• Intro to Strings
• Errors, Testing, and Debugging
• Short Review of What Happens When a Function is Called (Parameter Passing, Local Variables, and Return)

Recall: Python objects (primitive elements)
• type (constrains the range of allowable values and operations)
• value(s)
• [memory location]

```
int          function
float        Turtle
bool         NoneType
str
```

Another data type – Strings
```
>>> 'Hello, World'
'Hello, World'

>>> print('Hello, World')
Hello, World

>>> answer = input('What is your name? ')

>>> type(answer)
<class 'str'>
```

Another data type - Strings
```
>>> 'Hello, World' >>> "Hello, World"
'Hello, World' 'Hello, World'

>>> "Hello, Duck's World"
"Hello, Duck's World"
```

Another data type - Strings
```
>>> h = 'Johnson\tUnthank\tFriendly\nHall'

>>> print(h)
Johnson    Unthank    Friendly
Hall
```

CIS 122 Intro to Programming and Computational Problem Solving
Another data type - Strings

```python
>>> amount = 99.12345
>>> f'${amount:.2f}'
'$99.12'
```

```python
>>> formula = '2\u03C0r'
>>> print(formula)
2\pi r
>>> '\u03C0'
# UTF-8
'π'
# (encode characters # as 0s and 1s)
>>> 'π'
# (encode characters

```

```python
>>> print(type(97403))
??
>>> print(type(True))
??
>>> print(type(Lincoln))
??
```

```
>>> '97403'
'97403'
>>> type('97403')
<class 'str'>
>>> type(True)
<class 'bool'>
>>> type(False)
<class 'bool'>
>>> type('Lincoln')
<class 'str'>
```

```
>>> 'GOOD MORNING, CIS122'
# string literal evaluates
'GOOD MORNING, CIS122'
# to itself
>>> s1 = 'GOOD MORNING, CIS122'
>>> type(s1)
<class 'str'>
>>> s1
'GOOD MORNING, CIS122'
# can be assigned
>>> type('GOOD MORNING, CIS122')
<class 'str'>
```

```
>>> len(s1)
20
>>> isinstance(s1, str)
True
>>> isinstance(s1, int)
False
```

```
NameError: name 'Lincoln' is not defined
```
Strings are sequences of characters.

```python
>>> x = 'PYTHON ROCKS'
>>> x[0]     >>> x[1]     >>> x[4]
'P'        'Y'        'O'
>>> x[-1]   >>> x[-2]
'S'        'K'
>>> len(x)  >>> x[len(x)]
12         IndexError: string index out of range
```
✓ Intro to Strings

• Errors, Testing, and Debugging

• Short Review of What Happens When a Function is Called (Parameter Passing, Local Variables, and Return)

Even more than the act of testing, the act of designing tests is one of the best bug preventers ...

GOAL: HIGH QUALITY COMPUTER PROGRAMS

REUSABLE/MAINTAINABLE

RELIABLE

ARE WE MEETING THIS GOAL?

GOAL: HIGH QUALITY COMPUTER PROGRAM

RELIABLE: program runs; produces correct output according to the specification

• program runs
• program results are correct
• program handles the unexpected gracefully
• program is fast and efficient
• program runs under extreme conditions
• ... meets other specifications, e.g., HCI/UX, platforms
GOAL: HIGH QUALITY COMPUTER PROGRAM
RELIABLE: program runs; produces correct output according to the specification

How do programs NOT meet this goal?

TYPES OF PROGRAMMING ERRORS
• syntax - program language keywords, grammar
  → program won’t run at all
• runtime - TypeError, NameError, IndexError, etc.
  → program starts to run and “crashes”

```python
def mybad(greeting):
    '''(greeting: str) -> str
    return first and last letters of a greeting
    >>> mybad('hello, world')
    'hd'
    begin_end $ greeting[1] + greeting[99]
    return begin_end
    >>> mybad('hello, world')
    Syntax Error – program won’t run
```
```python
def mybad(greeting):
    '''(greeting: str) -> str
    return first and last letters of a greeting
    >>> mybad('hello, world')
    'hd'
    begin_end = greeting[1] + greeting[99]
    return begin_end
    >>> mybad('hello, world')
    Traceback (most recent call last):
    File "~/kfreeman/Documents/cis122win22/w22-projects-solutions/test.py", line 33, in <module>
    File "~/kfreeman/Documents/cis122win22/w22-projects-solutions/test.py", line 30, in mybad
    NameError: name 'greeting' is not defined
    Runtime Error
    >>>

def mybad(greeting):
    '''(greeting: str) -> str
    return first and last letters of a greeting
    >>> mybad('hello, world')
    'hd'
    begin_end = greeting[1] + greeting[-1]
    return begin_end
    >>> mybad('hello, world')
    Traceback (most recent call last):
    File "~/kfreeman/Documents/cis122win22/w22-projects-solutions/test.py", line 33, in <module>
    File "~/kfreeman/Documents/cis122win22/w22-projects-solutions/test.py", line 30, in mybad
    IndexError: string index out of range
    Runtime error
    >>>

def mybad(greeting):
    '''(greeting: str) -> str
    return first and last letters of a greeting
    >>> mybad('hello, world')
    'hd'
    greeting = 'hello, world'
    begin_end = greeting[1] + greeting[-1]
    return begin_end
    >>> mybad('hello, world')
    'ed'
    Logic Error
    >>>

def mybad(greeting):
    '''(greeting: str) -> str
    return first and last letters of a greeting
    >>> mybad('hello, world')
    'hd'
    greeting = 'hello, world'
    begin_end = greeting[0] + greeting[-1]
    return begin_end
    >>> mybad('hello, world')
    OK?
    >>> mybad('hello, world')
    'ed'
    Logic Error
    >>> mybad()
    'hd'
```
GOAL: HIGH QUALITY COMPUTER PROGRAM
RELIABLE: program runs; produces correct output according to the specification

TYPES OF PROGRAMMING ERRORS
• syntax - program language keywords, grammar
• runtime - TypeError, NameError, IndexError, etc.
• logic/semantic - program runs and returns incorrect results

How can we detect logical errors?

Testing starts at program design
docstring:
(type contract)
brief description
basic examples of use
that reflect the project specification
automated testing (e.g., doctest.testmod)

A systematic approach to formulating testing goals:
– Simple/Basic examples
– Edge (boundary) conditions
– For different types of expected input
– For different types of expected output
CIS 122 Intro to Programming and Computational Problem Solving

Goal: reliable program that runs and results in correct output according to problem specification

Software engineering best practices:
- Style guidelines support development of reliable, reusable code (design, communication)
- Designing tests that can detect programming errors is an integral part of writing reliable code
- Automate testing to re-test every time you change your code in any way

From Testing to Debugging – Finding and Fixing Bugs
(Novice) programming → better way

Disengage from the task when trouble occurs
- Expect bugs; leave time for debugging

Neglect to track closely what programs do
- Know what output you are expecting

From Testing to Debugging – Finding and Fixing Bugs
(Novice) programming → better way

Try to repair bugs by haphazardly tinkering with code
- Repeat the failure; further testing
- Keep a copy of last working version

Have difficulty breaking problems down into parts suitable for separate chunks of code
- Good program design/keep functions small

Debugging – finding and fixing bugs

Concentrate on finding why the program is doing what it is doing (not why it isn’t doing what you want it to).

10 steps to debugging your code:

Create a bug log (on paper, and/or by talking out loud, to LA or anyone):
1) record steps to reproduce the bug
2) describe what you expect to happen
3) describe what does happen
4) generate an idea about what is going wrong
5) test your idea – record what happens
6) repeat 4) and 5) as needed
7) When you have a good idea of what the bug is - BACKUP THE CODE
8) edit code to try to fix the bug
9) RECORD these changes to the code
10) test changes – the bug fix and regression testing (add new tests)

Always keep a Backup Copy of your code!
Even more than the act of testing, the act of designing tests is one of the best bug preventers ...

```python
def is_even(n):
    '''
    result = (n % 2) == 0
    return result
def report_even(number):
    '''
    is_even()  # Bug
    if result:
        # Bug
        print(f'{number} is an even number.')  # Bug
    else:
        print(f'{number} is an odd number.')  # Bug
    return

>>> report_even(101)
```

```python
def is_even(n):
    '''
    result = (n % 2) == 0
    return result
def report_even(number):
    '''
    result = is_even(number)  # was NameError
    if result:
        # was NameError
        print(f'{number} is an even number.')
    else:
        print(f'{number} is an odd number.')
    return

>>> report_even(101)
```
### Welcome to CIS 122 Intro to Programming and Computational Problem Solving

**A Structured Approach to Computational Problem Solving**

**TASK/PREP** ➔ **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**

**HIGH QUALITY COMPUTER PROGRAM** ➔ **Execute** ➔ **AUTOMATIC, FAST, RELIABLE, REUSABLE COMPUTER-GENERATED SOLUTION TO TASK**

### CIS 122 Programming concepts so far

- Intro to Strings
- Errors, Testing, and Debugging
- Short Review of What Happens When a Function is Called (Parameter Passing, Local Variables, and Return)

Even more than the act of testing, the act of designing tests is one of the best bug preventers ...

### CIS 122

#### 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

### CIS 122

#### Python toolkit so far

- numeric data types (int, float) and operations (e.g., +, -, /, **, round, pow, abs)
- string data type and operations (+, *, str, index (string are sequences))
- Boolean data types and operations (relational, logical)
- variables (identifiers);
- assignment statement – associates an identifier (variable name) with an object
- expressions – combination of values, variables, and operators; a value or variable all by itself is an expression
- user-defined functions – def, docstring, return
- more built-in functions – print, type, help, exit, input, int
- Python doctest (stdlib examples) and user-defined tests (for loop iteration)
- Python for is a sequential operator
- Python-conditional statements (if, else)
- reference diagrams – keep track of identifiers, which must be unwrapping stack diagrams – keep track of function calls and local variables (PyTutor/Visualizer)
- import
- Python standard library – math, turtle, random, datetime
- IDLE interactive development environment to support Python coding
- IDLE: a REPL (read-eval-print loop) – code is executed immediately, returned value, if any, is automatically printed in the shell
- Errors: SyntaxError, NameError, TypeError, IndentationError

### CIS 122

#### Python-conditional statements (if, else)

- reference diagrams – keep track of identifiers, which must be unwrapping stack diagrams – keep track of function calls and local variables (PyTutor/Visualizer)

- import
- Python standard library – math, turtle, random, datetime
- IDLE interactive development environment to support Python coding
- IDLE: a REPL (read-eval-print loop) – code is executed immediately, returned value, if any, is automatically printed in the shell

Errors: SyntaxError, NameError, TypeError, IndentationError

### CIS 122

#### 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
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

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