122
- text, projects, labs 1st reading/preparation — 2
- scheduled class/lab — 4
- weekly computational problem solving projects — understand/solve/design/code/test&debug and help hour(s) — 4+
- review text/class & lab notes/project solutions — 2
CIS 122 Learning Outcomes

• use a computational problem-solving approach to generate computer solutions (programs) to a variety of problems

• write well-structured, well-documented programs using the Python programming language

• be prepared to continue to study programming or computer science on your own or in other courses